Gröbner Bases Theory

Fabian Immler and Alexander Maletzky*

March 19, 2025

Abstract

This formalization is concerned with the theory of Gröbner bases in (commutative) multivariate polynomial rings over fields, originally developed by Buchberger in his 1965 PhD thesis. Apart from the statement and proof of the main theorem of the theory, the formalization also implements algorithms for actually computing Gröbner bases, thus allowing to effectively decide ideal membership in finitely generated polynomial ideals. Furthermore, all functions can be executed on a concrete representation of multivariate polynomials as association lists.

Contents

*Supported by the Austrian Science Fund (FWF): grant no. W1214-N15 (project DK1) and grant no. P 29498-N31

1 Introduction

The theory of Gröbner bases, invented by Buchberger in [2, 3], is ubiquitous in many areas of computer algebra and beyond, as it allows to effectively solve a multitude of interesting, non-trivial problems of polynomial ideal theory. Since its invention in the mid-sixties, the theory has already seen a whole range of extensions and generalizations, some of which are present in this formalization:

- Following [11], the theory is formulated for vector-polynomials instead of ordinary scalar polynomials, thus allowing to compute Gröbner bases of syzygy modules.
- Besides Buchberger's original algorithm, the formalization also features Faugère's F_4 algorithm [8] for computing Gröbner bases.
- All algorithms for computing Gröbner bases incorporate criteria to avoid useless pairs; see [4] for details.
- Reduced Gröbner bases have been formalized and can be computed by a formally verified algorithm, too.

For further information about Gröbner bases theory the interested reader may consult the introductory paper [5] or literally any book on commutative/computer algebra, e. g. [1, 11].

1.1 Related Work

The theory of Gröbner bases has already been formalized in a couple of other proof assistants, listed below in alphabetical order:

- ACL2 [13],
- Coq [16, 10],
- Mizar [15], and
- Theorema [6, 12].

Please note that this formalization must not be confused with the *algebra* proof method based on Gröbner bases [7], which is a completely independent piece of work: our results could in principle be used to formally prove the correctness and, to some extent, completeness of said proof method.

1.2 Future Work

This formalization can be extended in several ways:

- One could formalize signature-based algorithms for computing Gröbner bases, as for instance Faugère's F_5 algorithm [9]. Such algorithms are typically more efficient than Buchberger's algorithm.
- One could establish the connection to *elimination theory*, exploiting the well-known *elimination property* of Gröbner bases w. r. t. certain term-orders (e. g. the purely lexicographic one). This would enable the effective simplification (and even solution, in some sense) of systems of algebraic equations.
- One could generalize the theory further to cover also *non-commutative* Gröbner bases [14].

2 General Utilities

theory *General*
  imports *Polynomials.Utils*
begin

A couple of general-purpose functions and lemmas, mainly related to lists.

2.1 Lists

lemma *distinct-reorder*: *distinct* (*xs* @ (*y* # *ys*)) = *distinct* (*y* # (*xs* @ *ys*)) **by** *auto*

lemma *set-reorder*: *set* (*xs* @ (*y* # *ys*)) = *set* (*y* # (*xs* @ *ys*)) **by** *simp*

lemma *distinctI*:
  assumes $\bigwedge i\ j.\ i < j \Longrightarrow i < \mathit{length}\ xs \Longrightarrow j < \mathit{length}\ xs \Longrightarrow xs\ !\ i \neq xs\ !\ j$
  shows *distinct xs*
  using *assms*
proof (*induct xs*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons x xs*)
  show *?case*
  proof (*simp*, *intro conjI*, *rule*)
    assume $x \in \mathit{set}\ xs$
    then obtain *j* **where** $j < \mathit{length}\ xs$ **and** $x = xs\ !\ j$ **by** (*metis in-set-conv-nth*)
    hence $\mathit{Suc}\ j < \mathit{length}\ (x \# xs)$ **by** *simp*
    have $(x \# xs)\ !\ 0 \neq (x \# xs)\ !\ (\mathit{Suc}\ j)$ **by** (*rule Cons(2)*, *simp*, *simp*, *fact*)
    thus *False* **by** (*simp add*: ‹$x = xs\ !\ j$›)

```
  next
    show distinct xs
    proof (rule Cons(1))
      fix i j
      assume i < j and i < length xs and j < length xs
      hence Suc i < Suc j and Suc i < length (x # xs) and Suc j < length (x #
xs) by simp-all
      hence (x # xs) ! (Suc i) ≠ (x # xs) ! (Suc j) by (rule Cons(2))
      thus xs ! i ≠ xs ! j by simp
    qed
  qed
qed

lemma filter-nth-pairE:
  assumes i < j and i < length (filter P xs) and j < length (filter P xs)
  obtains i' j' where i' < j' and i' < length xs and j' < length xs
    and (filter P xs) ! i = xs ! i' and (filter P xs) ! j = xs ! j'
  using assms
proof (induct xs arbitrary: i j thesis)
  case Nil
  from Nil(3) show ?case by simp
next
  case (Cons x xs)
  let ?ys = filter P (x # xs)
  show ?case
  proof (cases P x)
    case True
    hence *: ?ys = x # (filter P xs) by simp
    from ‹i < j› obtain j0 where j: j = Suc j0 using lessE by blast
    have len-ys: length ?ys = Suc (length (filter P xs)) and ys-j: ?ys ! j = (filter
P xs) ! j0
      by (simp only: * length-Cons, simp only: j * nth-Cons-Suc)
    from Cons(5) have j0 < length (filter P xs) unfolding len-ys j by auto
    show ?thesis
    proof (cases i = 0)
      case True
      from ‹j0 < length (filter P xs)› obtain j' where j' < length xs and **: (filter
P xs) ! j0 = xs ! j'
      by (metis (no-types, lifting) in-set-conv-nth mem-Collect-eq nth-mem set-filter)
      have 0 < Suc j' by simp
      thus ?thesis
        by (rule Cons(2), simp, simp add: ‹j' < length xs›, simp only: True *
nth-Cons-0,
          simp only: ys-j nth-Cons-Suc **)
    next
      case False
      then obtain i0 where i: i = Suc i0 using lessE by blast
      have ys-i: ?ys ! i = (filter P xs) ! i0 by (simp only: i * nth-Cons-Suc)
      from Cons(3) have i0 < j0 by (simp add: i j)
```

from *Cons*(*4*) **have** *i0* < *length* (*filter P xs*) **unfolding** *len-ys i* **by** *auto*
from - ‹*i0* < *j0*› *this* ‹*j0* < *length* (*filter P xs*)› **obtain** *i′ j′*
where *i′* < *j′* **and** *i′* < *length xs* **and** *j′* < *length xs*
and *i′*: *filter P xs* ! *i0* = *xs* ! *i′* **and** *j′*: *filter P xs* ! *j0* = *xs* ! *j′*
by (*rule Cons*(*1*))
from ‹*i′* < *j′*› **have** *Suc i′* < *Suc j′* **by** *simp*
thus *?thesis*
by (*rule Cons*(*2*), *simp add*: ‹*i′* < *length xs*›, *simp add*: ‹*j′* < *length xs*›,
simp only: *ys-i nth-Cons-Suc i′*, *simp only*: *ys-j nth-Cons-Suc j′*)
qed
next
case *False*
hence ∗: *?ys* = *filter P xs* **by** *simp*
with *Cons*(*4*) *Cons*(*5*) **have** *i* < *length* (*filter P xs*) **and** *j* < *length* (*filter P xs*) **by** *simp-all*
with - ‹*i* < *j*› **obtain** *i′ j′* **where** *i′* < *j′* **and** *i′* < *length xs* **and** *j′* < *length xs*
and *i′*: *filter P xs* ! *i* = *xs* ! *i′* **and** *j′*: *filter P xs* ! *j* = *xs* ! *j′*
by (*rule Cons*(*1*))
from ‹*i′* < *j′*› **have** *Suc i′* < *Suc j′* **by** *simp*
thus *?thesis*
by (*rule Cons*(*2*), *simp add*: ‹*i′* < *length xs*›, *simp add*: ‹*j′* < *length xs*›,
simp only: ∗ *nth-Cons-Suc i′*, *simp only*: ∗ *nth-Cons-Suc j′*)
qed
qed

lemma *distinct-filterI*:
assumes ⋀*i j*. *i* < *j* ⟹ *i* < *length xs* ⟹ *j* < *length xs* ⟹ *P* (*xs* ! *i*) ⟹ *P* (*xs* ! *j*) ⟹ *xs* ! *i* ≠ *xs* ! *j*
shows *distinct* (*filter P xs*)
proof (*rule distinctI*)
fix *i j*::*nat*
assume *i* < *j* **and** *i* < *length* (*filter P xs*) **and** *j* < *length* (*filter P xs*)
then obtain *i′ j′* **where** *i′* < *j′* **and** *i′* < *length xs* **and** *j′* < *length xs*
and *i*: (*filter P xs*) ! *i* = *xs* ! *i′* **and** *j*: (*filter P xs*) ! *j* = *xs* ! *j′* **by** (*rule filter-nth-pairE*)
from ‹*i′* < *j′*› ‹*i′* < *length xs*› ‹*j′* < *length xs*› **show** (*filter P xs*) ! *i* ≠ (*filter P xs*) ! *j* **unfolding** *i j*
proof (*rule assms*)
from ‹*i* < *length* (*filter P xs*)› **show** *P* (*xs* ! *i′*) **unfolding** *i*[*symmetric*] **using** *nth-mem* **by** *force*
next
from ‹*j* < *length* (*filter P xs*)› **show** *P* (*xs* ! *j′*) **unfolding** *j*[*symmetric*] **using** *nth-mem* **by** *force*
qed
qed

lemma *set-zip-map*: *set* (*zip* (*map f xs*) (*map g xs*)) = (λ*x*. (*f x*, *g x*)) ‘ (*set xs*)
proof −

have {(*map f xs* ! *i*, *map g xs* ! *i*) |*i*. *i* < *length xs*} = {(*f* (*xs* ! *i*), *g* (*xs* ! *i*)) |*i*. *i* < *length xs*}
proof (*rule Collect-eqI*, *rule*, *elim exE conjE*, *intro exI conjI*, *simp add*: *map-nth*, *assumption*,
elim exE conjE, *intro exI*)
fix *x i*
assume *x* = (*f* (*xs* ! *i*), *g* (*xs* ! *i*)) **and** *i* < *length xs*
thus *x* = (*map f xs* ! *i*, *map g xs* ! *i*) ∧ *i* < *length xs* **by** (*simp add*: *map-nth*)
qed
also have ... = (λ*x*. (*f x*, *g x*)) ‘ {*xs* ! *i* | *i*. *i* < *length xs*} **by** *blast*
finally show *set* (*zip* (*map f xs*) (*map g xs*)) = (λ*x*. (*f x*, *g x*)) ‘ (*set xs*)
by (*simp add*: *set-zip set-conv-nth*[*symmetric*])
qed

lemma *set-zip-map1*: *set* (*zip* (*map f xs*) *xs*) = (λ*x*. (*f x*, *x*)) ‘ (*set xs*)
proof −
have *set* (*zip* (*map f xs*) (*map id xs*)) = (λ*x*. (*f x*, *id x*)) ‘ (*set xs*) **by** (*rule set-zip-map*)
thus *?thesis* **by** *simp*
qed

lemma *set-zip-map2*: *set* (*zip xs* (*map f xs*)) = (λ*x*. (*x*, *f x*)) ‘ (*set xs*)
proof −
have *set* (*zip* (*map id xs*) (*map f xs*)) = (λ*x*. (*id x*, *f x*)) ‘ (*set xs*) **by** (*rule set-zip-map*)
thus *?thesis* **by** *simp*
qed

lemma *UN-upt*: (⋃*i*∈{*0*..<*length xs*}. *f* (*xs* ! *i*)) = (⋃*x*∈*set xs*. *f x*)
by (*metis image-image map-nth set-map set-upt*)

lemma *sum-list-zeroI′*:
assumes ⋀*i*. *i* < *length xs* ⟹ *xs* ! *i* = *0*
shows *sum-list xs* = *0*
proof (*rule sum-list-zeroI*, *rule*, *simp*)
fix *x*
assume *x* ∈ *set xs*
then obtain *i* **where** *i* < *length xs* **and** *x* = *xs* ! *i* **by** (*metis in-set-conv-nth*)
from *this*(*1*) **show** *x* = *0* **unfolding** ‹*x* = *xs* ! *i*› **by** (*rule assms*)
qed

lemma *sum-list-map2-plus*:
assumes *length xs* = *length ys*
shows *sum-list* (*map2* (+) *xs ys*) = *sum-list xs* + *sum-list* (*ys*::′*a*::*comm-monoid-add list*)
using *assms*
proof (*induct rule*: *list-induct2*)
case *Nil*
show *?case* **by** *simp*

```
next
  case (Cons x xs y ys)
  show ?case by (simp add: Cons(2) ac-simps)
qed

lemma sum-list-eq-nthI:
  assumes i < length xs and ⋀j. j < length xs ⟹ j ≠ i ⟹ xs ! j = 0
  shows sum-list xs = xs ! i
  using assms
proof (induct xs arbitrary: i)
  case Nil
  from Nil(1) show ?case by simp
next
  case (Cons x xs)
  have *: xs ! j = 0 if j < length xs and Suc j ≠ i for j
  proof −
    have xs ! j = (x # xs) ! (Suc j) by simp
    also have ... = 0 by (rule Cons(3), simp add: ‹j < length xs›, fact)
    finally show ?thesis .
  qed
  show ?case
  proof (cases i)
    case 0
    have sum-list xs = 0 by (rule sum-list-zeroI', erule *, simp add: 0)
    with 0 show ?thesis by simp
  next
    case (Suc k)
    with Cons(2) have k < length xs by simp
    hence sum-list xs = xs ! k
    proof (rule Cons(1))
      fix j
      assume j < length xs
      assume j ≠ k
      hence Suc j ≠ i by (simp add: Suc)
      with ‹j < length xs› show xs ! j = 0 by (rule *)
    qed
    moreover have x = 0
    proof −
      have x = (x # xs) ! 0 by simp
      also have ... = 0 by (rule Cons(3), simp-all add: Suc)
      finally show ?thesis .
    qed
    ultimately show ?thesis by (simp add: Suc)
  qed
qed
```

2.1.1 *max-list*

```
fun (in ord) max-list :: 'a list ⇒ 'a where
```

max-list (*x* # *xs*) = (*case xs of* [] ⇒ *x* | - ⇒ *max x* (*max-list xs*))

context *linorder*
begin

lemma *max-list-Max*: *xs* ≠ [] ⟹ *max-list xs* = *Max* (*set xs*)
  by (*induct xs rule*: *induct-list012*, *auto*)

lemma *max-list-ge*:
  assumes *x* ∈ *set xs*
  shows *x* ≤ *max-list xs*
proof −
  from *assms* **have** *xs* ≠ [] **by** *auto*
  from *finite-set assms* **have** *x* ≤ *Max* (*set xs*) **by** (*rule Max-ge*)
  also from ‹*xs* ≠ []› **have** *Max* (*set xs*) = *max-list xs* **by** (*rule max-list-Max*[*symmetric*])
  finally show *?thesis* **.**
qed

lemma *max-list-boundedI*:
  assumes *xs* ≠ [] **and** ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≤ *a*
  shows *max-list xs* ≤ *a*
proof −
  from *assms*(*1*) **have** *set xs* ≠ {} **by** *simp*
  from *assms*(*1*) **have** *max-list xs* = *Max* (*set xs*) **by** (*rule max-list-Max*)
  also from *finite-set* ‹*set xs* ≠ {}› *assms*(*2*) **have** . . . ≤ *a* **by** (*rule Max.boundedI*)
  finally show *?thesis* **.**
qed

end

2.1.2 *insort-wrt*

primrec *insort-wrt* :: (*'c* ⇒ *'c* ⇒ *bool*) ⇒ *'c* ⇒ *'c list* ⇒ *'c list* **where**
  insort-wrt - *x* [] = [*x*] |
  insort-wrt r x (*y* # *ys*) =
    (*if r x y then* (*x* # *y* # *ys*) *else y* # (*insort-wrt r x ys*))

lemma *insort-wrt-not-Nil* [*simp*]: *insort-wrt r x xs* ≠ []
  by (*induct xs*, *simp-all*)

lemma *length-insort-wrt* [*simp*]: *length* (*insort-wrt r x xs*) = *Suc* (*length xs*)
  by (*induct xs*, *simp-all*)

lemma *set-insort-wrt* [*simp*]: *set* (*insort-wrt r x xs*) = *insert x* (*set xs*)
  by (*induct xs*, *auto*)

lemma *sorted-wrt-insort-wrt-imp-sorted-wrt*:
  assumes *sorted-wrt r* (*insort-wrt s x xs*)
  shows *sorted-wrt r xs*

```
  using assms
proof (induct xs)
  case Nil
  show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof (cases s x a)
    case True
    with Cons.prems have sorted-wrt r (x # a # xs) by simp
    thus ?thesis by simp
  next
    case False
    with Cons(2) have sorted-wrt r (a # (insort-wrt s x xs)) by simp
    hence ∗: (∀ y∈set xs. r a y) and sorted-wrt r (insort-wrt s x xs)
      by (simp-all)
    from this(2) have sorted-wrt r xs by (rule Cons(1))
    with ∗ show ?thesis by (simp)
  qed
qed

lemma sorted-wrt-imp-sorted-wrt-insort-wrt:
  assumes transp r and ⋀a. r a x ∨ r x a and sorted-wrt r xs
  shows sorted-wrt r (insort-wrt r x xs)
  using assms(3)
proof (induct xs)
  case Nil
  show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof (cases r x a)
    case True
    with Cons(2) assms(1) show ?thesis by (auto dest: transpD)
  next
    case False
    with assms(2) have r a x by blast
    from Cons(2) have ∗: (∀ y∈set xs. r a y) and sorted-wrt r xs
      by (simp-all)
    from this(2) have sorted-wrt r (insort-wrt r x xs) by (rule Cons(1))
    with ‹r a x› ∗ show ?thesis by (simp add: False)
  qed
qed

corollary sorted-wrt-insort-wrt:
  assumes transp r and ⋀a. r a x ∨ r x a
  shows sorted-wrt r (insort-wrt r x xs) ⟷ sorted-wrt r xs (is ?l ⟷ ?r)
proof
  assume ?l
```

then show *?r* **by** (*rule sorted-wrt-insort-wrt-imp-sorted-wrt*)
next
assume *?r*
with *assms* **show** *?l* **by** (*rule sorted-wrt-imp-sorted-wrt-insort-wrt*)
qed

2.1.3 *diff-list* and *insert-list*

definition *diff-list* :: *'a list ⇒ 'a list ⇒ 'a list* (**infixl** ‹−−› *65*)
where *diff-list xs ys = fold removeAll ys xs*

lemma *set-diff-list*: *set* (*xs* −− *ys*) = *set xs* − *set ys*
by (*simp only*: *diff-list-def*, *induct ys arbitrary*: *xs*, *auto*)

lemma *diff-list-disjoint*: *set ys* ∩ *set* (*xs* −− *ys*) = {}
unfolding *set-diff-list* **by** (*rule Diff-disjoint*)

lemma *subset-append-diff-cancel*:
assumes *set ys* ⊆ *set xs*
shows *set* (*ys* @ (*xs* −− *ys*)) = *set xs*
by (*simp only*: *set-append set-diff-list Un-Diff-cancel*, *rule Un-absorb1*, *fact*)

definition *insert-list* :: *'a ⇒ 'a list ⇒ 'a list*
where *insert-list x xs* = (*if x* ∈ *set xs then xs else x* # *xs*)

lemma *set-insert-list*: *set* (*insert-list x xs*) = *insert x* (*set xs*)
by (*auto simp add*: *insert-list-def*)

2.1.4 *remdups-wrt*

primrec *remdups-wrt* :: (*'a ⇒ 'b*) ⇒ *'a list ⇒ 'a list* **where**
remdups-wrt-base: *remdups-wrt* - [] = [] |
remdups-wrt-rec: *remdups-wrt f* (*x* # *xs*) = (*if f x* ∈ *f* ‘ *set xs then remdups-wrt f xs else x* # *remdups-wrt f xs*)

lemma *set-remdups-wrt*: *f* ‘ *set* (*remdups-wrt f xs*) = *f* ‘ *set xs*
proof (*induct xs*)
case *Nil*
show *?case* **unfolding** *remdups-wrt-base* **..**
next
case (*Cons a xs*)
show *?case* **unfolding** *remdups-wrt-rec*
proof (*simp only*: *split*: *if-splits*, *intro conjI*, *intro impI*)
assume *f a* ∈ *f* ‘ *set xs*
have *f* ‘ *set* (*a* # *xs*) = *insert* (*f a*) (*f* ‘ *set xs*) **by** *simp*
have *f* ‘ *set* (*remdups-wrt f xs*) = *f* ‘ *set xs* **by** *fact*
also from ‹*f a* ∈ *f* ‘ *set xs*› **have** ... = *insert* (*f a*) (*f* ‘ *set xs*) **by** (*simp add*: *insert-absorb*)
also have ... = *f* ‘ *set* (*a* # *xs*) **by** *simp*
finally show *f* ‘ *set* (*remdups-wrt f xs*) = *f* ‘ *set* (*a* # *xs*) **.**

  qed (*simp add*: *Cons.hyps*)
qed

lemma *subset-remdups-wrt*: *set* (*remdups-wrt f xs*) $\subseteq$ *set xs*
  by (*induct xs*, *auto*)

lemma *remdups-wrt-distinct-wrt*:
  assumes $x \in$ *set* (*remdups-wrt f xs*) **and** $y \in$ *set* (*remdups-wrt f xs*) **and** $x \neq y$
  shows $f\ x \neq f\ y$
  using *assms*(*1*) *assms*(*2*)
proof (*induct xs*)
  case *Nil*
  thus *?case* **unfolding** *remdups-wrt-base* **by** *simp*
next
  case (*Cons a xs*)
  from *Cons*(*2*) *Cons*(*3*) **show** *?case* **unfolding** *remdups-wrt-rec*
  proof (*simp only*: *split*: *if-splits*)
    assume $x \in$ *set* (*remdups-wrt f xs*) **and** $y \in$ *set* (*remdups-wrt f xs*)
    thus $f\ x \neq f\ y$ **by** (*rule Cons.hyps*)
  next
    assume $\neg$ *True*
    thus $f\ x \neq f\ y$ **by** *simp*
  next
    assume $f\ a \notin f$ ‘ *set xs* **and** *xin*: $x \in$ *set* (*a* # *remdups-wrt f xs*) **and** *yin*: $y \in$ *set* (*a* # *remdups-wrt f xs*)
    from *yin* **have** *y*: $y = a \lor y \in$ *set* (*remdups-wrt f xs*) **by** *simp*
    from *xin* **have** $x = a \lor x \in$ *set* (*remdups-wrt f xs*) **by** *simp*
    thus $f\ x \neq f\ y$
    proof
      assume $x = a$
      from *y* **show** *?thesis*
      proof
        assume $y = a$
        with ‹$x \neq y$› **show** *?thesis* **unfolding** ‹$x = a$› **by** *simp*
      next
        assume $y \in$ *set* (*remdups-wrt f xs*)
        have $y \in$ *set xs* **by** (*rule*, *fact*, *rule subset-remdups-wrt*)
        hence $f\ y \in f$ ‘ *set xs* **by** *simp*
        with ‹$f\ a \notin f$ ‘ *set xs*› **show** *?thesis* **unfolding** ‹$x = a$› **by** *auto*
      qed
    next
      assume $x \in$ *set* (*remdups-wrt f xs*)
      from *y* **show** *?thesis*
      proof
        assume $y = a$
        have $x \in$ *set xs* **by** (*rule*, *fact*, *rule subset-remdups-wrt*)
        hence $f\ x \in f$ ‘ *set xs* **by** *simp*
        with ‹$f\ a \notin f$ ‘ *set xs*› **show** *?thesis* **unfolding** ‹$y = a$› **by** *auto*
      next

assume *y* ∈ *set* (*remdups-wrt f xs*)
with ‹*x* ∈ *set* (*remdups-wrt f xs*)› **show** *?thesis* **by** (*rule Cons.hyps*)
qed
qed
qed
qed

lemma *distinct-remdups-wrt*: *distinct* (*remdups-wrt f xs*)
proof (*induct xs*)
case *Nil*
show *?case* **unfolding** *remdups-wrt-base* **by** *simp*
next
case (*Cons a xs*)
show *?case* **unfolding** *remdups-wrt-rec*
proof (*split if-split*, *intro conjI impI*, *rule Cons.hyps*)
assume *f a* ∉ *f* ‘ *set xs*
hence *a* ∉ *set xs* **by** *auto*
hence *a* ∉ *set* (*remdups-wrt f xs*) **using** *subset-remdups-wrt*[*of f xs*] **by** *auto*
with *Cons.hyps* **show** *distinct* (*a* # *remdups-wrt f xs*) **by** *simp*
qed
qed

lemma *map-remdups-wrt*: *map f* (*remdups-wrt f xs*) = *remdups* (*map f xs*)
by (*induct xs*, *auto*)

lemma *remdups-wrt-append*:
remdups-wrt f (*xs* @ *ys*) = (*filter* (λ*a*. *f a* ∉ *f* ‘ *set ys*) (*remdups-wrt f xs*)) @ (*remdups-wrt f ys*)
by (*induct xs*, *auto*)

2.1.5 *map-idx*

primrec *map-idx* :: ('*a* ⇒ *nat* ⇒ '*b*) ⇒ '*a list* ⇒ *nat* ⇒ '*b list* **where**
map-idx f [] *n* = [] |
map-idx f (*x* # *xs*) *n* = (*f x n*) # (*map-idx f xs* (*Suc n*))

lemma *map-idx-eq-map2*: *map-idx f xs n* = *map2 f xs* [*n*..<*n* + *length xs*]
proof (*induct xs arbitrary*: *n*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
have *eq*: [*n*..<*n* + *length* (*x* # *xs*)] = *n* # [*Suc n*..<*Suc* (*n* + *length xs*)]
by (*metis add-Suc-right length-Cons less-add-Suc1 upt-conv-Cons*)
show *?case* **unfolding** *eq* **by** (*simp add*: *Cons del*: *upt-Suc*)
qed

lemma *length-map-idx* [*simp*]: *length* (*map-idx f xs n*) = *length xs*
by (*simp add*: *map-idx-eq-map2*)

lemma *map-idx-append*: *map-idx f* (*xs* @ *ys*) *n* = (*map-idx f xs n*) @ (*map-idx f ys* (*n* + *length xs*))
  by (*simp add*: *map-idx-eq-map2 ab-semigroup-add-class.add-ac*(*1*) *zip-append1*)

lemma *map-idx-nth*:
  assumes *i* < *length xs*
  shows (*map-idx f xs n*) ! *i* = *f* (*xs* ! *i*) (*n* + *i*)
  using *assms* **by** (*simp add*: *map-idx-eq-map2*)

lemma *map-map-idx*: *map f* (*map-idx g xs n*) = *map-idx* (λ*x i*. *f* (*g x i*)) *xs n*
  by (*auto simp add*: *map-idx-eq-map2*)

lemma *map-idx-map*: *map-idx f* (*map g xs*) *n* = *map-idx* (*f* ∘ *g*) *xs n*
  by (*simp add*: *map-idx-eq-map2 map-zip-map*)

lemma *map-idx-no-idx*: *map-idx* (λ*x* -. *f x*) *xs n* = *map f xs*
  by (*induct xs arbitrary*: *n*, *simp-all*)

lemma *map-idx-no-elem*: *map-idx* (λ-. *f*) *xs n* = *map f* [*n*..<*n* + *length xs*]
proof (*induct xs arbitrary*: *n*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons x xs*)
  have *eq*: [*n*..<*n* + *length* (*x* # *xs*)] = *n* # [*Suc n*..<*Suc* (*n* + *length xs*)]
    by (*metis add-Suc-right length-Cons less-add-Suc1 upt-conv-Cons*)
  show *?case* **unfolding** *eq* **by** (*simp add*: *Cons del*: *upt-Suc*)
qed

lemma *map-idx-eq-map*: *map-idx f xs n* = *map* (λ*i*. *f* (*xs* ! *i*) (*i* + *n*)) [*0*..<*length xs*]
proof (*induct xs arbitrary*: *n*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons x xs*)
  have *eq*: [*0*..<*length* (*x* # *xs*)] = *0* # [*Suc 0*..<*Suc* (*length xs*)]
    by (*metis length-Cons upt-conv-Cons zero-less-Suc*)
  have *map* (λ*i*. *f* ((*x* # *xs*) ! *i*) (*i* + *n*)) [*Suc 0*..<*Suc* (*length xs*)] =
        map ((λ*i*. *f* ((*x* # *xs*) ! *i*) (*i* + *n*)) ∘ *Suc*) [*0*..<*length xs*]
    by (*metis map-Suc-upt map-map*)
  also have ... = *map* (λ*i*. *f* (*xs* ! *i*) (*Suc* (*i* + *n*))) [*0*..<*length xs*]
    by (*rule map-cong*, *fact refl*, *simp*)
  finally show *?case* **unfolding** *eq* **by** (*simp add*: *Cons del*: *upt-Suc*)
qed

lemma *set-map-idx*: *set* (*map-idx f xs n*) = (λ*i*. *f* (*xs* ! *i*) (*i* + *n*)) ‘ {*0*..<*length xs*}

by (*simp add*: *map-idx-eq-map*)

2.1.6 *map-dup*

primrec *map-dup* :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a list* ⇒ '*b list* **where**
map-dup - - [] = []|
map-dup f g (*x* # *xs*) = (*if x* ∈ *set xs then g x else f x*) # (*map-dup f g xs*)

lemma *length-map-dup*[*simp*]: *length* (*map-dup f g xs*) = *length xs*
by (*induct xs*, *simp-all*)

lemma *map-dup-distinct*:
assumes *distinct xs*
shows *map-dup f g xs* = *map f xs*
using *assms* **by** (*induct xs*, *simp-all*)

lemma *filter-map-dup-const*:
filter (λ*x*. *x* ≠ *c*) (*map-dup f* (λ-. *c*) *xs*) = *filter* (λ*x*. *x* ≠ *c*) (*map f* (*remdups xs*))
by (*induct xs*, *simp-all*)

lemma *filter-zip-map-dup-const*:
filter (λ(*a*, *b*). *a* ≠ *c*) (*zip* (*map-dup f* (λ-. *c*) *xs*) *xs*) =
filter (λ(*a*, *b*). *a* ≠ *c*) (*zip* (*map f* (*remdups xs*)) (*remdups xs*))
by (*induct xs*, *simp-all*)

2.1.7 Filtering Minimal Elements

context
fixes *rel* :: '*a* ⇒ '*a* ⇒ *bool*
begin

primrec *filter-min-aux* :: '*a list* ⇒ '*a list* ⇒ '*a list* **where**
filter-min-aux [] *ys* = *ys*|
filter-min-aux (*x* # *xs*) *ys* =
(*if* (∃ *y*∈(*set xs* ∪ *set ys*). *rel y x*) *then* (*filter-min-aux xs ys*)
else (*filter-min-aux xs* (*x* # *ys*)))

definition *filter-min* :: '*a list* ⇒ '*a list*
where *filter-min xs* = *filter-min-aux xs* []

definition *filter-min-append* :: '*a list* ⇒ '*a list* ⇒ '*a list*
where *filter-min-append xs ys* =
(*let P* = (λ*zs*. λ*x*. ¬ (∃ *z*∈*set zs*. *rel z x*)); *ys1* = *filter* (*P xs*) *ys in*
(*filter* (*P ys1*) *xs*) @ *ys1*)

lemma *filter-min-aux-supset*: *set ys* ⊆ *set* (*filter-min-aux xs ys*)
proof (*induct xs arbitrary*: *ys*)
case *Nil*
show *?case* **by** *simp*

```
next
  case (Cons x xs)
  have set ys ⊆ set (x # ys) by auto
  also have set (x # ys) ⊆ set (filter-min-aux xs (x # ys)) by (rule Cons.hyps)
  finally have set ys ⊆ set (filter-min-aux xs (x # ys)) .
  moreover have set ys ⊆ set (filter-min-aux xs ys) by (rule Cons.hyps)
  ultimately show ?case by simp
qed

lemma filter-min-aux-subset: set (filter-min-aux xs ys) ⊆ set xs ∪ set ys
proof (induct xs arbitrary: ys)
  case Nil
  show ?case by simp
next
  case (Cons x xs)
  note Cons.hyps
  also have set xs ∪ set ys ⊆ set (x # xs) ∪ set ys by fastforce
  finally have c1: set (filter-min-aux xs ys) ⊆ set (x # xs) ∪ set ys .

  note Cons.hyps
  also have set xs ∪ set (x # ys) = set (x # xs) ∪ set ys by simp
  finally have set (filter-min-aux xs (x # ys)) ⊆ set (x # xs) ∪ set ys .
  with c1 show ?case by simp
qed

lemma filter-min-aux-relE:
  assumes transp rel and x ∈ set xs and x ∉ set (filter-min-aux xs ys)
  obtains y where y ∈ set (filter-min-aux xs ys) and rel y x
  using assms(2, 3)
proof (induct xs arbitrary: x ys thesis)
  case Nil
  from Nil(2) show ?case by simp
next
  case (Cons x0 xs)
  from Cons(3) have x = x0 ∨ x ∈ set xs by simp
  thus ?case
  proof
    assume x = x0
    from Cons(4) have *: ∃ y∈set xs ∪ set ys. rel y x0
    proof (simp add: ‹x = x0› split: if-splits)
      assume x0 ∉ set (filter-min-aux xs (x0 # ys))
       moreover from filter-min-aux-supset have x0 ∈ set (filter-min-aux xs (x0
# ys))
        by (rule subsetD) simp
      ultimately show False ..
    qed
    hence eq: filter-min-aux (x0 # xs) ys = filter-min-aux xs ys by simp
    from * obtain x1 where x1 ∈ set xs ∪ set ys and rel x1 x unfolding ‹x =
x0› ..
```

```
    from this(1) show ?thesis
    proof
      assume x1 ∈ set xs
      show ?thesis
      proof (cases x1 ∈ set (filter-min-aux xs ys))
        case True
        hence x1 ∈ set (filter-min-aux (x0 # xs) ys) by (simp only: eq)
        thus ?thesis using ‹rel x1 x› by (rule Cons(2))
      next
        case False
        with ‹x1 ∈ set xs› obtain y where y ∈ set (filter-min-aux xs ys) and rel
y x1
          using Cons.hyps by blast
        from this(1) have y ∈ set (filter-min-aux (x0 # xs) ys) by (simp only: eq)
       moreover from assms(1) ‹rel y x1› ‹rel x1 x› have rel y x by (rule transpD)
        ultimately show ?thesis by (rule Cons(2))
      qed
    next
      assume x1 ∈ set ys
      hence x1 ∈ set (filter-min-aux (x0 # xs) ys) using filter-min-aux-supset ..
      thus ?thesis using ‹rel x1 x› by (rule Cons(2))
    qed
  next
    assume x ∈ set xs
    show ?thesis
    proof (cases ∃ y∈set xs ∪ set ys. rel y x0)
      case True
      hence eq: filter-min-aux (x0 # xs) ys = filter-min-aux xs ys by simp
      with Cons(4) have x ∉ set (filter-min-aux xs ys) by simp
      with ‹x ∈ set xs› obtain y where y ∈ set (filter-min-aux xs ys) and rel y x
        using Cons.hyps by blast
      from this(1) have y ∈ set (filter-min-aux (x0 # xs) ys) by (simp only: eq)
      thus ?thesis using ‹rel y x› by (rule Cons(2))
    next
      case False
      hence eq: filter-min-aux (x0 # xs) ys = filter-min-aux xs (x0 # ys) by simp
      with Cons(4) have x ∉ set (filter-min-aux xs (x0 # ys)) by simp
      with ‹x ∈ set xs› obtain y where y ∈ set (filter-min-aux xs (x0 # ys)) and
rel y x
        using Cons.hyps by blast
      from this(1) have y ∈ set (filter-min-aux (x0 # xs) ys) by (simp only: eq)
      thus ?thesis using ‹rel y x› by (rule Cons(2))
    qed
  qed
qed

lemma filter-min-aux-minimal:
  assumes transp rel and x ∈ set (filter-min-aux xs ys) and y ∈ set (filter-min-aux
xs ys)
```

```
    and rel x y
  assumes ⋀a b. a ∈ set xs ∪ set ys ⟹ b ∈ set ys ⟹ rel a b ⟹ a = b
  shows x = y
  using assms(2−5)
proof (induct xs arbitrary: x y ys)
  case Nil
  from Nil(1) have x ∈ set [] ∪ set ys by simp
  moreover from Nil(2) have y ∈ set ys by simp
  ultimately show ?case using Nil(3) by (rule Nil(4))
next
  case (Cons x0 xs)
  show ?case
  proof (cases ∃ y∈set xs ∪ set ys. rel y x0)
    case True
    hence eq: filter-min-aux (x0 # xs) ys = filter-min-aux xs ys by simp
   with Cons(2, 3) have x ∈ set (filter-min-aux xs ys) and y ∈ set (filter-min-aux
xs ys)
      by simp-all
    thus ?thesis using Cons(4)
    proof (rule Cons.hyps)
      fix a b
      assume a ∈ set xs ∪ set ys
      hence a ∈ set (x0 # xs) ∪ set ys by simp
      moreover assume b ∈ set ys and rel a b
      ultimately show a = b by (rule Cons(5))
    qed
  next
    case False
    hence eq: filter-min-aux (x0 # xs) ys = filter-min-aux xs (x0 # ys) by simp
     with Cons(2, 3) have x ∈ set (filter-min-aux xs (x0 # ys)) and y ∈ set
(filter-min-aux xs (x0 # ys))
      by simp-all
    thus ?thesis using Cons(4)
    proof (rule Cons.hyps)
      fix a b
      assume a: a ∈ set xs ∪ set (x0 # ys) and b ∈ set (x0 # ys) and rel a b
      from this(2) have b = x0 ∨ b ∈ set ys by simp
      thus a = b
      proof
        assume b = x0
        from a have a = x0 ∨ a ∈ set xs ∪ set ys by simp
        thus ?thesis
        proof
          assume a = x0
          with ‹b = x0› show ?thesis by simp
        next
          assume a ∈ set xs ∪ set ys
          hence ∃ y∈set xs ∪ set ys. rel y x0 using ‹rel a b› unfolding ‹b = x0› ..
          with False show ?thesis ..
```

```
      qed
    next
      from a have a ∈ set (x0 # xs) ∪ set ys by simp
      moreover assume b ∈ set ys
      ultimately show ?thesis using ‹rel a b› by (rule Cons(5))
    qed
  qed
 qed
qed

lemma filter-min-aux-distinct:
 assumes reflp rel and distinct ys
 shows distinct (filter-min-aux xs ys)
 using assms(2)
proof (induct xs arbitrary: ys)
 case Nil
 thus ?case by simp
next
 case (Cons x xs)
 show ?case
 proof (simp split: if-split, intro conjI impI)
  from Cons(2) show distinct (filter-min-aux xs ys) by (rule Cons.hyps)
 next
  assume a: ∀ y∈set xs ∪ set ys. ¬ rel y x
  show distinct (filter-min-aux xs (x # ys))
  proof (rule Cons.hyps)
   have x ∉ set ys
   proof
    assume x ∈ set ys
    hence x ∈ set xs ∪ set ys by simp
    with a have ¬ rel x x ..
    moreover from assms(1) have rel x x by (rule reflpD)
    ultimately show False ..
   qed
   with Cons(2) show distinct (x # ys) by simp
  qed
 qed
qed

lemma filter-min-subset: set (filter-min xs) ⊆ set xs
 using filter-min-aux-subset[of xs []] by (simp add: filter-min-def)

lemma filter-min-cases:
 assumes transp rel and x ∈ set xs
 assumes x ∈ set (filter-min xs) ⟹ thesis
 assumes ⋀y. y ∈ set (filter-min xs) ⟹ x ∉ set (filter-min xs) ⟹ rel y x ⟹
thesis
 shows thesis
proof (cases x ∈ set (filter-min xs))
```

case *True*
thus *?thesis* **by** (*rule assms*(*3*))
next
case *False*
with *assms*(*1*, *2*) **obtain** *y* **where** $y \in$ *set* (*filter-min xs*) **and** *rel y x*
unfolding *filter-min-def* **by** (*rule filter-min-aux-relE*)
from *this*(*1*) *False this*(*2*) **show** *?thesis* **by** (*rule assms*(*4*))
qed

corollary *filter-min-relE*:
assumes *transp rel* **and** *reflp rel* **and** $x \in$ *set xs*
obtains *y* **where** $y \in$ *set* (*filter-min xs*) **and** *rel y x*
using *assms*(*1*, *3*)
proof (*rule filter-min-cases*)
assume $x \in$ *set* (*filter-min xs*)
moreover from *assms*(*2*) **have** *rel x x* **by** (*rule reflpD*)
ultimately show *?thesis* **..**
qed

lemma *filter-min-minimal*:
assumes *transp rel* **and** $x \in$ *set* (*filter-min xs*) **and** $y \in$ *set* (*filter-min xs*) **and** *rel x y*
shows $x = y$
using *assms* **unfolding** *filter-min-def* **by** (*rule filter-min-aux-minimal*) *simp*

lemma *filter-min-distinct*:
assumes *reflp rel*
shows *distinct* (*filter-min xs*)
unfolding *filter-min-def* **by** (*rule filter-min-aux-distinct*, *fact*, *simp*)

lemma *filter-min-append-subset*: *set* (*filter-min-append xs ys*) $\subseteq$ *set xs* $\cup$ *set ys*
by (*auto simp*: *filter-min-append-def*)

lemma *filter-min-append-cases*:
assumes *transp rel* **and** $x \in$ *set xs* $\cup$ *set ys*
assumes $x \in$ *set* (*filter-min-append xs ys*) $\Longrightarrow$ *thesis*
assumes $\bigwedge y.$ $y \in$ *set* (*filter-min-append xs ys*) $\Longrightarrow x \notin$ *set* (*filter-min-append xs ys*) $\Longrightarrow$ *rel y x* $\Longrightarrow$ *thesis*
shows *thesis*
proof (*cases* $x \in$ *set* (*filter-min-append xs ys*))
case *True*
thus *?thesis* **by** (*rule assms*(*3*))
next
case *False*
define *P* **where** $P = (\lambda zs.\ \lambda a.\ \neg\ (\exists z \in set\ zs.\ rel\ z\ a))$
from *assms*(*2*) **obtain** *y* **where** $y \in$ *set* (*filter-min-append xs ys*) **and** *rel y x*
proof
assume $x \in$ *set xs*
with *False* **obtain** *y* **where** $y \in$ *set* (*filter-min-append xs ys*) **and** *rel y x*

by (*auto simp*: *filter-min-append-def P-def*)
thus *?thesis* **..**
next
assume $x \in set\ ys$
with *False* **obtain** y **where** $y \in set\ xs$ **and** *rel y x*
by (*auto simp*: *filter-min-append-def P-def*)
show *?thesis*
proof (*cases* $y \in set$ (*filter-min-append xs ys*))
case *True*
thus *?thesis* **using** ‹*rel y x*› **..**
next
case *False*
with ‹$y \in set\ xs$› **obtain** y' **where** y': $y' \in set$ (*filter-min-append xs ys*) **and** *rel* y' y
by (*auto simp*: *filter-min-append-def P-def*)
from *assms*(*1*) *this*(*2*) ‹*rel y x*› **have** *rel* y' x **by** (*rule transpD*)
with y' **show** *?thesis* **..**
qed
qed
from *this*(*1*) *False this*(*2*) **show** *?thesis* **by** (*rule assms*(*4*))
qed

corollary *filter-min-append-relE*:
assumes *transp rel* **and** *reflp rel* **and** $x \in set\ xs \cup set\ ys$
obtains y **where** $y \in set$ (*filter-min-append xs ys*) **and** *rel y x*
using *assms*(*1*, *3*)
proof (*rule filter-min-append-cases*)
assume $x \in set$ (*filter-min-append xs ys*)
moreover from *assms*(*2*) **have** *rel x x* **by** (*rule reflpD*)
ultimately show *?thesis* **..**
qed

lemma *filter-min-append-minimal*:
assumes $\bigwedge x'\ y'.\ x' \in set\ xs \Longrightarrow y' \in set\ xs \Longrightarrow rel\ x'\ y' \Longrightarrow x' = y'$
and $\bigwedge x'\ y'.\ x' \in set\ ys \Longrightarrow y' \in set\ ys \Longrightarrow rel\ x'\ y' \Longrightarrow x' = y'$
and $x \in set$ (*filter-min-append xs ys*) **and** $y \in set$ (*filter-min-append xs ys*) **and** *rel x y*
shows $x = y$
proof −
define P **where** $P = (\lambda zs.\ \lambda a.\ \neg\ (\exists\, z{\in}set\ zs.\ rel\ z\ a))$
define *ys1* **where** *ys1* = *filter* (*P xs*) *ys*
from *assms*(*3*) **have** $x \in set\ xs \cup set\ ys1$
by (*auto simp*: *filter-min-append-def P-def ys1-def*)
moreover from *assms*(*4*) **have** $y \in set$ (*filter* (*P ys1*) *xs*) $\cup$ *set ys1*
by (*simp add*: *filter-min-append-def P-def ys1-def*)
ultimately show *?thesis*
proof (*elim UnE*)
assume $x \in set\ xs$
assume $y \in set$ (*filter* (*P ys1*) *xs*)

```
    hence y ∈ set xs by simp
    with ‹x ∈ set xs› show ?thesis using assms(5) by (rule assms(1))
  next
    assume y ∈ set ys1
    hence ⋀z. z ∈ set xs ⟹ ¬ rel z y by (simp add: ys1-def P-def)
    moreover assume x ∈ set xs
    ultimately have ¬ rel x y by blast
    thus ?thesis using ‹rel x y› ..
  next
    assume y ∈ set (filter (P ys1) xs)
    hence ⋀z. z ∈ set ys1 ⟹ ¬ rel z y by (simp add: P-def)
    moreover assume x ∈ set ys1
    ultimately have ¬ rel x y by blast
    thus ?thesis using ‹rel x y› ..
  next
    assume x ∈ set ys1 and y ∈ set ys1
    hence x ∈ set ys and y ∈ set ys by (simp-all add: ys1-def)
    thus ?thesis using assms(5) by (rule assms(2))
  qed
qed

lemma filter-min-append-distinct:
  assumes reflp rel and distinct xs and distinct ys
  shows distinct (filter-min-append xs ys)
proof −
  define P where P = (λzs. λa. ¬ (∃ z∈set zs. rel z a))
  define ys1 where ys1 = filter (P xs) ys
  from assms(2) have distinct (filter (P ys1) xs) by simp
  moreover from assms(3) have distinct ys1 by (simp add: ys1-def)
  moreover have set (filter (P ys1) xs) ∩ set ys1 = {}
  proof (simp add: set-eq-iff, intro allI impI notI)
    fix x
    assume P ys1 x
    hence ⋀z. z ∈ set ys1 ⟹ ¬ rel z x by (simp add: P-def)
    moreover assume x ∈ set ys1
    ultimately have ¬ rel x x by blast
    moreover from assms(1) have rel x x by (rule reflpD)
    ultimately show False ..
  qed
  ultimately show ?thesis by (simp add: filter-min-append-def ys1-def P-def)
qed

end

end
```

3 Properties of Binary Relations

```
theory Confluence
```

```
  imports Abstract−Rewriting.Abstract-Rewriting Open-Induction.Restricted-Predicates
begin
```

This theory formalizes some general properties of binary relations, in particular a very weak sufficient condition for a relation to be Church-Rosser.

3.1 *Restricted-Predicates.wfp-on*

```
lemma wfp-on-imp-wfP:
  assumes wfp-on r A
  shows wfP (λx y. r x y ∧ x ∈ A ∧ y ∈ A) (is wfP ?r)
proof (simp add: wfp-def wf-def, intro allI impI)
  fix P x
  assume ∀x. (∀y. r y x ∧ y ∈ A ∧ x ∈ A ⟶ P y) ⟶ P x
  hence ∗: ⋀x. (⋀y. x ∈ A ⟹ y ∈ A ⟹ r y x ⟹ P y) ⟹ P x by blast
  from assms have ∗∗: ⋀a. a ∈ A ⟹ (⋀x. x ∈ A ⟹ (⋀y. y ∈ A ⟹ r y x ⟹
P y) ⟹ P x) ⟹ P a
    by (rule wfp-on-induct) blast+
  show P x
  proof (cases x ∈ A)
    case True
    from this ∗ show ?thesis by (rule ∗∗)
  next
    case False
    show ?thesis
    proof (rule ∗)
      fix y
      assume x ∈ A
      with False show P y ..
    qed
  qed
qed

lemma wfp-onI-min:
  assumes ⋀x Q. x ∈ Q ⟹ Q ⊆ A ⟹ ∃z∈Q. ∀y∈A. r y z ⟶ y ∉ Q
  shows wfp-on r A
proof (intro inductive-on-imp-wfp-on minimal-imp-inductive-on allI impI)
  fix Q x
  assume x ∈ Q ∧ Q ⊆ A
  hence x ∈ Q and Q ⊆ A by simp-all
  hence ∃z∈Q. ∀y∈A. r y z ⟶ y ∉ Q by (rule assms)
  then obtain z where z ∈ Q and 1: ⋀y. y ∈ A ⟹ r y z ⟹ y ∉ Q by blast
  show ∃z∈Q. ∀y. r y z ⟶ y ∉ Q
  proof (intro bexI allI impI)
    fix y
    assume r y z
    show y ∉ Q
    proof (cases y ∈ A)
      case True
```

```
      thus ?thesis using ‹r y z› by (rule 1)
    next
      case False
      with ‹Q ⊆ A› show ?thesis by blast
    qed
  qed fact
qed

lemma wfp-onE-min:
  assumes wfp-on r A and x ∈ Q and Q ⊆ A
  obtains z where z ∈ Q and ⋀y. r y z ⟹ y ∉ Q
  using wfp-on-imp-minimal[OF assms(1)] assms(2, 3) by blast

lemma wfp-onI-chain: ¬ (∃f. ∀i. f i ∈ A ∧ r (f (Suc i)) (f i)) ⟹ wfp-on r A
  by (simp add: wfp-on-def)

lemma finite-minimalE:
  assumes finite A and A ≠ {} and irreflp rel and transp rel
  obtains a where a ∈ A and ⋀b. rel b a ⟹ b ∉ A
  using assms(1, 2)
proof (induct arbitrary: thesis)
  case empty
  from empty(2) show ?case by simp
next
  case (insert a A)
  show ?case
  proof (cases A = {})
    case True
    show ?thesis
    proof (rule insert(4))
      fix b
      assume rel b a
      with assms(3) show b ∉ insert a A by (auto simp: True irreflp-def)
    qed simp
  next
    case False
    with insert(3) obtain z where z ∈ A and *: ⋀b. rel b z ⟹ b ∉ A by blast
    show ?thesis
    proof (cases rel a z)
      case True
      show ?thesis
      proof (rule insert(4))
        fix b
        assume rel b a
        with assms(4) have rel b z using ‹rel a z› by (rule transpD)
        hence b ∉ A by (rule *)
        moreover from ‹rel b a› assms(3) have b ≠ a by (auto simp: irreflp-def)
        ultimately show b ∉ insert a A by simp
      qed simp
```

next
  case *False*
  show *?thesis*
  proof (*rule insert*(*4*))
    fix *b*
    assume *rel b z*
    hence $b \notin A$ **by** (*rule* ∗)
    moreover from ‹*rel b z*› *False* **have** $b \neq a$ **by** *blast*
    ultimately show $b \notin$ *insert a A* **by** *simp*
  next
    from ‹$z \in A$› **show** $z \in$ *insert a A* **by** *simp*
  qed
qed
qed
qed

lemma *wfp-on-finite*:
assumes *irreflp rel* **and** *transp rel* **and** *finite A*
shows *wfp-on rel A*
proof (*rule wfp-onI-min*)
fix *x Q*
assume $x \in Q$ **and** $Q \subseteq A$
from *this*(*2*) *assms*(*3*) **have** *finite Q* **by** (*rule finite-subset*)
moreover from ‹$x \in Q$› **have** $Q \neq \{\}$ **by** *blast*
ultimately obtain *z* **where** $z \in Q$ **and** $\bigwedge y.$ *rel y z* $\Longrightarrow y \notin Q$ **using** *assms*(*1*, *2*)
by (*rule finite-minimalE*) *blast*
thus $\exists z{\in}Q.\ \forall y{\in}A.$ *rel y z* $\longrightarrow y \notin Q$ **by** *blast*
qed

3.2 Relations

locale *relation* = **fixes** *r*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹→› *50*)
begin

abbreviation *rtc*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹$\rightarrow^*$› *50*)
where *rtc a b* ≡ r^{**} *a b*

abbreviation *sc*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹↔› *50*)
where *sc a b* ≡ $a \rightarrow b \vee b \rightarrow a$

definition *is-final*::′*a* ⇒ *bool* **where**
is-final a ≡ ¬ (∃ *b*. *r a b*)

definition *srtc*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹$\leftrightarrow^*$› *50*) **where**
srtc a b ≡ sc^{**} *a b*
definition *cs*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹$\downarrow^*$› *50*) **where**
cs a b ≡ $(\exists s.\ (a \rightarrow^* s) \wedge (b \rightarrow^* s))$

definition *is-confluent-on* :: *'a set* ⇒ *bool*
  where *is-confluent-on A* ⟷ (∀ *a*∈*A*. ∀ *b1 b2*. (*a* →* *b1* ∧ *a* →* *b2*) ⟶ *b1* ↓* *b2*)

definition *is-confluent* :: *bool*
  where *is-confluent* ≡ *is-confluent-on UNIV*

definition *is-loc-confluent* :: *bool*
  where *is-loc-confluent* ≡ (∀ *a b1 b2*. (*a* → *b1* ∧ *a* → *b2*) ⟶ *b1* ↓* *b2*)

definition *is-ChurchRosser* :: *bool*
  where *is-ChurchRosser* ≡ (∀ *a b*. *a* ↔* *b* ⟶ *a* ↓* *b*)

definition *dw-closed* :: *'a set* ⇒ *bool*
  where *dw-closed A* ⟷ (∀ *a*∈*A*. ∀ *b*. *a* → *b* ⟶ *b* ∈ *A*)

lemma *dw-closedI* [*intro*]:
  assumes ⋀*a b*. *a* ∈ *A* ⟹ *a* → *b* ⟹ *b* ∈ *A*
  shows *dw-closed A*
  unfolding *dw-closed-def* **using** *assms* **by** *auto*

lemma *dw-closedD*:
  assumes *dw-closed A* **and** *a* ∈ *A* **and** *a* → *b*
  shows *b* ∈ *A*
  using *assms* **unfolding** *dw-closed-def* **by** *auto*

lemma *dw-closed-rtrancl*:
  assumes *dw-closed A* **and** *a* ∈ *A* **and** *a* →* *b*
  shows *b* ∈ *A*
  using *assms*(*3*)
proof (*induct b*)
  case *base*
  from *assms*(*2*) **show** *?case* **.**
next
  case (*step y z*)
  from *assms*(*1*) *step*(*3*) *step*(*2*) **show** *?case* **by** (*rule dw-closedD*)
qed

lemma *dw-closed-empty*: *dw-closed* {}
  by (*rule*, *simp*)

lemma *dw-closed-UNIV*: *dw-closed UNIV*
  by (*rule*, *intro UNIV-I*)

3.3 Setup for Connection to Theory *Abstract−Rewriting.Abstract-Rewriting*

abbreviation (*input*) *relset*::(*'a* ∗ *'a*) *set* **where**
  relset ≡ {(*x*, *y*). *x* → *y*}

lemma *rtc-rtranclI*:
  assumes $a \rightarrow^* b$
  shows $(a, b) \in relset^*$
using *assms* **by** (*simp only*: *Enum.rtranclp-rtrancl-eq*)

lemma *final-NF*: (*is-final a*) = ($a \in$ *NF relset*)
unfolding *is-final-def NF-def* **by** *simp*

lemma *sc-symcl*: $(a \leftrightarrow b) = ((a, b) \in relset^{\leftrightarrow})$
by *simp*

lemma *srtc-conversion*: $(a \leftrightarrow^* b) = ((a, b) \in relset^{\leftrightarrow *})$
proof −
  have $\{(a, b). (a, b) \in \{(x, y). x \rightarrow y\}^{\leftrightarrow}\} = \{(a, b). a \rightarrow b\}^{\leftrightarrow}$ **by** *auto*
  thus *?thesis* **unfolding** *srtc-def conversion-def sc-symcl Enum.rtranclp-rtrancl-eq*
by *simp*
qed

lemma *cs-join*: $(a \downarrow^* b) = ((a, b) \in relset^{\downarrow})$
  unfolding *cs-def join-def* **by** (*auto simp add*: *Enum.rtranclp-rtrancl-eq rtrancl-converse*)

lemma *confluent-CR*: *is-confluent* = *CR relset*
  by (*auto simp add*: *is-confluent-def is-confluent-on-def CR-defs Enum.rtranclp-rtrancl-eq cs-join*)

lemma *ChurchRosser-conversion*: *is-ChurchRosser* = $(relset^{\leftrightarrow *} \subseteq relset^{\downarrow})$
  by (*auto simp add*: *is-ChurchRosser-def cs-join srtc-conversion*)

lemma *loc-confluent-WCR*:
  shows *is-loc-confluent* = *WCR relset*
unfolding *is-loc-confluent-def WCR-defs* **by** (*auto simp add*: *cs-join*)

lemma *wf-converse*:
  shows $(wfP\ r\hat{}{-}{-}1) = (wf\ (relset^{-1}))$
unfolding *wfp-def converse-def* **by** *simp*

lemma *wf-SN*:
  shows $(wfP\ r\hat{}{-}{-}1) = (SN\ relset)$
unfolding *wf-converse wf-iff-no-infinite-down-chain SN-on-def* **by** *auto*

3.4 Simple Lemmas

lemma *rtrancl-is-final*:
  assumes $a \rightarrow^* b$ **and** *is-final a*
  shows $a = b$
proof −
  from *rtranclpD*[*OF* ‹$a \rightarrow^* b$›] **show** *?thesis*
  proof
    assume $a \neq b \wedge (\rightarrow)^{++}\ a\ b$

    hence $(\rightarrow)^{++}$ *a b* **by** *simp*
    from ‹*is-final a*› *final-NF* **have** $a \in$ *NF relset* **by** *simp*
    from *NF-no-trancl-step*[*OF this*] **have** $(a, b) \notin \{(x, y).\ x \rightarrow y\}^{+}$ **..**
    thus *?thesis* **using** ‹$(\rightarrow)^{++}$ *a b*› **unfolding** *tranclp-unfold* **..**
  qed
qed

lemma *cs-refl*:
  shows $x \downarrow^{*} x$
unfolding *cs-def*
proof
  show $x \rightarrow^{*} x \wedge x \rightarrow^{*} x$ **by** *simp*
qed

lemma *cs-sym*:
  assumes $x \downarrow^{*} y$
  shows $y \downarrow^{*} x$
using *assms* **unfolding** *cs-def*
proof
  fix *z*
  assume *a*: $x \rightarrow^{*} z \wedge y \rightarrow^{*} z$
  show $\exists s.\ y \rightarrow^{*} s \wedge x \rightarrow^{*} s$
  proof
    from *a* **show** $y \rightarrow^{*} z \wedge x \rightarrow^{*} z$ **by** *simp*
  qed
qed

lemma *rtc-implies-cs*:
  assumes $x \rightarrow^{*} y$
  shows $x \downarrow^{*} y$
proof −
  from *joinI-left*[*OF rtc-rtranclI*[*OF assms*]] *cs-join* **show** *?thesis* **by** *simp*
qed

lemma *rtc-implies-srtc*:
  assumes $a \rightarrow^{*} b$
  shows $a \leftrightarrow^{*} b$
proof −
  from *conversionI'*[*OF rtc-rtranclI*[*OF assms*]] *srtc-conversion* **show** *?thesis* **by** *simp*
qed

lemma *srtc-symmetric*:
  assumes $a \leftrightarrow^{*} b$
  shows $b \leftrightarrow^{*} a$
proof −
  from *symD*[*OF conversion-sym*[*of relset*], *of a b*] *assms srtc-conversion* **show** *?thesis* **by** *simp*
qed

lemma *srtc-transitive*:
assumes $a \leftrightarrow^* b$ **and** $b \leftrightarrow^* c$
shows $a \leftrightarrow^* c$
proof −
from *rtranclp-trans*[*of* $(\leftrightarrow)$ *a b c*] *assms* **show** $a \leftrightarrow^* c$ **unfolding** *srtc-def* **.**
qed

lemma *cs-implies-srtc*:
assumes $a \downarrow^* b$
shows $a \leftrightarrow^* b$
proof −
from *assms cs-join* **have** $(a, b) \in relset^{\downarrow}$ **by** *simp*
hence $(a, b) \in relset^{\leftrightarrow *}$ **using** *join-imp-conversion* **by** *auto*
thus *?thesis* **using** *srtc-conversion* **by** *simp*
qed

lemma *confluence-equiv-ChurchRosser*: *is-confluent* = *is-ChurchRosser*
by (*simp only*: *ChurchRosser-conversion confluent-CR*, *fact CR-iff-conversion-imp-join*)

corollary *confluence-implies-ChurchRosser*:
assumes *is-confluent*
shows *is-ChurchRosser*
using *assms* **by** (*simp only*: *confluence-equiv-ChurchRosser*)

lemma *ChurchRosser-unique-final*:
assumes *is-ChurchRosser* **and** $a \rightarrow^* b1$ **and** $a \rightarrow^* b2$ **and** *is-final b1* **and** *is-final b2*
shows $b1 = b2$
proof −
from ‹*is-ChurchRosser*› *confluence-equiv-ChurchRosser confluent-CR* **have** *CR relset* **by** *simp*
from *CR-imp-UNF*[*OF this*] *assms* **show** *?thesis* **unfolding** *UNF-defs normalizability-def*
by (*auto simp add*: *Enum.rtranclp-rtrancl-eq final-NF*)
qed

lemma *wf-on-imp-nf-ex*:
assumes *wfp-on* $((\rightarrow)^{-1-1})$ *A* **and** *dw-closed A* **and** $a \in A$
obtains *b* **where** $a \rightarrow^* b$ **and** *is-final b*
proof −
let $?A = \{b \in A.\ a \rightarrow^* b\}$
note *assms*(*1*)
moreover from *assms*(*3*) **have** $a \in ?A$ **by** *simp*
moreover have $?A \subseteq A$ **by** *auto*
ultimately show *?thesis*
proof (*rule wfp-onE-min*)
fix *z*
assume $z \in ?A$ **and** $\bigwedge y.\ (\rightarrow)^{-1-1}\ y\ z \Longrightarrow y \notin ?A$

from *this*(*2*) **have** $*$: $\bigwedge y.\ z \to y \Longrightarrow y \notin\ ?A$ **by** *simp*
from ‹$z \in\ ?A$› **have** $z \in A$ **and** $a \to^* z$ **by** *simp-all*
show *thesis*
proof (*rule*, *fact*)
show *is-final* z **unfolding** *is-final-def*
proof
assume $\exists\, y.\ z \to y$
then obtain y **where** $z \to y$ **..**
hence $y \notin\ ?A$ **by** (*rule* $*$)
moreover from *assms*(*2*) ‹$z \in A$› ‹$z \to y$› **have** $y \in A$ **by** (*rule dw-closedD*)
ultimately have $\neg\ (a \to^* y)$ **by** *simp*
with *rtranclp-trans*[*OF* ‹$a \to^* z$›, *of* y] ‹$z \to y$› **show** *False* **by** *auto*
qed
qed
qed
qed

lemma *unique-nf-imp-confluence-on*:
assumes *major*: $\bigwedge a\ b1\ b2.\ a \in A \Longrightarrow (a \to^* b1) \Longrightarrow (a \to^* b2) \Longrightarrow$ *is-final* $b1$ $\Longrightarrow$ *is-final* $b2 \Longrightarrow b1 = b2$
and *wf*: *wfp-on* $((\to)^{-1-1})$ A **and** *dw*: *dw-closed* A
shows *is-confluent-on* A
unfolding *is-confluent-on-def*
proof (*intro ballI allI impI*)
fix $a\ b1\ b2$
assume $a \to^* b1 \wedge a \to^* b2$
hence $a \to^* b1$ **and** $a \to^* b2$ **by** *simp-all*
assume $a \in A$
from *dw this* ‹$a \to^* b1$› **have** $b1 \in A$ **by** (*rule dw-closed-rtrancl*)
from *wf dw this* **obtain** *c1* **where** $b1 \to^* c1$ **and** *is-final c1* **by** (*rule wf-on-imp-nf-ex*)
from *dw* ‹$a \in A$› ‹$a \to^* b2$› **have** $b2 \in A$ **by** (*rule dw-closed-rtrancl*)
from *wf dw this* **obtain** *c2* **where** $b2 \to^* c2$ **and** *is-final c2* **by** (*rule wf-on-imp-nf-ex*)
have $c1 = c2$
by (*rule major*, *fact*, *rule rtranclp-trans*[*OF* ‹$a \to^* b1$›], *fact*, *rule rtranclp-trans*[*OF* ‹$a \to^* b2$›], *fact*+)
show $b1 \downarrow^* b2$ **unfolding** *cs-def*
proof (*intro exI*, *intro conjI*)
show $b1 \to^* c1$ **by** *fact*
next
show $b2 \to^* c1$ **unfolding** ‹$c1 = c2$› **by** *fact*
qed
qed

corollary *wf-imp-nf-ex*:
assumes *wfP* $((\to)^{-1-1})$
obtains b **where** $a \to^* b$ **and** *is-final* b
proof $-$
from *assms* **have** *wfp-on* $(r\widehat{\ }{-}{-}1)$ *UNIV* **by** *simp*
moreover note *dw-closed-UNIV*

moreover have $a \in UNIV$ **..**
ultimately obtain b **where** $a \to^* b$ **and** *is-final* b **by** (*rule wf-on-imp-nf-ex*)
thus *?thesis* **..**
qed

corollary *unique-nf-imp-confluence*:
assumes $\bigwedge a\ b1\ b2.\ (a \to^* b1) \Longrightarrow (a \to^* b2) \Longrightarrow$ *is-final* $b1 \Longrightarrow$ *is-final* $b2 \Longrightarrow b1 = b2$
and *wfP* $((\to)^{-1-1})$
shows *is-confluent*
unfolding *is-confluent-def*
by (*rule unique-nf-imp-confluence-on*, *erule assms*(*1*), *assumption*+, *simp add*: *assms*(*2*), *fact dw-closed-UNIV*)

end

3.5 Advanced Results and the Generalized Newman Lemma

definition *relbelow-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*)
where *relbelow-on A ord z rel a b* ≡ ($a \in A \wedge b \in A \wedge$ *rel a b* ∧ *ord a z* ∧ *ord b z*)

definition *cbelow-on-1* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*)
where *cbelow-on-1 A ord z rel* ≡ (*relbelow-on A ord z rel*)$^{++}$

definition *cbelow-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*)
where *cbelow-on A ord z rel a b* ≡ ($a = b \wedge b \in A \wedge$ *ord b z*) ∨ *cbelow-on-1 A ord z rel a b*

Note that *cbelow-on* cannot be defined as the reflexive-transitive closure of *relbelow-on*, since it is in general not reflexive!

definition *is-loc-connective-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *is-loc-connective-on A ord r* ⟷ (∀ *a*∈*A*. ∀ *b1 b2*. *r a b1* ∧ *r a b2* ⟶ *cbelow-on A ord a* (*relation.sc r*) *b1 b2*)

Note that *Restricted-Predicates.wfp-on* is *not* the same as *SN-on*, since in the definition of *SN-on* only the *first* element of the chain must be in the set.

lemma *cbelow-on-first-below*:
assumes *cbelow-on A ord z rel a b*
shows *ord a z*
using *assms* **unfolding** *cbelow-on-def*
proof
assume *cbelow-on-1 A ord z rel a b*

```
 thus ord a z unfolding cbelow-on-1-def by (induct rule: tranclp-induct, simp
add: relbelow-on-def)
qed simp

lemma cbelow-on-second-below:
  assumes cbelow-on A ord z rel a b
  shows ord b z
  using assms unfolding cbelow-on-def
proof
  assume cbelow-on-1 A ord z rel a b
  thus ord b z unfolding cbelow-on-1-def
    by (induct rule: tranclp-induct, simp-all add: relbelow-on-def)
qed simp

lemma cbelow-on-first-in:
  assumes cbelow-on A ord z rel a b
  shows a ∈ A
  using assms unfolding cbelow-on-def
proof
  assume cbelow-on-1 A ord z rel a b
 thus ?thesis unfolding cbelow-on-1-def by (induct rule: tranclp-induct, simp
add: relbelow-on-def)
qed simp

lemma cbelow-on-second-in:
  assumes cbelow-on A ord z rel a b
  shows b ∈ A
  using assms unfolding cbelow-on-def
proof
  assume cbelow-on-1 A ord z rel a b
  thus ?thesis unfolding cbelow-on-1-def
    by (induct rule: tranclp-induct, simp-all add: relbelow-on-def)
qed simp

lemma cbelow-on-intro [intro]:
  assumes main: cbelow-on A ord z rel a b and c ∈ A and rel b c and ord c z
  shows cbelow-on A ord z rel a c
proof −
  from main have b ∈ A by (rule cbelow-on-second-in)
  from main show ?thesis unfolding cbelow-on-def
  proof (intro disjI2)
    assume cases: (a = b ∧ b ∈ A ∧ ord b z) ∨ cbelow-on-1 A ord z rel a b
    from ‹b ∈ A› ‹c ∈ A› ‹rel b c› ‹ord c z› cbelow-on-second-below[OF main]
      have bc: relbelow-on A ord z rel b c by (simp add: relbelow-on-def)
    from cases show cbelow-on-1 A ord z rel a c
    proof
      assume a = b ∧ b ∈ A ∧ ord b z
      from this bc have relbelow-on A ord z rel a c by simp
      thus ?thesis by (simp add: cbelow-on-1-def)
```

```
    next
      assume cbelow-on-1 A ord z rel a b
     from this bc show ?thesis unfolding cbelow-on-1-def by (rule tranclp.intros(2))
    qed
  qed
qed

lemma cbelow-on-induct [consumes 1, case-names base step]:
  assumes a: cbelow-on A ord z rel a b
    and base: a ∈ A ⟹ ord a z ⟹ P a
    and ind: ⋀b c. [| cbelow-on A ord z rel a b; rel b c; c ∈ A; ord c z; P b |] ==>
P c
  shows P b
  using a unfolding cbelow-on-def
proof
  assume a = b ∧ b ∈ A ∧ ord b z
  from this base show P b by simp
next
  assume cbelow-on-1 A ord z rel a b
  thus P b unfolding cbelow-on-1-def
  proof (induct x≡a b)
    fix b
    assume relbelow-on A ord z rel a b
    hence rel a b and a ∈ A and b ∈ A and ord a z and ord b z
      by (simp-all add: relbelow-on-def)
    hence cbelow-on A ord z rel a a by (simp add: cbelow-on-def)
    from this ‹rel a b› ‹b ∈ A› ‹ord b z› base[OF ‹a ∈ A› ‹ord a z›] show P b by
(rule ind)
  next
    fix b c
    assume IH: (relbelow-on A ord z rel)⁺⁺ a b and P b and relbelow-on A ord z
rel b c
    hence rel b c and b ∈ A and c ∈ A and ord b z and ord c z
      by (simp-all add: relbelow-on-def)
     from IH have cbelow-on A ord z rel a b by (simp add: cbelow-on-def cbe-
low-on-1-def)
    from this ‹rel b c› ‹c ∈ A› ‹ord c z› ‹P b› show P c by (rule ind)
  qed
qed

lemma cbelow-on-symmetric:
  assumes main: cbelow-on A ord z rel a b and symp rel
  shows cbelow-on A ord z rel b a
  using main unfolding cbelow-on-def
proof
  assume a1: a = b ∧ b ∈ A ∧ ord b z
  show b = a ∧ a ∈ A ∧ ord a z ∨ cbelow-on-1 A ord z rel b a
  proof
    from a1 show b = a ∧ a ∈ A ∧ ord a z by simp
```

```
  qed
next
  assume a2: cbelow-on-1 A ord z rel a b
  show b = a ∧ a ∈ A ∧ ord a z ∨ cbelow-on-1 A ord z rel b a
  proof (rule disjI2)
    from ‹symp rel› have symp (relbelow-on A ord z rel) unfolding symp-def
    proof (intro allI impI)
      fix x y
      assume rel-sym: ∀ x y. rel x y ⟶ rel y x
      assume relbelow-on A ord z rel x y
      hence rel x y and x ∈ A and y ∈ A and ord x z and ord y z
        by (simp-all add: relbelow-on-def)
      show relbelow-on A ord z rel y x unfolding relbelow-on-def
      proof (intro conjI)
        from rel-sym ‹rel x y› show rel y x by simp
      qed fact+
    qed
    from sym-trancl[to-pred, OF this] a2 show cbelow-on-1 A ord z rel b a
      by (simp add: symp-def cbelow-on-1-def)
  qed
qed

lemma cbelow-on-transitive:
  assumes cbelow-on A ord z rel a b and cbelow-on A ord z rel b c
  shows cbelow-on A ord z rel a c
proof (induct rule: cbelow-on-induct[OF ‹cbelow-on A ord z rel b c›])
  from ‹cbelow-on A ord z rel a b› show cbelow-on A ord z rel a b .
next
  fix c0 c
  assume cbelow-on A ord z rel b c0 and rel c0 c and c ∈ A and ord c z and
cbelow-on A ord z rel a c0
  show cbelow-on A ord z rel a c by (rule, fact+)
qed

lemma cbelow-on-mono:
  assumes cbelow-on A ord z rel a b and A ⊆ B
  shows cbelow-on B ord z rel a b
  using assms(1)
proof (induct rule: cbelow-on-induct)
  case base
  show ?case by (simp add: cbelow-on-def, intro disjI1 conjI, rule, fact+)
next
  case (step b c)
  from step(3) assms(2) have c ∈ B ..
  from step(5) this step(2) step (4) show ?case ..
qed

locale relation-order = relation +
  fixes ord::'a ⇒ 'a ⇒ bool
```

```
  fixes A::'a set
  assumes trans: ord x y ⟹ ord y z ⟹ ord x z
  assumes wf: wfp-on ord A
  assumes refines: (→) ≤ ord⁻¹⁻¹
begin

lemma relation-refines:
  assumes a → b
  shows ord b a
  using refines assms by auto

lemma relation-wf: wfp-on (→)⁻¹⁻¹ A
  using subset-refl - wf
proof (rule wfp-on-mono)
  fix x y
  assume (→)⁻¹⁻¹ x y
  hence y → x by simp
  with refines have (ord)⁻¹⁻¹ y x ..
  thus ord x y by simp
qed

lemma rtc-implies-cbelow-on:
  assumes dw-closed A and main: a →* b and a ∈ A and ord a c
  shows cbelow-on A ord c (↔) a b
  using main
proof (induct rule: rtranclp-induct)
  from assms(3) assms(4) show cbelow-on A ord c (↔) a a by (simp add: cbelow-on-def)
next
  fix b0 b
  assume a →* b0 and b0 → b and IH: cbelow-on A ord c (↔) a b0
  from assms(1) assms(3) ‹a →* b0› have b0 ∈ A by (rule dw-closed-rtrancl)
  from assms(1) this ‹b0 → b› have b ∈ A by (rule dw-closedD)
  show cbelow-on A ord c (↔) a b
  proof
    from ‹b0 → b› show b0 ↔ b by simp
  next
    from relation-refines[OF ‹b0 → b›] cbelow-on-second-below[OF IH] show ord b c by (rule trans)
  qed fact+
qed

lemma cs-implies-cbelow-on:
  assumes dw-closed A and a ↓* b and a ∈ A and b ∈ A and ord a c and ord b c
  shows cbelow-on A ord c (↔) a b
proof -
  from ‹a ↓* b› obtain s where a →* s and b →* s unfolding cs-def by auto
  have sym: symp (↔) unfolding symp-def
```

```
  proof (intro allI, intro impI)
    fix x y
    assume x ↔ y
    thus y ↔ x by auto
  qed
  from assms(1) ‹a →* s› assms(3) assms(5) have cbelow-on A ord c (↔) a s
    by (rule rtc-implies-cbelow-on)
  also have cbelow-on A ord c (↔) s b
  proof (rule cbelow-on-symmetric)
    from assms(1) ‹b →* s› assms(4) assms(6) show cbelow-on A ord c (↔) b s
      by (rule rtc-implies-cbelow-on)
  qed fact
  finally(cbelow-on-transitive) show ?thesis .
qed
```

The generalized Newman lemma, taken from [17]:

```
lemma loc-connectivity-implies-confluence:
  assumes is-loc-connective-on A ord (→) and dw-closed A
  shows is-confluent-on A
  using assms(1) unfolding is-loc-connective-on-def is-confluent-on-def
proof (intro ballI allI impI)
  fix z x y::'a
  assume ∀ a∈A. ∀ b1 b2. a → b1 ∧ a → b2 ⟶ cbelow-on A ord a (↔) b1 b2
  hence A: ⋀a b1 b2. a ∈ A ⟹ a → b1 ⟹ a → b2 ⟹ cbelow-on A ord a (↔)
b1 b2 by simp
  assume z ∈ A and z →* x ∧ z →* y
  with wf show x ↓* y
  proof (induct z arbitrary: x y rule: wfp-on-induct)
    fix z x y::'a
    assume IH: ⋀z0 x0 y0. z0 ∈ A ⟹ ord z0 z ⟹ z0 →* x0 ∧ z0 →* y0 ⟹
x0 ↓* y0
      and z →* x ∧ z →* y
    hence z →* x and z →* y by auto
    assume z ∈ A
    from converse-rtranclpE[OF ‹z →* x›] obtain x1 where x = z ∨ (z → x1 ∧
x1 →* x) by auto
    thus x ↓* y
    proof
      assume x = z
      show ?thesis unfolding cs-def
      proof
        from ‹x = z› ‹z →* y› show x →* y ∧ y →* y by simp
      qed
    next
      assume z → x1 ∧ x1 →* x
      hence z → x1 and x1 →* x by auto
      from assms(2) ‹z ∈ A› this(1) have x1 ∈ A by (rule dw-closedD)
      from converse-rtranclpE[OF ‹z →* y›] obtain y1 where y = z ∨ (z → y1
∧ y1 →* y) by auto
```

```
    thus ?thesis
    proof
      assume y = z
      show ?thesis unfolding cs-def
      proof
        from ‹y = z› ‹z →* x› show x →* x ∧ y →* x by simp
      qed
    next
      assume z → y1 ∧ y1 →* y
      hence z → y1 and y1 →* y by auto
      from assms(2) ‹z ∈ A› this(1) have y1 ∈ A by (rule dw-closedD)
      have x1 ↓* y1
        proof (induct rule: cbelow-on-induct[OF A[OF ‹z ∈ A› ‹z → x1› ‹z → y1›]])
        from cs-refl[of x1] show x1 ↓* x1 .
      next
        fix b c
        assume cbelow-on A ord z (↔) x1 b and b ↔ c and c ∈ A and ord c z and x1 ↓* b
        from this(1) have b ∈ A by (rule cbelow-on-second-in)
        from ‹x1 ↓* b› obtain w1 where x1 →* w1 and b →* w1 unfolding cs-def by auto
        from ‹b ↔ c› show x1 ↓* c
        proof
          assume b → c
          hence b →* c by simp
              from ‹cbelow-on A ord z (↔) x1 b› have ord b z by (rule cbelow-on-second-below)
          from IH[OF ‹b ∈ A› this] ‹b →* c› ‹b →* w1› have c ↓* w1 by simp
          then obtain w2 where c →* w2 and w1 →* w2 unfolding cs-def by auto
          show ?thesis unfolding cs-def
          proof
            from rtranclp-trans[OF ‹x1 →* w1› ‹w1 →* w2›] ‹c →* w2›
              show x1 →* w2 ∧ c →* w2 by simp
          qed
        next
          assume c → b
          hence c →* b by simp
          show ?thesis unfolding cs-def
          proof
            from rtranclp-trans[OF ‹c →* b› ‹b →* w1›] ‹x1 →* w1›
              show x1 →* w1 ∧ c →* w1 by simp
          qed
        qed
      qed
      then obtain w1 where x1 →* w1 and y1 →* w1 unfolding cs-def by auto
      from IH[OF ‹x1 ∈ A› relation-refines[OF ‹z → x1›]] ‹x1 →* x› ‹x1 →*
```

w1›
    have *x* ↓* *w1* **by** *simp*
   then obtain *v* **where** *x* →* *v* **and** *w1* →* *v* **unfolding** *cs-def* **by** *auto*
   from *IH*[*OF* ‹*y1* ∈ *A*› *relation-refines*[*OF* ‹*z* → *y1*›]]
     rtranclp-trans[*OF* ‹*y1* →* *w1*› ‹*w1* →* *v*›] ‹*y1* →* *y*›
    have *v* ↓* *y* **by** *simp*
   then obtain *w* **where** *v* →* *w* **and** *y* →* *w* **unfolding** *cs-def* **by** *auto*
   show *?thesis* **unfolding** *cs-def*
   proof
    from *rtranclp-trans*[*OF* ‹*x* →* *v*› ‹*v* →* *w*›] ‹*y* →* *w*› **show** *x* →* *w* ∧ *y*
→* *w* **by** *simp*
   qed
  qed
 qed
 qed
qed

end

theorem *loc-connectivity-equiv-ChurchRosser*:
 assumes *relation-order r ord UNIV*
 shows *relation.is-ChurchRosser r* = *is-loc-connective-on UNIV ord r*
proof
 assume *relation.is-ChurchRosser r*
 show *is-loc-connective-on UNIV ord r* **unfolding** *is-loc-connective-on-def*
 proof (*intro ballI allI impI*)
  fix *a b1 b2*
  assume *r a b1* ∧ *r a b2*
  hence *r a b1* **and** *r a b2* **by** *simp-all*
  hence *r*** *a b1* **and** *r*** *a b2* **by** *simp-all*
  from *relation.rtc-implies-srtc*[*OF* ‹*r*** *a b1*›] **have** *relation.srtc r b1 a* **by** (*rule relation.srtc-symmetric*)
  from *relation.srtc-transitive*[*OF this relation.rtc-implies-srtc*[*OF* ‹*r*** *a b2*›]]
have *relation.srtc r b1 b2* **.**
  with ‹*relation.is-ChurchRosser r*› **have** *relation.cs r b1 b2* **by** (*simp add*: *relation.is-ChurchRosser-def*)
  from *relation-order.cs-implies-cbelow-on*[*OF assms relation.dw-closed-UNIV this*]
   relation-order.relation-refines[*OF assms*, *of a*] ‹*r a b1*› ‹*r a b2*›
  show *cbelow-on UNIV ord a* (*relation.sc r*) *b1 b2* **by** *simp*
 qed
next
 assume *is-loc-connective-on UNIV ord r*
 from *assms this relation.dw-closed-UNIV* **have** *relation.is-confluent-on r UNIV*
  by (*rule relation-order.loc-connectivity-implies-confluence*)
 hence *relation.is-confluent r* **by** (*simp only*: *relation.is-confluent-def*)
 thus *relation.is-ChurchRosser r* **by** (*simp add*: *relation.confluence-equiv-ChurchRosser*)
qed

end

4 Polynomial Reduction

theory *Reduction*
imports *Polynomials.MPoly-Type-Class-Ordered Confluence*
begin

This theory formalizes the concept of *reduction* of polynomials by polynomials.

context *ordered-term*
begin

definition *red-single* :: $('t \Rightarrow_0 'b::field) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow bool$
  where *red-single p q f t* $\longleftrightarrow$ $(f \neq 0 \wedge lookup\ p\ (t \oplus lt\ f) \neq 0 \wedge$
  $q = p - monom\text{-}mult\ ((lookup\ p\ (t \oplus lt\ f))\ /\ lc\ f)\ t\ f)$

definition *red* :: $('t \Rightarrow_0 'b::field)\ set \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
  where *red F p q* $\longleftrightarrow$ $(\exists f \in F.\ \exists t.\ red\text{-}single\ p\ q\ f\ t)$

definition *is-red* :: $('t \Rightarrow_0 'b::field)\ set \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
  where *is-red F a* $\longleftrightarrow$ $\neg$ *relation.is-final* (*red F*) *a*

4.1 Basic Properties of Reduction

lemma *red-setI*:
  assumes $f \in F$ **and** *a*: *red-single p q f t*
  shows *red F p q*
  unfolding *red-def*
proof
  from ‹$f \in F$› **show** $f \in F$ **.**
next
  from *a* **show** $\exists t.$ *red-single p q f t* **..**
qed

lemma *red-setE*:
  assumes *red F p q*
  obtains *f* **and** *t* **where** $f \in F$ **and** *red-single p q f t*
proof −
  from *assms* **obtain** *f* **where** $f \in F$ **and** *t*: $\exists t.$ *red-single p q f t* **unfolding** *red-def* **by** *auto*
  from *t* **obtain** *t* **where** *red-single p q f t* **..**
  from ‹$f \in F$› *this* **show** *?thesis* **..**
qed

lemma *red-empty*: $\neg$ *red* {} *p q*
  by (*rule*, *elim red-setE*, *simp*)

lemma *red-singleton-zero*: $\neg$ *red* {*0*} *p q*

by (*rule*, *elim red-setE*, *simp add*: *red-single-def*)

lemma *red-union*: *red* (*F* ∪ *G*) *p q* = (*red F p q* ∨ *red G p q*)
proof
  assume *red* (*F* ∪ *G*) *p q*
  from *red-setE*[*OF this*] **obtain** *f t* **where** *f* ∈ *F* ∪ *G* **and** *r*: *red-single p q f t* **.**
  from ‹*f* ∈ *F* ∪ *G*› **have** *f* ∈ *F* ∨ *f* ∈ *G* **by** *simp*
  thus *red F p q* ∨ *red G p q*
  proof
    assume *f* ∈ *F*
    show *?thesis* **by** (*intro disjI1*, *rule red-setI*[*OF* ‹*f* ∈ *F*› *r*])
  next
    assume *f* ∈ *G*
    show *?thesis* **by** (*intro disjI2*, *rule red-setI*[*OF* ‹*f* ∈ *G*› *r*])
  qed
next
  assume *red F p q* ∨ *red G p q*
  thus *red* (*F* ∪ *G*) *p q*
  proof
    assume *red F p q*
    from *red-setE*[*OF this*] **obtain** *f t* **where** *f* ∈ *F* **and** *red-single p q f t* **.**
    show *?thesis* **by** (*intro red-setI*[*of f - - - t*], *rule UnI1*, *rule* ‹*f* ∈ *F*›, *fact*)
  next
    assume *red G p q*
    from *red-setE*[*OF this*] **obtain** *f t* **where** *f* ∈ *G* **and** *red-single p q f t* **.**
    show *?thesis* **by** (*intro red-setI*[*of f - - - t*], *rule UnI2*, *rule* ‹*f* ∈ *G*›, *fact*)
  qed
qed

lemma *red-unionI1*:
  assumes *red F p q*
  shows *red* (*F* ∪ *G*) *p q*
  unfolding *red-union* **by** (*rule disjI1*, *fact*)

lemma *red-unionI2*:
  assumes *red G p q*
  shows *red* (*F* ∪ *G*) *p q*
  unfolding *red-union* **by** (*rule disjI2*, *fact*)

lemma *red-subset*:
  assumes *red G p q* **and** *G* ⊆ *F*
  shows *red F p q*
proof −
  from ‹*G* ⊆ *F*› **obtain** *H* **where** *F* = *G* ∪ *H* **by** *auto*
  show *?thesis* **unfolding** ‹*F* = *G* ∪ *H*› **by** (*rule red-unionI1*, *fact*)
qed

lemma *red-union-singleton-zero*: *red* (*F* ∪ {*0*}) = *red F*
  by (*intro ext*, *simp only*: *red-union red-singleton-zero*, *simp*)

```
lemma red-minus-singleton-zero: red (F − {0}) = red F
  by (metis Un-Diff-cancel2 red-union-singleton-zero)

lemma red-rtrancl-subset:
  assumes major: (red G)** p q and G ⊆ F
  shows (red F)** p q
  using major
proof (induct rule: rtranclp-induct)
  show (red F)** p p ..
next
  fix r q
  assume red G r q and (red F)** p r
  show (red F)** p q
  proof
    show (red F)** p r by fact
  next
    from red-subset[OF ‹red G r q› ‹G ⊆ F›] show red F r q .
  qed
qed

lemma red-singleton: red {f} p q ⟷ (∃ t. red-single p q f t)
  unfolding red-def
proof
  assume ∃f∈{f}. ∃ t. red-single p q f t
  from this obtain f0 where f0 ∈ {f} and a: ∃ t. red-single p q f0 t ..
  from ‹f0 ∈ {f}› have f0 = f by simp
  from this a show ∃ t. red-single p q f t by simp
next
  assume a: ∃ t. red-single p q f t
  show ∃f∈{f}. ∃ t. red-single p q f t
  proof (rule, simp)
    from a show ∃ t. red-single p q f t .
  qed
qed

lemma red-single-lookup:
  assumes red-single p q f t
  shows lookup q (t ⊕ lt f) = 0
  using assms unfolding red-single-def
proof
  assume f ≠ 0 and lookup p (t ⊕ lt f) ≠ 0 ∧ q = p − monom-mult (lookup p
(t ⊕ lt f) / lc f) t f
  hence lookup p (t ⊕ lt f) ≠ 0 and q-def: q = p − monom-mult (lookup p (t ⊕
lt f) / lc f) t f
    by auto
  from lookup-minus[of p monom-mult (lookup p (t ⊕ lt f) / lc f) t f t ⊕ lt f]
       lookup-monom-mult-plus[of lookup p (t ⊕ lt f) / lc f t f lt f]
       lc-not-0[OF ‹f ≠ 0›]
```

show *?thesis* **unfolding** *q-def lc-def* **by** *simp*
qed

lemma *red-single-higher*:
assumes *red-single p q f t*
shows *higher q* (*t* ⊕ *lt f*) = *higher p* (*t* ⊕ *lt f*)
using *assms* **unfolding** *higher-eq-iff red-single-def*
proof (*intro allI*, *intro impI*)
fix *u*
assume *a*: *t* ⊕ *lt f* $\prec_t$ *u*
and *f* ≠ *0* ∧ *lookup p* (*t* ⊕ *lt f*) ≠ *0* ∧ *q* = *p* − *monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*
hence *f* ≠ *0*
and *lookup p* (*t* ⊕ *lt f*) ≠ *0*
and *q-def*: *q* = *p* − *monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*
by *simp-all*
from ‹*lookup p* (*t* ⊕ *lt f*) ≠ *0*› *lc-not-0*[*OF* ‹*f* ≠ *0*›] **have** *c-not-0*: *lookup p* (*t* ⊕ *lt f*) / *lc f* ≠ *0*
by (*simp add*: *field-simps*)
from *q-def lookup-minus*[*of p monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*]
have *q-lookup*: ⋀*s*. *lookup q s* = *lookup p s* − *lookup* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*) *s*
by *simp*
from *a lt-monom-mult*[*OF c-not-0* ‹*f* ≠ *0*›, *of t*]
have ¬ *u* $\preceq_t$ *lt* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*) **by** *simp*
with *lt-max*[*of monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f u*]
have *lookup* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*) *u* = *0* **by** *auto*
thus *lookup q u* = *lookup p u* **using** *q-lookup*[*of u*] **by** *simp*
qed

lemma *red-single-ord*:
assumes *red-single p q f t*
shows *q* $\prec_p$ *p*
unfolding *ord-strict-higher*
proof (*intro exI*, *intro conjI*)
from *red-single-lookup*[*OF assms*] **show** *lookup q* (*t* ⊕ *lt f*) = *0* **.**
next
from *assms* **show** *lookup p* (*t* ⊕ *lt f*) ≠ *0* **unfolding** *red-single-def* **by** *simp*
next
from *red-single-higher*[*OF assms*] **show** *higher q* (*t* ⊕ *lt f*) = *higher p* (*t* ⊕ *lt f*) **.**
qed

lemma *red-single-nonzero1*:
assumes *red-single p q f t*
shows *p* ≠ *0*
proof
assume *p* = *0*
from *this red-single-ord*[*OF assms*] *ord-p-zero-min*[*of q*] **show** *False* **by** *simp*

qed

lemma *red-single-nonzero2*:
  assumes *red-single p q f t*
  shows $f \neq 0$
proof
  assume $f = 0$
  from *assms monom-mult-zero-right* **have** $f \neq 0$ **by** (*simp add*: *red-single-def*)
  from *this* ‹$f = 0$› **show** *False* **by** *simp*
qed

lemma *red-single-self*:
  assumes $p \neq 0$
  shows *red-single p 0 p 0*
proof −
  from *lc-not-0*[*OF assms*] **have** *lc*: $lc\ p \neq 0$ **.**
  show *?thesis* **unfolding** *red-single-def*
  proof (*intro conjI*)
    show $p \neq 0$ **by** *fact*
  next
    from *lc* **show** *lookup p* $(0 \oplus lt\ p) \neq 0$ **unfolding** *lc-def* **by** (*simp add*: *term-simps*)
  next
    from *lc* **have** (*lookup p* $(0 \oplus lt\ p)$) / *lc p* = *1* **unfolding** *lc-def* **by** (*simp add*: *term-simps*)
    from *this monom-mult-one-left*[*of p*] **show** *0* = *p* − *monom-mult* (*lookup p* $(0 \oplus lt\ p)$ / *lc p*) *0 p*
      by *simp*
  qed
qed

lemma *red-single-trans*:
  assumes *red-single p p0 f t* **and** *lt g* $adds_t$ *lt f* **and** $g \neq 0$
  obtains *p1* **where** *red-single p p1 g* (*t* + (*lp f* − *lp g*))
proof −
  let *?s* = *t* + (*lp f* − *lp g*)
  let *?p* = *p* − *monom-mult* (*lookup p* $(?s \oplus lt\ g)$ / *lc g*) *?s g*
  have *red-single p ?p g ?s* **unfolding** *red-single-def*
  proof (*intro conjI*)
    from *assms*(*2*) **have** *eq*: $?s \oplus lt\ g = t \oplus lt\ f$ **using** *adds-term-alt splus-assoc*
      by (*auto simp*: *term-simps*)
    from ‹*red-single p p0 f t*› **have** *lookup p* $(t \oplus lt\ f) \neq 0$ **unfolding** *red-single-def* **by** *simp*
    thus *lookup p* $(?s \oplus lt\ g) \neq 0$ **by** (*simp add*: *eq*)
  qed (*fact*, *fact refl*)
  thus *?thesis* **..**
qed

lemma *red-nonzero*:

assumes *red F p q*
shows *p* ≠ *0*
proof –
from *red-setE*[*OF assms*] **obtain** *f t* **where** *red-single p q f t* **.**
show *?thesis* **by** (*rule red-single-nonzero1*, *fact*)
qed

lemma *red-self*:
assumes *p* ≠ *0*
shows *red* {*p*} *p 0*
unfolding *red-singleton*
proof
from *red-single-self*[*OF assms*] **show** *red-single p 0 p 0* **.**
qed

lemma *red-ord*:
assumes *red F p q*
shows $q \prec_p p$
proof –
from *red-setE*[*OF assms*] **obtain** *f* **and** *t* **where** *red-single p q f t* **.**
from *red-single-ord*[*OF this*] **show** $q \prec_p p$ **.**
qed

lemma *red-indI1*:
assumes *f* ∈ *F* **and** *f* ≠ *0* **and** *p* ≠ *0* **and** *adds*: *lt f* $adds_t$ *lt p*
shows *red F p* (*p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) *f*)
proof (*intro red-setI*[*OF* ‹*f* ∈ *F*›])
let *?s* = *lp p* − *lp f*
have *c*: *lookup p* (*?s* ⊕ *lt f*) = *lc p* **unfolding** *lc-def*
by (*metis add-diff-cancel-right′ adds adds-termE pp-of-term-splus*)
show *red-single p* (*p* − *monom-mult* (*lc p* / *lc f*) *?s f*) *f ?s* **unfolding** *red-single-def*
proof (*intro conjI*, *fact*)
from *c lc-not-0*[*OF* ‹*p* ≠ *0*›] **show** *lookup p* (*?s* ⊕ *lt f*) ≠ *0* **by** *simp*
next
from *c* **show** *p* − *monom-mult* (*lc p* / *lc f*) *?s f* = *p* − *monom-mult* (*lookup p* (*?s* ⊕ *lt f*) / *lc f*) *?s f*
by *simp*
qed
qed

lemma *red-indI2*:
assumes *p* ≠ *0* **and** *r*: *red F* (*tail p*) *q*
shows *red F p* (*q* + *monomial* (*lc p*) (*lt p*))
proof –
from *red-setE*[*OF r*] **obtain** *f t* **where** *f* ∈ *F* **and** *rs*: *red-single* (*tail p*) *q f t* **by** *auto*
from *rs* **have** *f* ≠ *0* **and** *ct*: *lookup* (*tail p*) (*t* ⊕ *lt f*) ≠ *0*
and *q*: *q* = *tail p* − *monom-mult* (*lookup* (*tail p*) (*t* ⊕ *lt f*) / *lc f*) *t f*
unfolding *red-single-def* **by** *simp-all*

from *ct lookup-tail*[*of p t* ⊕ *lt f*] **have** *t* ⊕ *lt f* $\prec_t$ *lt p* **by** (*auto split*: *if-splits*)
hence *c*: *lookup* (*tail p*) (*t* ⊕ *lt f*) = *lookup p* (*t* ⊕ *lt f*) **using** *lookup-tail*[*of p*] **by** *simp*
show *?thesis*
proof (*intro red-setI*[*OF* ‹*f* ∈ *F*›])
show *red-single p* (*q* + *Poly-Mapping.single* (*lt p*) (*lc p*)) *f t* **unfolding** *red-single-def*
proof (*intro conjI*, *fact*)
from *ct c* **show** *lookup p* (*t* ⊕ *lt f*) ≠ *0* **by** *simp*
next
from *q* **have** *q* + *monomial* (*lc p*) (*lt p*) =
(*monomial* (*lc p*) (*lt p*) + *tail p*) − *monom-mult* (*lookup* (*tail p*) (*t* ⊕ *lt f*) / *lc f*) *t f*
by *simp*
also have . . . = *p* − *monom-mult* (*lookup* (*tail p*) (*t* ⊕ *lt f*) / *lc f*) *t f*
using *leading-monomial-tail*[*of p*] **by** *auto*
finally show *q* + *monomial* (*lc p*) (*lt p*) = *p* − *monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*
by (*simp only*: *c*)
qed
qed
qed

lemma *red-indE*:
assumes *red F p q*
shows (∃*f*∈*F*. *f* ≠ *0* ∧ *lt f* $adds_t$ *lt p* ∧
(*q* = *p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) *f*)) ∨
red F (*tail p*) (*q* − *monomial* (*lc p*) (*lt p*))
proof −
from *red-nonzero*[*OF assms*] **have** *p* ≠ *0* **.**
from *red-setE*[*OF assms*] **obtain** *f t* **where** *f* ∈ *F* **and** *rs*: *red-single p q f t* **by** *auto*
from *rs* **have** *f* ≠ *0*
and *cn0*: *lookup p* (*t* ⊕ *lt f*) ≠ *0*
and *q*: *q* = *p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
unfolding *red-single-def* **by** *simp-all*
show *?thesis*
proof (*cases lt p* = *t* ⊕ *lt f*)
case *True*
hence *lt f* $adds_t$ *lt p* **by** (*simp add*: *term-simps*)
from *True* **have** *eq1*: *lp p* − *lp f* = *t* **by** (*simp add*: *term-simps*)
from *True* **have** *eq2*: *lc p* = *lookup p* (*t* ⊕ *lt f*) **unfolding** *lc-def* **by** *simp*
show *?thesis*
proof (*intro disjI1*, *rule bexI*[*of* - *f*], *intro conjI*, *fact*+)
from *q eq1 eq2* **show** *q* = *p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) *f*
by *simp*
qed (*fact*)
next
case *False*

from *this lookup-tail-2*[*of p t* ⊕ *lt f*]
have *ct*: *lookup* (*tail p*) (*t* ⊕ *lt f*) = *lookup p* (*t* ⊕ *lt f*) **by** *simp*
show *?thesis*
proof (*intro disjI2*, *intro red-setI*[*of f*], *fact*)
show *red-single* (*tail p*) (*q* − *monomial* (*lc p*) (*lt p*)) *f t* **unfolding** *red-single-def*
proof (*intro conjI*, *fact*)
from *cn0 ct* **show** *lookup* (*tail p*) (*t* ⊕ *lt f*) ≠ *0* **by** *simp*
next
from *leading-monomial-tail*[*of p*]
have *p* − *monomial* (*lc p*) (*lt p*) = (*monomial* (*lc p*) (*lt p*) + *tail p*) − *monomial* (*lc p*) (*lt p*)
by *simp*
also have ... = *tail p* **by** *simp*
finally have *eq*: *p* − *monomial* (*lc p*) (*lt p*) = *tail p* **.**
from *q* **have** *q* − *monomial* (*lc p*) (*lt p*) =
(*p* − *monomial* (*lc p*) (*lt p*)) − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f* **by** *simp*
also from *eq* **have** ... = *tail p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f* **by** *simp*
finally show *q* − *monomial* (*lc p*) (*lt p*) = *tail p* − *monom-mult* (*lookup* (*tail p*) (*t* ⊕ *lt f*) / *lc f*) *t f*
using *ct* **by** *simp*
qed
qed
qed
qed

lemma *is-redI*:
assumes *red F a b*
shows *is-red F a*
unfolding *is-red-def relation.is-final-def* **by** (*simp*, *intro exI*[*of - b*], *fact*)

lemma *is-redE*:
assumes *is-red F a*
obtains *b* **where** *red F a b*
using *assms* **unfolding** *is-red-def relation.is-final-def*
proof *simp*
assume *r*: ⋀*b*. *red F a b* ⟹ *thesis* **and** *b*: ∃ *x*. *red F a x*
from *b* **obtain** *b* **where** *red F a b* **..**
show *thesis* **by** (*rule r*[*of b*], *fact*)
qed

lemma *is-red-alt*:
shows *is-red F a* ⟷ (∃ *b*. *red F a b*)
proof
assume *is-red F a*
from *is-redE*[*OF this*] **obtain** *b* **where** *red F a b* **.**
show ∃ *b*. *red F a b* **by** (*intro exI*[*of - b*], *fact*)
next

assume ∃ *b*. *red F a b*
from *this* **obtain** *b* **where** *red F a b* **..**
show *is-red F a* **by** (*rule is-redI*, *fact*)
qed

lemma *is-red-singletonI*:
assumes *is-red F q*
obtains *p* **where** *p* ∈ *F* **and** *is-red* {*p*} *q*
proof −
from *assms* **obtain** *q0* **where** *red F q q0* **unfolding** *is-red-alt* **..**
from *this red-def*[*of F q q0*] **obtain** *p* **where** *p* ∈ *F* **and** *t*: ∃ *t*. *red-single q q0 p t* **by** *auto*
have *is-red* {*p*} *q* **unfolding** *is-red-alt*
proof
from *red-singleton*[*of p q q0*] *t* **show** *red* {*p*} *q q0* **by** *simp*
qed
from ‹*p* ∈ *F*› *this* **show** *?thesis* **..**
qed

lemma *is-red-singletonD*:
assumes *is-red* {*p*} *q* **and** *p* ∈ *F*
shows *is-red F q*
proof −
from *assms*(*1*) **obtain** *q0* **where** *red* {*p*} *q q0* **unfolding** *is-red-alt* **..**
from *red-singleton*[*of p q q0*] *this* **have** ∃ *t*. *red-single q q0 p t* **..**
from *this* **obtain** *t* **where** *red-single q q0 p t* **..**
show *?thesis* **unfolding** *is-red-alt*
by (*intro exI*[*of - q0*], *intro red-setI*[*OF assms*(*2*), *of q q0 t*], *fact*)
qed

lemma *is-red-singleton-trans*:
assumes *is-red* {*f*} *p* **and** *lt g* $adds_t$ *lt f* **and** *g* ≠ *0*
shows *is-red* {*g*} *p*
proof −
from ‹*is-red* {*f*} *p*› **obtain** *q* **where** *red* {*f*} *p q* **unfolding** *is-red-alt* **..**
from *this red-singleton*[*of f p q*] **obtain** *t* **where** *red-single p q f t* **by** *auto*
from *red-single-trans*[*OF this assms*(*2*, *3*)] **obtain** *q0* **where**
red-single p q0 g (*t* + (*lp f* − *lp g*)) **.**
show *?thesis*
proof (*rule is-redI*[*of* {*g*} *p q0*])
show *red* {*g*} *p q0* **unfolding** *red-def*
by (*intro bexI*[*of - g*], *intro exI*[*of - t* + (*lp f* − *lp g*)], *fact*, *simp*)
qed
qed

lemma *is-red-singleton-not-0*:
assumes *is-red* {*f*} *p*
shows *f* ≠ *0*
using *assms* **unfolding** *is-red-alt*

proof
  fix *q*
  assume *red* {*f*} *p q*
  from *this red-singleton*[*of f p q*] **obtain** *t* **where** *red-single p q f t* **by** *auto*
  thus *?thesis* **unfolding** *red-single-def* **..**
qed

lemma *irred-0*:
  shows ¬ *is-red F 0*
proof (*rule*, *rule is-redE*)
  fix *b*
  assume *red F 0 b*
  from *ord-p-zero-min*[*of b*] *red-ord*[*OF this*] **show** *False* **by** *simp*
qed

lemma *is-red-indI1*:
  assumes *f* ∈ *F* **and** *f* ≠ *0* **and** *p* ≠ *0* **and** *lt f* $adds_t$ *lt p*
  shows *is-red F p*
by (*intro is-redI*, *rule red-indI1*[*OF assms*])

lemma *is-red-indI2*:
  assumes *p* ≠ *0* **and** *is-red F* (*tail p*)
  shows *is-red F p*
proof −
  from *is-redE*[*OF* ‹*is-red F* (*tail p*)›] **obtain** *q* **where** *red F* (*tail p*) *q* **.**
  show *?thesis* **by** (*intro is-redI*, *rule red-indI2*[*OF* ‹*p* ≠ *0*›], *fact*)
qed

lemma *is-red-indE*:
  assumes *is-red F p*
  shows (∃*f*∈*F*. *f* ≠ *0* ∧ *lt f* $adds_t$ *lt p*) ∨ *is-red F* (*tail p*)
proof −
  from *is-redE*[*OF assms*] **obtain** *q* **where** *red F p q* **.**
  from *red-indE*[*OF this*] **show** *?thesis*
  proof
    assume ∃*f*∈*F*. *f* ≠ *0* ∧ *lt f* $adds_t$ *lt p* ∧ *q* = *p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) *f*
    from *this* **obtain** *f* **where** *f* ∈ *F* **and** *f* ≠ *0* **and** *lt f* $adds_t$ *lt p* **by** *auto*
    show *?thesis* **by** (*intro disjI1*, *rule bexI*[*of - f*], *intro conjI*, *fact*+)
  next
    assume *red F* (*tail p*) (*q* − *monomial* (*lc p*) (*lt p*))
    show *?thesis* **by** (*intro disjI2*, *intro is-redI*, *fact*)
  qed
qed

lemma *rtrancl-0*:
  assumes (*red F*)** *0 x*
  shows *x* = *0*
proof −

from *irred-0*[*of F*] **have** *relation.is-final* (*red F*) *0* **unfolding** *is-red-def* **by** *simp*
from *relation.rtrancl-is-final*[*OF* ‹(*red F*)** *0 x*› *this*] **show** *?thesis* **by** *simp*
qed

lemma *red-rtrancl-ord*:
assumes (*red F*)** *p q*
shows $q \preceq_p p$
using *assms*
proof *induct*
case *base*
show *?case* **..**
next
case (*step y z*)
from *step*(*2*) **have** $z \prec_p y$ **by** (*rule red-ord*)
hence $z \preceq_p y$ **by** *simp*
also note *step*(*3*)
finally show *?case* **.**
qed

lemma *components-red-subset*:
assumes *red F p q*
shows *component-of-term* ‘ *keys q* ⊆ *component-of-term* ‘ *keys p* ∪ *component-of-term* ‘ *Keys F*
proof −
from *assms* **obtain** *f t* **where** *f* ∈ *F* **and** *red-single p q f t* **by** (*rule red-setE*)
from *this*(*2*) **have** *q*: *q* = *p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
by (*simp add*: *red-single-def*)
have *component-of-term* ‘ *keys q* ⊆
component-of-term ‘ (*keys p* ∪ *keys* (*monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*))
by (*rule image-mono*, *simp add*: *q keys-minus*)
also have ... ⊆ *component-of-term* ‘ *keys p* ∪ *component-of-term* ‘ *Keys F*
proof (*simp add*: *image-Un*, *rule*)
fix *k*
assume *k* ∈ *component-of-term* ‘ *keys* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*)
then obtain *v* **where** *v* ∈ *keys* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*)
and *k* = *component-of-term v* **..**
from *this*(*1*) *keys-monom-mult-subset* **have** *v* ∈ (⊕) *t* ‘ *keys f* **..**
then obtain *u* **where** *u* ∈ *keys f* **and** *v* = *t* ⊕ *u* **..**
have *k* = *component-of-term u* **by** (*simp add*: ‹*k* = *component-of-term v*› ‹*v* = *t* ⊕ *u*› *term-simps*)
with ‹*u* ∈ *keys f*› **have** *k* ∈ *component-of-term* ‘ *keys f* **by** *fastforce*
also have ... ⊆ *component-of-term* ‘ *Keys F* **by** (*rule image-mono*, *rule keys-subset-Keys*, *fact*)
finally show *k* ∈ *component-of-term* ‘ *keys p* ∪ *component-of-term* ‘ *Keys F* **by** *simp*
qed
finally show *?thesis* **.**

qed

corollary *components-red-rtrancl-subset*:
assumes (*red F*)** *p q*
shows *component-of-term* ‘ *keys q* ⊆ *component-of-term* ‘ *keys p* ∪ *component-of-term* ‘ *Keys F*
using *assms*
proof (*induct*)
case *base*
show *?case* **by** *simp*
next
case (*step q r*)
from *step*(*2*) **have** *component-of-term* ‘ *keys r* ⊆ *component-of-term* ‘ *keys q* ∪ *component-of-term* ‘ *Keys F*
by (*rule components-red-subset*)
also from *step*(*3*) **have** ... ⊆ *component-of-term* ‘ *keys p* ∪ *component-of-term* ‘ *Keys F* **by** *blast*
finally show *?case* **.**
qed

4.2 Reducibility and Addition & Multiplication

lemma *red-single-monom-mult*:
assumes *red-single p q f t* **and** *c* ≠ *0*
shows *red-single* (*monom-mult c s p*) (*monom-mult c s q*) *f* (*s* + *t*)
proof −
from *assms*(*1*) **have** *f* ≠ *0*
and *lookup p* (*t* ⊕ *lt f*) ≠ *0*
and *q-def*: *q* = *p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
unfolding *red-single-def* **by** *auto*
have *assoc*: (*s* + *t*) ⊕ *lt f* = *s* ⊕ (*t* ⊕ *lt f*) **by** (*simp add*: *ac-simps*)
have *g2*: *lookup* (*monom-mult c s p*) ((*s* + *t*) ⊕ *lt f*) ≠ *0*
proof
assume *lookup* (*monom-mult c s p*) ((*s* + *t*) ⊕ *lt f*) = *0*
hence *c* ∗ *lookup p* (*t* ⊕ *lt f*) = *0* **using** *assoc* **by** (*simp add*: *lookup-monom-mult-plus*)
thus *False* **using** ‹*c* ≠ *0*› ‹*lookup p* (*t* ⊕ *lt f*) ≠ *0*› **by** *simp*
qed
have *g3*: *monom-mult c s q* =
(*monom-mult c s p*) − *monom-mult* ((*lookup* (*monom-mult c s p*) ((*s* + *t*) ⊕ *lt f*)) / *lc f*) (*s* + *t*) *f*
proof −
from *q-def monom-mult-dist-right-minus*[*of c s p*]
have *monom-mult c s q* =
monom-mult c s p − *monom-mult c s* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*) **by** *simp*
also from *monom-mult-assoc*[*of c s lookup p* (*t* ⊕ *lt f*) / *lc f t f*] *assoc*
have *monom-mult c s* (*monom-mult* (*lookup p* (*t* ⊕ *lt f*) / *lc f*) *t f*) =
monom-mult ((*lookup* (*monom-mult c s p*) ((*s* + *t*) ⊕ *lt f*)) / *lc f*) (*s* + *t*) *f*

by (*simp add*: *lookup-monom-mult-plus*)
finally show *?thesis* **.**
qed
from ‹*f* ≠ *0*› *g2 g3* **show** *?thesis* **unfolding** *red-single-def* **by** *auto*
qed

lemma *red-single-plus-1*:
assumes *red-single p q f t* **and** *t* ⊕ *lt f* ∉ *keys* (*p* + *r*)
shows *red-single* (*q* + *r*) (*p* + *r*) *f t*
proof −
from *assms* **have** *f* ≠ *0* **and** *lookup p* (*t* ⊕ *lt f*) ≠ *0*
and *q*: *q* = *p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
by (*simp-all add*: *red-single-def*)
from *assms*(*1*) **have** *cq-0*: *lookup q* (*t* ⊕ *lt f*) = *0* **by** (*rule red-single-lookup*)
from *assms*(*2*) **have** *lookup* (*p* + *r*) (*t* ⊕ *lt f*) = *0*
by (*simp add*: *in-keys-iff*)
with *neg-eq-iff-add-eq-0*[*of lookup p* (*t* ⊕ *lt f*) *lookup r* (*t* ⊕ *lt f*)]
have *cr*: *lookup r* (*t* ⊕ *lt f*) = − (*lookup p* (*t* ⊕ *lt f*)) **by** (*simp add*: *lookup-add*)
hence *cr-not-0*: *lookup r* (*t* ⊕ *lt f*) ≠ *0* **using** ‹*lookup p* (*t* ⊕ *lt f*) ≠ *0*› **by** *simp*
from ‹*f* ≠ *0*› **show** *?thesis* **unfolding** *red-single-def*
proof (*intro conjI*)
from *cr-not-0* **show** *lookup* (*q* + *r*) (*t* ⊕ *lt f*) ≠ *0* **by** (*simp add*: *lookup-add cq-0*)
next
from *lc-not-0*[*OF* ‹*f* ≠ *0*›]
have *monom-mult* ((*lookup* (*q* + *r*) (*t* ⊕ *lt f*)) / *lc f*) *t f* =
monom-mult ((*lookup r* (*t* ⊕ *lt f*)) / *lc f*) *t f*
by (*simp add*: *field-simps lookup-add cq-0*)
thus *p* + *r* = *q* + *r* − *monom-mult* (*lookup* (*q* + *r*) (*t* ⊕ *lt f*) / *lc f*) *t f*
by (*simp add*: *cr q monom-mult-uminus-left*)
qed
qed

lemma *red-single-plus-2*:
assumes *red-single p q f t* **and** *t* ⊕ *lt f* ∉ *keys* (*q* + *r*)
shows *red-single* (*p* + *r*) (*q* + *r*) *f t*
proof −
from *assms* **have** *f* ≠ *0* **and** *cp*: *lookup p* (*t* ⊕ *lt f*) ≠ *0*
and *q*: *q* = *p* − *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
by (*simp-all add*: *red-single-def*)
from *assms*(*1*) **have** *cq-0*: *lookup q* (*t* ⊕ *lt f*) = *0* **by** (*rule red-single-lookup*)
with *assms*(*2*) **have** *cr-0*: *lookup r* (*t* ⊕ *lt f*) = *0*
by (*simp add*: *lookup-add in-keys-iff*)
from ‹*f* ≠ *0*› **show** *?thesis* **unfolding** *red-single-def*
proof (*intro conjI*)
from *cp* **show** *lookup* (*p* + *r*) (*t* ⊕ *lt f*) ≠ *0* **by** (*simp add*: *lookup-add cr-0*)
next
show *q* + *r* = *p* + *r* − *monom-mult* (*lookup* (*p* + *r*) (*t* ⊕ *lt f*) / *lc f*) *t f*
by (*simp add*: *cr-0 q lookup-add*)

  qed
qed

lemma *red-single-plus-3*:
  assumes *red-single p q f t* **and** *t* ⊕ *lt f* ∈ *keys* (*p* + *r*) **and** *t* ⊕ *lt f* ∈ *keys* (*q* + *r*)
  shows ∃ *s*. *red-single* (*p* + *r*) *s f t* ∧ *red-single* (*q* + *r*) *s f t*
proof −
  let *?t* = *t* ⊕ *lt f*
  from *assms* **have** *f* ≠ *0* **and** *lookup p ?t* ≠ *0*
    and *q*: *q* = *p* − *monom-mult* ((*lookup p ?t*) / *lc f*) *t f*
    by (*simp-all add*: *red-single-def*)
  from *assms*(*2*) **have** *cpr*: *lookup* (*p* + *r*) *?t* ≠ *0* **by** (*simp add*: *in-keys-iff*)
  from *assms*(*3*) **have** *cqr*: *lookup* (*q* + *r*) *?t* ≠ *0* **by** (*simp add*: *in-keys-iff*)
  from *assms*(*1*) **have** *cq-0*: *lookup q ?t* = *0* **by** (*rule red-single-lookup*)
  let *?s* = (*p* + *r*) − *monom-mult* ((*lookup* (*p* + *r*) *?t*) / *lc f*) *t f*
  from ‹*f* ≠ *0*› *cpr* **have** *red-single* (*p* + *r*) *?s f t* **by** (*simp add*: *red-single-def*)
  moreover from ‹*f* ≠ *0*› **have** *red-single* (*q* + *r*) *?s f t* **unfolding** *red-single-def*
  proof (*intro conjI*)
    from *cqr* **show** *lookup* (*q* + *r*) *?t* ≠ *0* **.**
  next
    from *lc-not-0*[*OF* ‹*f* ≠ *0*›]
      monom-mult-dist-left[*of* (*lookup p ?t*) / *lc f* (*lookup r ?t*) / *lc f t f*]
      have *monom-mult* ((*lookup* (*p* + *r*) *?t*) / *lc f*) *t f* =
            (*monom-mult* ((*lookup p ?t*) / *lc f*) *t f*) +
              (*monom-mult* ((*lookup r ?t*) / *lc f*) *t f*)
        by (*simp add*: *field-simps lookup-add*)
    moreover from *lc-not-0*[*OF* ‹*f* ≠ *0*›]
      monom-mult-dist-left[*of* (*lookup q ?t*) / *lc f* (*lookup r ?t*) / *lc f t f*]
      have *monom-mult* ((*lookup* (*q* + *r*) *?t*) / *lc f*) *t f* =
            monom-mult ((*lookup r ?t*) / *lc f*) *t f*
        by (*simp add*: *field-simps lookup-add cq-0*)
    ultimately show *p* + *r* − *monom-mult* (*lookup* (*p* + *r*) *?t* / *lc f*) *t f* =
                q + *r* − *monom-mult* (*lookup* (*q* + *r*) *?t* / *lc f*) *t f* **by** (*simp add*: *q*)
  qed
  ultimately show *?thesis* **by** *auto*
qed

lemma *red-single-plus*:
  assumes *red-single p q f t*
  shows *red-single* (*p* + *r*) (*q* + *r*) *f t* ∨
        red-single (*q* + *r*) (*p* + *r*) *f t* ∨
        (∃ *s*. *red-single* (*p* + *r*) *s f t* ∧ *red-single* (*q* + *r*) *s f t*) (**is** *?A* ∨ *?B* ∨ *?C*)
proof (*cases t* ⊕ *lt f* ∈ *keys* (*p* + *r*))
  case *True*
  show *?thesis*
  proof (*cases t* ⊕ *lt f* ∈ *keys* (*q* + *r*))
    case *True*

```
    with assms ‹t ⊕ lt f ∈ keys (p + r)› have ?C by (rule red-single-plus-3)
    thus ?thesis by simp
  next
    case False
    with assms have ?A by (rule red-single-plus-2)
    thus ?thesis ..
  qed
next
  case False
  with assms have ?B by (rule red-single-plus-1)
  thus ?thesis by simp
qed

lemma red-single-diff:
  assumes red-single (p − q) r f t
  shows red-single p (r + q) f t ∨ red-single q (p − r) f t ∨
        (∃ p' q'. red-single p p' f t ∧ red-single q q' f t ∧ r = p' − q') (is ?A ∨ ?B
∨ ?C)
proof −
  let ?s = t ⊕ lt f
  from assms have f ≠ 0
    and lookup (p − q) ?s ≠ 0
    and r: r = p − q − monom-mult ((lookup (p − q) ?s) / lc f) t f
    unfolding red-single-def by auto
  from this(2) have diff: lookup p ?s ≠ lookup q ?s by (simp add: lookup-minus)
  show ?thesis
  proof (cases lookup p ?s = 0)
    case True
    with diff have ?s ∈ keys q by (simp add: in-keys-iff)
   moreover have lookup (p − q) ?s = − lookup q ?s by (simp add: lookup-minus
True)
    ultimately have ?B using ‹f ≠ 0› by (simp add: in-keys-iff red-single-def r
monom-mult-uminus-left)
    thus ?thesis by simp
  next
    case False
    hence ?s ∈ keys p by (simp add: in-keys-iff)
    show ?thesis
    proof (cases lookup q ?s = 0)
      case True
      hence lookup (p − q) ?s = lookup p ?s by (simp add: lookup-minus)
     hence ?A using ‹f ≠ 0› ‹?s ∈ keys p› by (simp add: in-keys-iff red-single-def
r monom-mult-uminus-left)
      thus ?thesis ..
    next
      case False
      hence ?s ∈ keys q by (simp add: in-keys-iff)
      let ?p = p − monom-mult ((lookup p ?s) / lc f) t f
      let ?q = q − monom-mult ((lookup q ?s) / lc f) t f
```

have *?C*
proof (*intro exI conjI*)
from ‹*f* ≠ *0*› ‹*?s* ∈ *keys p*› **show** *red-single p ?p f t* **by** (*simp add*: *in-keys-iff red-single-def*)
next
from ‹*f* ≠ *0*› ‹*?s* ∈ *keys q*› **show** *red-single q ?q f t* **by** (*simp add*: *in-keys-iff red-single-def*)
next
from ‹*f* ≠ *0*› **have** *lc f* ≠ *0* **by** (*rule lc-not-0*)
hence *eq*: (*lookup p ?s* − *lookup q ?s*) / *lc f* =
lookup p ?s / *lc f* − *lookup q ?s* / *lc f* **by** (*simp add*: *field-simps*)
show *r* = *?p* − *?q* **by** (*simp add*: *r lookup-minus eq monom-mult-dist-left-minus*)
qed
thus *?thesis* **by** *simp*
qed
qed
qed

lemma *red-monom-mult*:
assumes *a*: *red F p q* **and** *c* ≠ *0*
shows *red F* (*monom-mult c s p*) (*monom-mult c s q*)
proof −
from *red-setE*[*OF a*] **obtain** *f* **and** *t* **where** *f* ∈ *F* **and** *rs*: *red-single p q f t* **by** *auto*
from *red-single-monom-mult*[*OF rs* ‹*c* ≠ *0*›, *of s*] **show** *?thesis* **by** (*intro red-setI*[*OF* ‹*f* ∈ *F*›])
qed

lemma *red-plus-keys-disjoint*:
assumes *red F p q* **and** *keys p* ∩ *keys r* = {}
shows *red F* (*p* + *r*) (*q* + *r*)
proof −
from *assms*(*1*) **obtain** *f t* **where** *f* ∈ *F* **and** ∗: *red-single p q f t* **by** (*rule red-setE*)
from *this*(*2*) **have** *red-single* (*p* + *r*) (*q* + *r*) *f t*
proof (*rule red-single-plus-2*)
from ∗ **have** *lookup q* (*t* ⊕ *lt f*) = *0*
by (*simp add*: *red-single-def lookup-minus lookup-monom-mult lc-def*[*symmetric*] *lc-not-0 term-simps*)
hence *t* ⊕ *lt f* ∉ *keys q* **by** (*simp add*: *in-keys-iff*)
moreover have *t* ⊕ *lt f* ∉ *keys r*
proof
assume *t* ⊕ *lt f* ∈ *keys r*
moreover from ∗ **have** *t* ⊕ *lt f* ∈ *keys p* **by** (*simp add*: *in-keys-iff red-single-def*)
ultimately have *t* ⊕ *lt f* ∈ *keys p* ∩ *keys r* **by** *simp*
with *assms*(*2*) **show** *False* **by** *simp*
qed
ultimately have *t* ⊕ *lt f* ∉ *keys q* ∪ *keys r* **by** *simp*
thus *t* ⊕ *lt f* ∉ *keys* (*q* + *r*)

by (*meson Poly-Mapping.keys-add subsetD*)
qed
with ‹$f \in F$› **show** *?thesis* **by** (*rule red-setI*)
qed

lemma *red-plus*:
assumes *red F p q*
obtains *s* **where** (*red F*)** ($p + r$) *s* **and** (*red F*)** ($q + r$) *s*
proof −
from *red-setE*[*OF assms*] **obtain** *f* **and** *t* **where** $f \in F$ **and** *rs*: *red-single p q f t* **by** *auto*
from *red-single-plus*[*OF rs*, *of r*] **show** *?thesis*
proof
assume *c1*: *red-single* ($p + r$) ($q + r$) *f t*
show *?thesis*
proof
from *c1* **show** (*red F*)** ($p + r$) ($q + r$) **by** (*intro r-into-rtranclp*, *intro red-setI*[*OF* ‹$f \in F$›])
next
show (*red F*)** ($q + r$) ($q + r$) **..**
qed
next
assume *red-single* ($q + r$) ($p + r$) *f t* ∨ (∃ *s*. *red-single* ($p + r$) *s f t* ∧ *red-single* ($q + r$) *s f t*)
thus *?thesis*
proof
assume *c2*: *red-single* ($q + r$) ($p + r$) *f t*
show *?thesis*
proof
show (*red F*)** ($p + r$) ($p + r$) **..**
next
from *c2* **show** (*red F*)** ($q + r$) ($p + r$) **by** (*intro r-into-rtranclp*, *intro red-setI*[*OF* ‹$f \in F$›])
qed
next
assume ∃ *s*. *red-single* ($p + r$) *s f t* ∧ *red-single* ($q + r$) *s f t*
then obtain *s* **where** *s1*: *red-single* ($p + r$) *s f t* **and** *s2*: *red-single* ($q + r$) *s f t* **by** *auto*
show *?thesis*
proof
from *s1* **show** (*red F*)** ($p + r$) *s* **by** (*intro r-into-rtranclp*, *intro red-setI*[*OF* ‹$f \in F$›])
next
from *s2* **show** (*red F*)** ($q + r$) *s* **by** (*intro r-into-rtranclp*, *intro red-setI*[*OF* ‹$f \in F$›])
qed
qed
qed
qed

```
corollary red-plus-cs:
  assumes red F p q
  shows relation.cs (red F) (p + r) (q + r)
  unfolding relation.cs-def
proof −
  from assms obtain s where (red F)** (p + r) s and (red F)** (q + r) s by
(rule red-plus)
  show ∃ s. (red F)** (p + r) s ∧ (red F)** (q + r) s by (intro exI, intro conjI,
fact, fact)
qed

lemma red-uminus:
  assumes red F p q
  shows red F (−p) (−q)
  using red-monom-mult[OF assms, of −1 0] by (simp add: uminus-monom-mult)

lemma red-diff:
  assumes red F (p − q) r
  obtains p′ q′ where (red F)** p p′ and (red F)** q q′ and r = p′ − q′
proof −
  from assms obtain f t where f ∈ F and red-single (p − q) r f t by (rule
red-setE)
  from red-single-diff[OF this(2)] show ?thesis
  proof (elim disjE)
    assume red-single p (r + q) f t
    with ‹f ∈ F› have ∗: red F p (r + q) by (rule red-setI)
    show ?thesis
    proof
      from ∗ show (red F)** p (r + q) ..
    next
      show (red F)** q q ..
    qed simp
  next
    assume red-single q (p − r) f t
    with ‹f ∈ F› have ∗: red F q (p − r) by (rule red-setI)
    show ?thesis
    proof
      show (red F)** p p ..
    next
      from ∗ show (red F)** q (p − r) ..
    qed simp
  next
    assume ∃ p′ q′. red-single p p′ f t ∧ red-single q q′ f t ∧ r = p′ − q′
    then obtain p′ q′ where 1: red-single p p′ f t and 2: red-single q q′ f t and
r = p′ − q′
      by blast
    from ‹f ∈ F› 2 have red F q q′ by (rule red-setI)
    from ‹f ∈ F› 1 have red F p p′ by (rule red-setI)
```

hence (*red F*)** *p p′* ..
moreover from ‹*red F q q′*› **have** (*red F*)** *q q′* ..
moreover note ‹$r = p' - q'$›
ultimately show *?thesis* ..
qed
qed

lemma *red-diff-rtrancl′*:
assumes (*red F*)** $(p - q)$ *r*
obtains *p′ q′* **where** (*red F*)** *p p′* **and** (*red F*)** *q q′* **and** $r = p' - q'$
using *assms*
proof (*induct arbitrary*: *thesis rule*: *rtranclp-induct*)
case *base*
show *?case* **by** (*rule base*, *fact rtrancl-refl*[*to-pred*], *fact rtrancl-refl*[*to-pred*], *fact refl*)
next
case (*step y z*)
obtain *p1 q1* **where** *p1*: (*red F*)** *p p1* **and** *q1*: (*red F*)** *q q1* **and** *y*: $y = p1 - q1$ **by** (*rule step*(*3*))
from *step*(*2*) **obtain** *p′ q′* **where** *p′*: (*red F*)** *p1 p′* **and** *q′*: (*red F*)** *q1 q′* **and** *z*: $z = p' - q'$
unfolding *y* **by** (*rule red-diff*)
show *?case*
proof (*rule step*(*4*))
from *p1 p′* **show** (*red F*)** *p p′* **by** *simp*
next
from *q1 q′* **show** (*red F*)** *q q′* **by** *simp*
qed *fact*
qed

lemma *red-diff-rtrancl*:
assumes (*red F*)** $(p - q)$ *0*
obtains *s* **where** (*red F*)** *p s* **and** (*red F*)** *q s*
proof −
from *assms* **obtain** *p′ q′* **where** *p′*: (*red F*)** *p p′* **and** *q′*: (*red F*)** *q q′* **and** $0 = p' - q'$
by (*rule red-diff-rtrancl′*)
from *this*(*3*) **have** $q' = p'$ **by** *simp*
from *p′ q′* **show** *?thesis* **unfolding** ‹$q' = p'$› ..
qed

corollary *red-diff-rtrancl-cs*:
assumes (*red F*)** $(p - q)$ *0*
shows *relation.cs* (*red F*) *p q*
unfolding *relation.cs-def*
proof −
from *assms* **obtain** *s* **where** (*red F*)** *p s* **and** (*red F*)** *q s* **by** (*rule red-diff-rtrancl*)
show ∃ *s*. (*red F*)** *p s* ∧ (*red F*)** *q s* **by** (*intro exI*, *intro conjI*, *fact*, *fact*)
qed

4.3 Confluence of Reducibility

lemma *confluent-distinct-aux*:
  assumes *r1*: *red-single p q1 f1 t1* **and** *r2*: *red-single p q2 f2 t2*
    and $t1 \oplus lt\ f1 \prec_t t2 \oplus lt\ f2$ **and** $f1 \in F$ **and** $f2 \in F$
  obtains *s* **where** $(red\ F)^{**}\ q1\ s$ **and** $(red\ F)^{**}\ q2\ s$
proof −
  from *r1* **have** $f1 \neq 0$ **and** *c1*: $lookup\ p\ (t1 \oplus lt\ f1) \neq 0$
    and *q1-def*: $q1 = p - monom\text{-}mult\ (lookup\ p\ (t1 \oplus lt\ f1)\ /\ lc\ f1)\ t1\ f1$
    unfolding *red-single-def* **by** *auto*
  from *r2* **have** $f2 \neq 0$ **and** *c2*: $lookup\ p\ (t2 \oplus lt\ f2) \neq 0$
    and *q2-def*: $q2 = p - monom\text{-}mult\ (lookup\ p\ (t2 \oplus lt\ f2)\ /\ lc\ f2)\ t2\ f2$
    unfolding *red-single-def* **by** *auto*
  from ‹$t1 \oplus lt\ f1 \prec_t t2 \oplus lt\ f2$›
  have $lookup\ (monom\text{-}mult\ (lookup\ p\ (t1 \oplus lt\ f1)\ /\ lc\ f1)\ t1\ f1)\ (t2 \oplus lt\ f2) = 0$
    by (*simp add*: *lookup-monom-mult-eq-zero*)
  from *lookup-minus*[*of p - t2* ⊕ *lt f2*] *this* **have** *c*: $lookup\ q1\ (t2 \oplus lt\ f2) = lookup\ p\ (t2 \oplus lt\ f2)$
    unfolding *q1-def* **by** *simp*
  define *q3* **where** $q3 \equiv q1 - monom\text{-}mult\ ((lookup\ q1\ (t2 \oplus lt\ f2))\ /\ lc\ f2)\ t2\ f2$
  have *red-single q1 q3 f2 t2* **unfolding** *red-single-def*
  proof (*rule*, *fact*, *rule*)
    from *c c2* **show** $lookup\ q1\ (t2 \oplus lt\ f2) \neq 0$ **by** *simp*
  next
    show $q3 = q1 - monom\text{-}mult\ (lookup\ q1\ (t2 \oplus lt\ f2)\ /\ lc\ f2)\ t2\ f2$ **unfolding** *q3-def* **..**
  qed
  hence *red F q1 q3* **by** (*intro red-setI*[*OF* ‹$f2 \in F$›])
  hence *q1q3*: $(red\ F)^{**}\ q1\ q3$ **by** (*intro r-into-rtranclp*)
  from *r1* **have** *red F p q1* **by** (*intro red-setI*[*OF* ‹$f1 \in F$›])
  from *red-plus*[*OF this*, *of* − *monom-mult* ((*lookup p* (*t2* ⊕ *lt f2*)) / *lc f2*) *t2 f2*]
obtain *s*
    where *r3*: $(red\ F)^{**}\ (p - monom\text{-}mult\ (lookup\ p\ (t2 \oplus lt\ f2)\ /\ lc\ f2)\ t2\ f2)\ s$
    and *r4*: $(red\ F)^{**}\ (q1 - monom\text{-}mult\ (lookup\ p\ (t2 \oplus lt\ f2)\ /\ lc\ f2)\ t2\ f2)\ s$
by *auto*
  from *r3* **have** *q2s*: $(red\ F)^{**}\ q2\ s$ **unfolding** *q2-def* **by** *simp*
  from *r4 c* **have** *q3s*: $(red\ F)^{**}\ q3\ s$ **unfolding** *q3-def* **by** *simp*
  show *?thesis*
  proof
    from *rtranclp-trans*[*OF q1q3 q3s*] **show** $(red\ F)^{**}\ q1\ s$ **.**
  next
    from *q2s* **show** $(red\ F)^{**}\ q2\ s$ **.**
  qed
qed

lemma *confluent-distinct*:
  assumes *r1*: *red-single p q1 f1 t1* **and** *r2*: *red-single p q2 f2 t2*
    and *ne*: $t1 \oplus lt\ f1 \neq t2 \oplus lt\ f2$ **and** $f1 \in F$ **and** $f2 \in F$
  obtains *s* **where** $(red\ F)^{**}\ q1\ s$ **and** $(red\ F)^{**}\ q2\ s$

proof −
  from *ne* **have** $t1 \oplus lt\ f1 \prec_t t2 \oplus lt\ f2 \lor t2 \oplus lt\ f2 \prec_t t1 \oplus lt\ f1$ **by** *auto*
  thus *?thesis*
  proof
    assume *a1*: $t1 \oplus lt\ f1 \prec_t t2 \oplus lt\ f2$
    from *confluent-distinct-aux*[*OF r1 r2 a1* ‹*f1* ∈ *F*› ‹*f2* ∈ *F*›] **obtain** *s* **where**
      (*red F*)** *q1 s* **and** (*red F*)** *q2 s* **.**
    thus *?thesis* **..**
  next
    assume *a2*: $t2 \oplus lt\ f2 \prec_t t1 \oplus lt\ f1$
    from *confluent-distinct-aux*[*OF r2 r1 a2* ‹*f2* ∈ *F*› ‹*f1* ∈ *F*›] **obtain** *s* **where**
      (*red F*)** *q1 s* **and** (*red F*)** *q2 s* **.**
    thus *?thesis* **..**
  qed
qed

corollary *confluent-same*:
  assumes *r1*: *red-single p q1 f t1* **and** *r2*: *red-single p q2 f t2* **and** *f* ∈ *F*
  obtains *s* **where** (*red F*)** *q1 s* **and** (*red F*)** *q2 s*
proof (*cases t1* = *t2*)
  case *True*
  with *r1 r2* **have** *q1* = *q2* **by** (*simp add*: *red-single-def*)
  show *?thesis*
  proof
    show (*red F*)** *q1 q2* **unfolding** ‹*q1* = *q2*› **..**
  next
    show (*red F*)** *q2 q2* **..**
  qed
next
  case *False*
  hence $t1 \oplus lt\ f \neq t2 \oplus lt\ f$ **by** (*simp add*: *term-simps*)
  from *r1 r2 this* ‹*f* ∈ *F*› ‹*f* ∈ *F*› **obtain** *s* **where** (*red F*)** *q1 s* **and** (*red F*)**
q2 s
    by (*rule confluent-distinct*)
  thus *?thesis* **..**
qed

4.4 Reducibility and Module Membership

lemma *srtc-in-pmdl*:
  assumes *relation.srtc* (*red F*) *p q*
  shows *p* − *q* ∈ *pmdl F*
  using *assms* **unfolding** *relation.srtc-def*
proof (*induct rule*: *rtranclp.induct*)
  fix *p*
  show *p* − *p* ∈ *pmdl F* **by** (*simp add*: *pmdl.span-zero*)
next
  fix *p r q*
  assume *pr-in*: *p* − *r* ∈ *pmdl F* **and** *red*: *red F r q* ∨ *red F q r*

```
  from red obtain f c t where f ∈ F and q = r − monom-mult c t f
  proof
    assume red F r q
    from red-setE[OF this] obtain f t where f ∈ F and red-single r q f t .
     hence q = r − monom-mult (lookup r (t ⊕ lt f) / lc f) t f by (simp add:
red-single-def)
    show thesis by (rule, fact, fact)
  next
    assume red F q r
    from red-setE[OF this] obtain f t where f ∈ F and red-single q r f t .
     hence r = q − monom-mult (lookup q (t ⊕ lt f) / lc f) t f by (simp add:
red-single-def)
    hence q = r + monom-mult (lookup q (t ⊕ lt f) / lc f) t f by simp
    hence q = r − monom-mult (−(lookup q (t ⊕ lt f) / lc f)) t f
      using monom-mult-uminus-left[of - t f] by simp
    show thesis by (rule, fact, fact)
  qed
  hence eq: p − q = (p − r) + monom-mult c t f by simp
  show p − q ∈ pmdl F unfolding eq
    by (rule pmdl.span-add, fact, rule monom-mult-in-pmdl, fact)
qed

lemma in-pmdl-srtc:
  assumes p ∈ pmdl F
  shows relation.srtc (red F) p 0
  using assms
proof (induct p rule: pmdl-induct)
  show relation.srtc (red F) 0 0 unfolding relation.srtc-def ..
next
  fix a f c t
  assume a-in: a ∈ pmdl F and IH: relation.srtc (red F) a 0 and f ∈ F
  show relation.srtc (red F) (a + monom-mult c t f) 0
  proof (cases c = 0)
    assume c = 0
    hence a + monom-mult c t f = a by simp
    thus ?thesis using IH by simp
  next
    assume c ≠ 0
    show ?thesis
    proof (cases f = 0)
      assume f = 0
      hence a + monom-mult c t f = a by simp
      thus ?thesis using IH by simp
    next
      assume f ≠ 0
      from lc-not-0[OF this] have lc f ≠ 0 .
      have red F (monom-mult c t f) 0
      proof (intro red-setI[OF ‹f ∈ F›])
        from lookup-monom-mult-plus[of c t f lt f]
```

```
        have eq: lookup (monom-mult c t f) (t ⊕ lt f) = c ∗ lc f unfolding lc-def
.
        show red-single (monom-mult c t f) 0 f t unfolding red-single-def eq
        proof (intro conjI, fact)
          from ‹c ≠ 0› ‹lc f ≠ 0› show c ∗ lc f ≠ 0 by simp
        next
          from ‹lc f ≠ 0› show 0 = monom-mult c t f − monom-mult (c ∗ lc f /
lc f) t f by simp
        qed
      qed
      from red-plus[OF this, of a] obtain s where
        s1: (red F)** (monom-mult c t f + a) s and s2: (red F)** (0 + a) s .
      have relation.cs (red F) (a + monom-mult c t f) a unfolding relation.cs-def
      proof (intro exI[of - s], intro conjI)
          from s1 show (red F)** (a + monom-mult c t f) s by (simp only:
add.commute)
      next
        from s2 show (red F)** a s by simp
      qed
      from relation.srtc-transitive[OF relation.cs-implies-srtc[OF this] IH] show
?thesis .
    qed
  qed
qed

lemma red-rtranclp-diff-in-pmdl:
  assumes (red F)** p q
  shows p − q ∈ pmdl F
proof −
  from assms have relation.srtc (red F) p q
    by (simp add: r-into-rtranclp relation.rtc-implies-srtc)
  thus ?thesis by (rule srtc-in-pmdl)
qed

corollary red-diff-in-pmdl:
  assumes red F p q
  shows p − q ∈ pmdl F
  by (rule red-rtranclp-diff-in-pmdl, rule r-into-rtranclp, fact)

corollary red-rtranclp-0-in-pmdl:
  assumes (red F)** p 0
  shows p ∈ pmdl F
  using assms red-rtranclp-diff-in-pmdl by fastforce

lemma pmdl-closed-red:
  assumes pmdl B ⊆ pmdl A and p ∈ pmdl A and red B p q
  shows q ∈ pmdl A
proof −
  have q − p ∈ pmdl A
```

proof
have *p* − *q* ∈ *pmdl B* **by** (*rule red-diff-in-pmdl*, *fact*)
hence − (*p* − *q*) ∈ *pmdl B* **by** (*rule pmdl.span-neg*)
thus *q* − *p* ∈ *pmdl B* **by** *simp*
qed *fact*
from *pmdl.span-add*[*OF this* ‹*p* ∈ *pmdl A*›] **show** *?thesis* **by** *simp*
qed

4.5 More Properties of *red*, *red-single* and *is-red*

lemma *red-rtrancl-mult*:
assumes (*red F*)** *p q*
shows (*red F*)** (*monom-mult c t p*) (*monom-mult c t q*)
proof (*cases c* = *0*)
case *True*
have (*red F*)** *0 0* **by** *simp*
thus *?thesis* **by** (*simp only*: *True monom-mult-zero-left*)
next
case *False*
from *assms* **show** *?thesis*
proof (*induct rule*: *rtranclp-induct*)
show (*red F*)** (*monom-mult c t p*) (*monom-mult c t p*) **by** *simp*
next
fix *q0 q*
assume (*red F*)** *p q0* **and** *red F q0 q* **and** (*red F*)** (*monom-mult c t p*) (*monom-mult c t q0*)
show (*red F*)** (*monom-mult c t p*) (*monom-mult c t q*)
proof (*rule rtranclp.intros*(*2*)[*OF* ‹(*red F*)** (*monom-mult c t p*) (*monom-mult c t q0*)›])
from *red-monom-mult*[*OF* ‹*red F q0 q*› *False*, *of t*] **show** *red F* (*monom-mult c t q0*) (*monom-mult c t q*) **.**
qed
qed
qed

corollary *red-rtrancl-uminus*:
assumes (*red F*)** *p q*
shows (*red F*)** (−*p*) (−*q*)
using *red-rtrancl-mult*[*OF assms*, *of* −*1 0*] **by** (*simp add*: *uminus-monom-mult*)

lemma *red-rtrancl-diff-induct* [*consumes 1*, *case-names base step*]:
assumes *a*: (*red F*)** (*p* − *q*) *r*
and *cases*: *P p p* !!*y z*. [| (*red F*)** (*p* − *q*) *z*; *red F z y*; *P p* (*q* + *z*)|] ==> *P p* (*q* + *y*)
shows *P p* (*q* + *r*)
using *a*
proof (*induct rule*: *rtranclp-induct*)
from *cases*(*1*) **show** *P p* (*q* + (*p* − *q*)) **by** *simp*
next

fix *y z*
assume (*red F*)** (*p* − *q*) *z red F z y P p* (*q* + *z*)
thus *P p* (*q* + *y*) **using** *cases*(*2*) **by** *simp*
qed

lemma *red-rtrancl-diff-0-induct* [*consumes 1*, *case-names base step*]:
assumes *a*: (*red F*)** (*p* − *q*) *0*
and *base*: *P p p* **and** *ind*: ⋀*y z*. [| (*red F*)** (*p* − *q*) *y*; *red F y z*; *P p* (*y* + *q*)|] ==> *P p* (*z* + *q*)
shows *P p q*
proof −
from *ind red-rtrancl-diff-induct*[*of F p q 0 P*, *OF a base*] **have** *P p* (*0* + *q*)
by (*simp add*: *ac-simps*)
thus *?thesis* **by** *simp*
qed

lemma *is-red-union*: *is-red* (*A* ∪ *B*) *p* ⟷ (*is-red A p* ∨ *is-red B p*)
unfolding *is-red-alt red-union* **by** *auto*

lemma *red-single-0-lt*:
assumes *red-single f 0 h t*
shows *lt f* = *t* ⊕ *lt h*
proof −
from *red-single-nonzero1*[*OF assms*] **have** *f* ≠ *0* **.**
{
assume *h* ≠ *0* **and** *neq*: *lookup f* (*t* ⊕ *lt h*) ≠ *0* **and**
eq: *f* = *monom-mult* (*lookup f* (*t* ⊕ *lt h*) / *lc h*) *t h*
from *lc-not-0*[*OF* ‹*h* ≠ *0*›] **have** *lc h* ≠ *0* **.**
with *neq* **have** (*lookup f* (*t* ⊕ *lt h*) / *lc h*) ≠ *0* **by** *simp*
from *eq lt-monom-mult*[*OF this* ‹*h* ≠ *0*›, *of t*] **have** *lt f* = *t* ⊕ *lt h* **by** *simp*
hence *lt f* = *t* ⊕ *lt h* **by** (*simp add*: *ac-simps*)
}
with *assms* **show** *?thesis* **unfolding** *red-single-def* **by** *auto*
qed

lemma *red-single-lt-distinct-lt*:
assumes *rs*: *red-single f g h t* **and** *g* ≠ *0* **and** *lt g* ≠ *lt f*
shows *lt f* = *t* ⊕ *lt h*
proof −
from *red-single-nonzero1*[*OF rs*] **have** *f* ≠ *0* **.**
from *red-single-ord*[*OF rs*] **have** *g* $\preceq_p$ *f* **by** *simp*
from *ord-p-lt*[*OF this*] ‹*lt g* ≠ *lt f*› **have** *lt g* $\prec_t$ *lt f* **by** *simp*
{
assume *h* ≠ *0* **and** *neq*: *lookup f* (*t* ⊕ *lt h*) ≠ *0* **and**
eq: *f* = *g* + *monom-mult* (*lookup f* (*t* ⊕ *lt h*) / *lc h*) *t h* (**is** *f* = *g* + *?R*)
from *lc-not-0*[*OF* ‹*h* ≠ *0*›] **have** *lc h* ≠ *0* **.**
with *neq* **have** (*lookup f* (*t* ⊕ *lt h*) / *lc h*) ≠ *0* (**is** *?c* ≠ *0*) **by** *simp*
from *eq lt-monom-mult*[*OF this* ‹*h* ≠ *0*›, *of t*] **have** *ltR*: *lt ?R* = *t* ⊕ *lt h* **by** *simp*

from *monom-mult-eq-zero-iff*[*of ?c t h*] ‹*?c* ≠ *0*› ‹*h* ≠ *0*› **have** *?R* ≠ *0* **by** *auto*
from *lt-plus-lessE*[*of g*] *eq* ‹*lt g* $\prec_t$ *lt f*› **have** *lt g* $\prec_t$ *lt ?R* **by** *auto*
from *lt-plus-eqI*[*OF this*] *eq ltR* **have** *lt f* = *t* ⊕ *lt h* **by** (*simp add*: *ac-simps*)
}
with *assms* **show** *?thesis* **unfolding** *red-single-def* **by** *auto*
qed

lemma *zero-reducibility-implies-lt-divisibility'*:
assumes (*red F*)** *f 0* **and** *f* ≠ *0*
shows ∃ *h*∈*F*. *h* ≠ *0* ∧ (*lt h* $adds_t$ *lt f*)
using *assms*
proof (*induct rule*: *converse-rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step f g*)
show *?case*
proof (*cases g* = *0*)
case *True*
with *step.hyps* **have** *red F f 0* **by** *simp*
from *red-setE*[*OF this*] **obtain** *h t* **where** *h* ∈ *F* **and** *rs*: *red-single f 0 h t* **by** *auto*
show *?thesis*
proof
from *red-single-0-lt*[*OF rs*] **have** *lt h* $adds_t$ *lt f* **by** (*simp add*: *term-simps*)
also from *rs* **have** *h* ≠ *0* **by** (*simp add*: *red-single-def*)
ultimately show *h* ≠ *0* ∧ *lt h* $adds_t$ *lt f* **by** *simp*
qed (*rule* ‹*h* ∈ *F*›)
next
case *False*
show *?thesis*
proof (*cases lt g* = *lt f*)
case *True*
with *False step.hyps* **show** *?thesis* **by** *simp*
next
case *False*
from *red-setE*[*OF* ‹*red F f g*›] **obtain** *h t* **where** *h* ∈ *F* **and** *rs*: *red-single f g h t* **by** *auto*
show *?thesis*
proof
from *red-single-lt-distinct-lt*[*OF rs* ‹*g* ≠ *0*› *False*] **have** *lt h* $adds_t$ *lt f*
by (*simp add*: *term-simps*)
also from *rs* **have** *h* ≠ *0* **by** (*simp add*: *red-single-def*)
ultimately show *h* ≠ *0* ∧ *lt h* $adds_t$ *lt f* **by** *simp*
qed (*rule* ‹*h* ∈ *F*›)
qed
qed
qed

```
lemma zero-reducibility-implies-lt-divisibility:
  assumes (red F)** f 0 and f ≠ 0
  obtains h where h ∈ F and h ≠ 0 and lt h adds_t lt f
  using zero-reducibility-implies-lt-divisibility'[OF assms] by auto

lemma is-red-addsI:
  assumes f ∈ F and f ≠ 0 and v ∈ keys p and lt f adds_t v
  shows is-red F p
  using assms
proof (induction p rule: poly-mapping-tail-induct)
  case 0
  from ‹v ∈ keys 0› show ?case by auto
next
  case (tail p)
  from tail.IH[OF ‹f ∈ F› ‹f ≠ 0› - ‹lt f adds_t v›] have imp: v ∈ keys (tail p)
⟹ is-red F (tail p) .
  show ?case
  proof (cases v = lt p)
    case True
    show ?thesis
    proof (rule is-red-indI1[OF ‹f ∈ F› ‹f ≠ 0› ‹p ≠ 0›])
      from ‹lt f adds_t v› True show lt f adds_t lt p by simp
    qed
  next
    case False
    with ‹v ∈ keys p› ‹p ≠ 0› have v ∈ keys (tail p)
      by (simp add: lookup-tail-2 in-keys-iff)
    from is-red-indI2[OF ‹p ≠ 0› imp[OF this]] show ?thesis .
  qed
qed

lemma is-red-addsE':
  assumes is-red F p
  shows ∃f∈F. ∃v∈keys p. f ≠ 0 ∧ lt f adds_t v
  using assms
proof (induction p rule: poly-mapping-tail-induct)
  case 0
  with irred-0[of F] show ?case by simp
next
  case (tail p)
  from is-red-indE[OF ‹is-red F p›] show ?case
  proof
    assume ∃f∈F. f ≠ 0 ∧ lt f adds_t lt p
    then obtain f where f ∈ F and f ≠ 0 and lt f adds_t lt p by auto
    show ?case
    proof
      show ∃v∈keys p. f ≠ 0 ∧ lt f adds_t v
      proof (intro bexI, intro conjI)
```

    from ‹$p \neq 0$› **show** *lt p* ∈ *keys p* **by** (*metis in-keys-iff lc-def lc-not-0*)
   qed (*rule* ‹$f \neq 0$›, *rule* ‹*lt f* $adds_t$ *lt p*›)
  qed (*rule* ‹$f \in F$›)
 next
  assume *is-red F* (*tail p*)
  from *tail.IH*[*OF this*] **obtain** *f v*
   where $f \in F$ **and** $f \neq 0$ **and** *v-in-keys-tail*: *v* ∈ *keys* (*tail p*) **and** *lt f* $adds_t$ *v* **by** *auto*
  from *tail.hyps v-in-keys-tail* **have** *v-in-keys*: *v* ∈ *keys p* **by** (*metis lookup-tail in-keys-iff*)
  show *?case*
  proof
   show ∃ *v*∈*keys p*. $f \neq 0$ ∧ *lt f* $adds_t$ *v*
    by (*intro bexI*, *intro conjI*, *rule* ‹$f \neq 0$›, *rule* ‹*lt f* $adds_t$ *v*›, *rule v-in-keys*)
  qed (*rule* ‹$f \in F$›)
 qed
qed

lemma *is-red-addsE*:
 assumes *is-red F p*
 obtains *f v* **where** $f \in F$ **and** *v* ∈ *keys p* **and** $f \neq 0$ **and** *lt f* $adds_t$ *v*
 using *is-red-addsE′*[*OF assms*] **by** *auto*

lemma *is-red-adds-iff*:
 shows (*is-red F p*) ⟷ (∃ *f*∈*F*. ∃ *v*∈*keys p*. $f \neq 0$ ∧ *lt f* $adds_t$ *v*)
 using *is-red-addsE′ is-red-addsI* **by** *auto*

lemma *is-red-subset*:
 assumes *red*: *is-red A p* **and** *sub*: $A \subseteq B$
 shows *is-red B p*
proof −
 from *red* **obtain** *f v* **where** $f \in A$ **and** *v* ∈ *keys p* **and** $f \neq 0$ **and** *lt f* $adds_t$ *v* **by** (*rule is-red-addsE*)
 show *?thesis* **by** (*rule is-red-addsI*, *rule*, *fact+*)
qed

lemma *not-is-red-empty*: ¬ *is-red* {} *f*
 by (*simp add*: *is-red-adds-iff*)

lemma *red-single-mult-const*:
 assumes *red-single p q f t* **and** $c \neq 0$
 shows *red-single p q* (*monom-mult c 0 f*) *t*
proof −
 let *?s* = *t* ⊕ *lt f*
 let *?f* = *monom-mult c 0 f*
 from *assms*(*1*) **have** $f \neq 0$ **and** *lookup p ?s* ≠ *0*
   and *q* = *p* − *monom-mult* ((*lookup p ?s*) / *lc f*) *t f* **by** (*simp-all add*: *red-single-def*)
 from *this*(*1*) *assms*(*2*) **have** *lt*: *lt ?f* = *lt f* **and** *lc*: *lc ?f* = *c* ∗ *lc f*

by (*simp add*: *lt-monom-mult term-simps*, *simp*)
show *?thesis* **unfolding** *red-single-def*
proof (*intro conjI*)
from ‹$f \neq 0$› *assms*(*2*) **show** $?f \neq 0$ **by** (*simp add*: *monom-mult-eq-zero-iff*)
next
from ‹*lookup p ?s* $\neq 0$› **show** *lookup p* ($t \oplus$ *lt ?f*) $\neq 0$ **by** (*simp add*: *lt*)
next
show $q = p -$ *monom-mult* (*lookup p* ($t \oplus$ *lt ?f*) / *lc ?f*) *t ?f*
by (*simp add*: *lt monom-mult-assoc lc assms*(*2*), *fact*)
qed
qed

lemma *red-rtrancl-plus-higher*:
assumes (*red F*)** *p q* **and** $\bigwedge u\ v.\ u \in$ *keys p* $\Longrightarrow v \in$ *keys r* $\Longrightarrow u \prec_t v$
shows (*red F*)** ($p + r$) ($q + r$)
using *assms*(*1*)
proof *induct*
case *base*
show *?case* **..**
next
case (*step y z*)
from *step*(*1*) **have** $y \preceq_p p$ **by** (*rule red-rtrancl-ord*)
hence *lt y* $\preceq_t$ *lt p* **by** (*rule ord-p-lt*)
from *step*(*2*) **have** *red F* ($y + r$) ($z + r$)
proof (*rule red-plus-keys-disjoint*)
show *keys y* $\cap$ *keys r* = {}
proof (*rule ccontr*)
assume *keys y* $\cap$ *keys r* $\neq$ {}
then obtain *v* **where** $v \in$ *keys y* **and** $v \in$ *keys r* **by** *auto*
from *this*(*1*) **have** $v \preceq_t$ *lt y* **and** $y \neq 0$ **using** *lt-max* **by** (*auto simp*: *in-keys-iff*)
with ‹$y \preceq_p p$› **have** $p \neq 0$ **using** *ord-p-zero-min*[*of y*] **by** *auto*
hence *lt p* $\in$ *keys p* **by** (*rule lt-in-keys*)
from *this* ‹$v \in$ *keys r*› **have** *lt p* $\prec_t v$ **by** (*rule assms*(*2*))
with ‹*lt y* $\preceq_t$ *lt p*› **have** *lt y* $\prec_t v$ **by** *simp*
with ‹$v \preceq_t$ *lt y*› **show** *False* **by** *simp*
qed
qed
with *step*(*3*) **show** *?case* **..**
qed

lemma *red-mult-scalar-leading-monomial*: (*red* {*f*})** ($p \odot$ *monomial* (*lc f*) (*lt f*)) ($- p \odot$ *tail f*)
proof (*cases* $f = 0$)
case *True*
show *?thesis* **by** (*simp add*: *True lc-def*)
next
case *False*
show *?thesis*

proof (*induct p rule*: *punit.poly-mapping-tail-induct*)
case *0*
show *?case* **by** *simp*
next
case (*tail p*)
from *False* **have** $lc\ f \neq 0$ **by** (*rule lc-not-0*)
from *tail*(*1*) **have** $punit.lc\ p \neq 0$ **by** (*rule punit.lc-not-0*)
let $?t = punit.tail\ p \odot monomial\ (lc\ f)\ (lt\ f)$
let $?m = monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (monomial\ (lc\ f)\ (lt\ f))$
from ‹$lc\ f \neq 0$› **have** *kt*: $keys\ ?t = (\lambda t.\ t \oplus lt\ f)$ ‘ $keys\ (punit.tail\ p)$
by (*rule keys-mult-scalar-monomial-right*)
have *km*: $keys\ ?m = \{punit.lt\ p \oplus lt\ f\}$
by (*simp add*: *keys-monom-mult*[*OF* ‹$punit.lc\ p \neq 0$›] ‹$lc\ f \neq 0$›)
from *tail*(*2*) **have** $(red\ \{f\})^{**}\ (?t + ?m)\ (-\ punit.tail\ p \odot tail\ f + ?m)$
proof (*rule red-rtrancl-plus-higher*)
fix *u v*
assume $u \in keys\ ?t$ **and** $v \in keys\ ?m$
from *this*(*1*) **obtain** *s* **where** $s \in keys\ (punit.tail\ p)$ **and** *u*: $u = s \oplus lt\ f$ **unfolding** *kt* **..**
from *this*(*1*) **have** $punit.tail\ p \neq 0$ **and** $s \preceq punit.lt\ (punit.tail\ p)$ **using** *punit.lt-max* **by** (*auto simp*: *in-keys-iff*)
moreover from ‹$punit.tail\ p \neq 0$› **have** $punit.lt\ (punit.tail\ p) \prec punit.lt\ p$ **by** (*rule punit.lt-tail*)
ultimately have $s \prec punit.lt\ p$ **by** *simp*
moreover from ‹$v \in keys\ ?m$› **have** $v = punit.lt\ p \oplus lt\ f$ **by** (*simp only*: *km*, *simp*)
ultimately show $u \prec_t v$ **by** (*simp add*: *u splus-mono-strict-left*)
qed
hence $*$: $(red\ \{f\})^{**}\ (p \odot monomial\ (lc\ f)\ (lt\ f))\ (?m - punit.tail\ p \odot tail\ f)$
by (*simp add*: *punit.leading-monomial-tail*[*symmetric*, *of p*] *mult-scalar-monomial*[*symmetric*] *mult-scalar-distrib-right*[*symmetric*] *add.commute*[*of punit.tail p*])
have $red\ \{f\}\ ?m\ (-\ (monomial\ (punit.lc\ p)\ (punit.lt\ p)) \odot tail\ f)$ **unfolding** *red-singleton*
proof
show $red\text{-}single\ ?m\ (-\ (monomial\ (punit.lc\ p)\ (punit.lt\ p)) \odot tail\ f)\ f\ (punit.lt\ p)$
proof (*simp add*: *red-single-def* ‹$f \neq 0$› *km lookup-monom-mult* ‹$lc\ f \neq 0$› ‹$punit.lc\ p \neq 0$› *term-simps*,
simp add: *monom-mult-dist-right-minus*[*symmetric*] *mult-scalar-monomial*)
have $monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (monomial\ (lc\ f)\ (lt\ f) - f) =$
$-\ monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (f - monomial\ (lc\ f)\ (lt\ f))$
by (*metis minus-diff-eq monom-mult-uminus-right*)
also have $... = -\ monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (tail\ f)$ **by** (*simp only*: *tail-alt-2*)
finally show $-\ monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (tail\ f) =$
$monom\text{-}mult\ (punit.lc\ p)\ (punit.lt\ p)\ (monomial\ (lc\ f)\ (lt\ f) - f)$ **by** *simp*
qed
qed

hence *red* {*f*} (*?m* + (− *punit.tail p* ⊙ *tail f*))
(− (*monomial* (*punit.lc p*) (*punit.lt p*)) ⊙ *tail f* + (− *punit.tail p* ⊙ *tail f*))
proof (*rule red-plus-keys-disjoint*)
show *keys ?m* ∩ *keys* (− *punit.tail p* ⊙ *tail f*) = {}
proof (*cases punit.tail p* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *True*)
next
case *False*
from *tail*(*2*) **have** − *punit.tail p* ⊙ *tail f* $\preceq_p$ *?t* **by** (*rule red-rtrancl-ord*)
hence *lt* (− *punit.tail p* ⊙ *tail f*) $\preceq_t$ *lt ?t* **by** (*rule ord-p-lt*)
also from ‹*lc f* ≠ *0*› *False* **have** ... = *punit.lt* (*punit.tail p*) ⊕ *lt f*
by (*rule lt-mult-scalar-monomial-right*)
also from *punit.lt-tail*[*OF False*] **have** ... $\prec_t$ *punit.lt p* ⊕ *lt f* **by** (*rule splus-mono-strict-left*)
finally have *punit.lt p* ⊕ *lt f* ∉ *keys* (− *punit.tail p* ⊙ *tail f*) **using** *lt-gr-keys* **by** *blast*
thus *?thesis* **by** (*simp add*: *km*)
qed
qed
hence *red* {*f*} (*?m* − *punit.tail p* ⊙ *tail f*)
(− (*monomial* (*punit.lc p*) (*punit.lt p*)) ⊙ *tail f* − *punit.tail p* ⊙ *tail f*)
by (*simp add*: *term-simps*)
also have ... = − *p* ⊙ *tail f* **using** *punit.leading-monomial-tail*[*symmetric*, *of p*]
by (*metis* (*mono-tags*, *lifting*) *add-uminus-conv-diff minus-add-distrib mult-scalar-distrib-right mult-scalar-minus-mult-left*)
finally have *red* {*f*} (*?m* − *punit.tail p* ⊙ *tail f*) (− *p* ⊙ *tail f*) **.**
with ∗ **show** *?case* **..**
qed
qed

corollary *red-mult-scalar-lt*:
assumes *f* ≠ *0*
shows (*red* {*f*})** (*p* ⊙ *monomial c* (*lt f*)) (*monom-mult* (− *c* / *lc f*) *0* (*p* ⊙ *tail f*))
proof −
from *assms* **have** *lc f* ≠ *0* **by** (*rule lc-not-0*)
hence *1*: *p* ⊙ *monomial c* (*lt f*) = *punit.monom-mult* (*c* / *lc f*) *0 p* ⊙ *monomial* (*lc f*) (*lt f*)
by (*simp add*: *punit.mult-scalar-monomial*[*symmetric*] *mult.commute mult-scalar-assoc mult-scalar-monomial-monomial term-simps*)
have *2*: *monom-mult* (− *c* / *lc f*) *0* (*p* ⊙ *tail f*) = − *punit.monom-mult* (*c* / *lc f*) *0 p* ⊙ *tail f*
by (*simp add*: *times-monomial-left*[*symmetric*] *mult-scalar-assoc monom-mult-uminus-left mult-scalar-monomial*)
show *?thesis* **unfolding** *1 2* **by** (*fact red-mult-scalar-leading-monomial*)
qed

lemma *is-red-monomial-iff*: *is-red F* (*monomial c v*) ⟷ (*c* ≠ *0* ∧ (∃*f*∈*F*. *f* ≠ *0* ∧ *lt f* $adds_t$ *v*))
  by (*simp add*: *is-red-adds-iff*)

lemma *is-red-monomialI*:
  assumes *c* ≠ *0* **and** *f* ∈ *F* **and** *f* ≠ *0* **and** *lt f* $adds_t$ *v*
  shows *is-red F* (*monomial c v*)
  unfolding *is-red-monomial-iff* **using** *assms* **by** *blast*

lemma *is-red-monomialD*:
  assumes *is-red F* (*monomial c v*)
  shows *c* ≠ *0*
  using *assms* **unfolding** *is-red-monomial-iff* **..**

lemma *is-red-monomialE*:
  assumes *is-red F* (*monomial c v*)
  obtains *f* **where** *f* ∈ *F* **and** *f* ≠ *0* **and** *lt f* $adds_t$ *v*
  using *assms* **unfolding** *is-red-monomial-iff* **by** *blast*

lemma *replace-lt-adds-stable-is-red*:
  assumes *red*: *is-red F f* **and** *q* ≠ *0* **and** *lt q* $adds_t$ *lt p*
  shows *is-red* (*insert q* (*F* − {*p*})) *f*
proof −
  from *red* **obtain** *g v* **where** *g* ∈ *F* **and** *g* ≠ *0* **and** *v* ∈ *keys f* **and** *lt g* $adds_t$ *v*
    by (*rule is-red-addsE*)
  show *?thesis*
  proof (*cases g* = *p*)
    case *True*
    show *?thesis*
    proof (*rule is-red-addsI*)
      show *q* ∈ *insert q* (*F* − {*p*}) **by** *simp*
    next
      have *lt q* $adds_t$ *lt p* **by** *fact*
      also have ... $adds_t$ *v* **using** ‹*lt g* $adds_t$ *v*› **unfolding** *True* **.**
      finally show *lt q* $adds_t$ *v* **.**
    qed (*fact*+)
  next
    case *False*
    with ‹*g* ∈ *F*› **have** *g* ∈ *insert q* (*F* − {*p*}) **by** *blast*
   from *this* ‹*g* ≠ *0*› ‹*v* ∈ *keys f*› ‹*lt g* $adds_t$ *v*› **show** *?thesis* **by** (*rule is-red-addsI*)
  qed
qed

lemma *conversion-property*:
  assumes *is-red* {*p*} *f* **and** *red* {*r*} *p q*
  shows *is-red* {*q*} *f* ∨ *is-red* {*r*} *f*
proof −
  let *?s* = *lp p* − *lp r*

```
  from ‹is-red {p} f› obtain v where v ∈ keys f and lt p adds_t v and p ≠ 0
    by (rule is-red-addsE, simp)
  from red-indE[OF ‹red {r} p q›]
    have (r ≠ 0 ∧ lt r adds_t lt p ∧ q = p − monom-mult (lc p / lc r) ?s r) ∨
          red {r} (tail p) (q − monomial (lc p) (lt p)) by simp
  thus ?thesis
  proof
    assume r ≠ 0 ∧ lt r adds_t lt p ∧ q = p − monom-mult (lc p / lc r) ?s r
    hence r ≠ 0 and lt r adds_t lt p by simp-all
     show ?thesis by (intro disjI2, rule is-red-singleton-trans, rule ‹is-red {p} f›,
fact+)
  next
    assume red {r} (tail p) (q − monomial (lc p) (lt p)) (is red - ?p' ?q')
    with red-ord have ?q' ≺_p ?p' .
    hence ?p' ≠ 0
      and assm: (?q' = 0 ∨ ((lt ?q') ≺_t (lt ?p') ∨ (lt ?q') = (lt ?p')))
      unfolding ord-strict-p-rec[of ?q' ?p'] by (auto simp add: Let-def lc-def)
    have lt ?p' ≺_t lt p by (rule lt-tail, fact)
    let ?m = monomial (lc p) (lt p)
    from monomial-0D[of lt p lc p] lc-not-0[OF ‹p ≠ 0›] have ?m ≠ 0 by blast
    have lt ?m = lt p by (rule lt-monomial, rule lc-not-0, fact)
    have q ≠ 0 ∧ lt q = lt p
    proof (cases ?q' = 0)
      case True
      hence q = ?m by simp
      with ‹?m ≠ 0› ‹lt ?m = lt p› show ?thesis by simp
    next
      case False
      from assm show ?thesis
      proof
        assume (lt ?q') ≺_t (lt ?p') ∨ (lt ?q') = (lt ?p')
        hence lt ?q' ⪯_t lt ?p' by auto
        also have ... ≺_t lt p by fact
        finally have lt ?q' ≺_t lt p .
        hence lt ?q' ≺_t lt ?m unfolding ‹lt ?m = lt p› .
        from lt-plus-eqI[OF this] ‹lt ?m = lt p› have lt q = lt p by simp
        show ?thesis
        proof (intro conjI, rule ccontr)
          assume ¬ q ≠ 0
          hence q = 0 by simp
          hence ?q' = −?m by simp
          hence lt ?q' = lt (−?m) by simp
          also have ... = lt ?m using lt-uminus .
          finally have lt ?q' = lt ?m .
          with ‹lt ?q' ≺_t lt ?m› show False by simp
        qed (fact)
      next
        assume ?q' = 0
        with False show ?thesis ..
```

qed
qed
hence $q \neq 0$ **and** *lt q adds*$_t$ *lt p* **by** (*simp-all add*: *term-simps*)
show *?thesis* **by** (*intro disjI1*, *rule is-red-singleton-trans*, *rule* ‹*is-red* {*p*} *f*›, *fact+*)
qed
qed

lemma *replace-red-stable-is-red*:
assumes *a1*: *is-red F f* **and** *a2*: *red* (*F* − {*p*}) *p q*
shows *is-red* (*insert q* (*F* − {*p*})) *f* (**is** *is-red ?F′ f*)
proof −
from *a1* **obtain** *g* **where** *g* ∈ *F* **and** *is-red* {*g*} *f* **by** (*rule is-red-singletonI*)
show *?thesis*
proof (*cases g* = *p*)
case *True*
from *a2* **obtain** *h* **where** *h* ∈ *F* − {*p*} **and** *red* {*h*} *p q* **unfolding** *red-def* **by** *auto*
from ‹*is-red* {*g*} *f*› **have** *is-red* {*p*} *f* **unfolding** *True* **.**
have *is-red* {*q*} *f* ∨ *is-red* {*h*} *f* **by** (*rule conversion-property*, *fact+*)
thus *?thesis*
proof
assume *is-red* {*q*} *f*
show *?thesis*
proof (*rule is-red-singletonD*)
show *q* ∈ *?F′* **by** *auto*
qed *fact*
next
assume *is-red* {*h*} *f*
show *?thesis*
proof (*rule is-red-singletonD*)
from ‹*h* ∈ *F* − {*p*}› **show** *h* ∈ *?F′* **by** *simp*
qed *fact*
qed
next
case *False*
show *?thesis*
proof (*rule is-red-singletonD*)
from ‹*g* ∈ *F*› *False* **show** *g* ∈ *?F′* **by** *blast*
qed *fact*
qed
qed

lemma *is-red-map-scale*:
assumes *is-red F* (*c* · *p*)
shows *is-red F p*
proof −
from *assms* **obtain** *f u* **where** *f* ∈ *F* **and** *u* ∈ *keys* (*c* · *p*) **and** $f \neq 0$
and *a*: *lt f adds*$_t$ *u* **by** (*rule is-red-addsE*)

from *this*(*2*) *keys-map-scale-subset* **have** $u \in$ *keys p* **..**
with ‹$f \in F$› ‹$f \neq 0$› **show** *?thesis* **using** *a* **by** (*rule is-red-addsI*)
qed

corollary *is-irred-map-scale*: ¬ *is-red F p* ⟹ ¬ *is-red F* ($c \cdot p$)
by (*auto dest*: *is-red-map-scale*)

lemma *is-red-map-scale-iff*: *is-red F* ($c \cdot p$) ⟷ ($c \neq 0$ ∧ *is-red F p*)
proof (*intro iffI conjI notI*)
assume *is-red F* ($c \cdot p$) **and** $c = 0$
thus *False* **by** (*simp add*: *irred-0*)
next
assume *is-red F* ($c \cdot p$)
thus *is-red F p* **by** (*rule is-red-map-scale*)
next
assume $c \neq 0$ ∧ *is-red F p*
hence *is-red F* (*inverse* $c \cdot c \cdot p$) **by** (*simp add*: *map-scale-assoc*)
thus *is-red F* ($c \cdot p$) **by** (*rule is-red-map-scale*)
qed

lemma *is-red-uminus*: *is-red F* ($-$ *p*) ⟷ *is-red F p*
by (*auto elim*!: *is-red-addsE simp*: *keys-uminus intro*: *is-red-addsI*)

lemma *is-red-plus*:
assumes *is-red F* ($p + q$)
shows *is-red F p* ∨ *is-red F q*
proof −
from *assms* **obtain** *f u* **where** $f \in F$ **and** $u \in$ *keys* ($p + q$) **and** $f \neq 0$
and *a*: *lt f* $adds_t$ *u* **by** (*rule is-red-addsE*)
from *this*(*2*) **have** $u \in$ *keys p* ∪ *keys q*
by (*meson Poly-Mapping.keys-add subsetD*)
thus *?thesis*
proof
assume $u \in$ *keys p*
with ‹$f \in F$› ‹$f \neq 0$› **have** *is-red F p* **using** *a* **by** (*rule is-red-addsI*)
thus *?thesis* **..**
next
assume $u \in$ *keys q*
with ‹$f \in F$› ‹$f \neq 0$› **have** *is-red F q* **using** *a* **by** (*rule is-red-addsI*)
thus *?thesis* **..**
qed
qed

lemma *is-irred-plus*: ¬ *is-red F p* ⟹ ¬ *is-red F q* ⟹ ¬ *is-red F* ($p + q$)
by (*auto dest*: *is-red-plus*)

lemma *is-red-minus*:
assumes *is-red F* ($p - q$)
shows *is-red F p* ∨ *is-red F q*

proof −
  from *assms* **have** *is-red F* (*p* + (− *q*)) **by** *simp*
  hence *is-red F p* ∨ *is-red F* (− *q*) **by** (*rule is-red-plus*)
  thus *?thesis* **by** (*simp only*: *is-red-uminus*)
qed

lemma *is-irred-minus*: ¬ *is-red F p* ⟹ ¬ *is-red F q* ⟹ ¬ *is-red F* (*p* − *q*)
  by (*auto dest*: *is-red-minus*)

end

4.6 Well-foundedness and Termination

context *gd-term*
begin

lemma *dgrad-set-le-red-single*:
  assumes *dickson-grading d* **and** *red-single p q f t*
  shows *dgrad-set-le d* {*t*} (*pp-of-term* ‘ *keys p*)
proof (*rule dgrad-set-leI*, *simp*)
  have *t adds t* + *lp f* **by** *simp*
  with *assms*(*1*) **have** *d t* ≤ *d* (*pp-of-term* (*t* ⊕ *lt f*))
    by (*simp add*: *term-simps*, *rule dickson-grading-adds-imp-le*)
  moreover from *assms*(*2*) **have** *t* ⊕ *lt f* ∈ *keys p* **by** (*simp add*: *in-keys-iff red-single-def*)
  ultimately show ∃ *v*∈*keys p*. *d t* ≤ *d* (*pp-of-term v*) **..**
qed

lemma *dgrad-p-set-le-red-single*:
  assumes *dickson-grading d* **and** *red-single p q f t*
  shows *dgrad-p-set-le d* {*q*} {*f*, *p*}
proof −
  let *?f* = *monom-mult* ((*lookup p* (*t* ⊕ *lt f*)) / *lc f*) *t f*
  from *assms*(*2*) **have** *t* ⊕ *lt f* ∈ *keys p* **and** *q*: *q* = *p* − *?f* **by** (*simp-all add*: *red-single-def in-keys-iff*)
  have *dgrad-p-set-le d* {*q*} {*p*, *?f*} **unfolding** *q* **by** (*fact dgrad-p-set-le-minus*)
  also have *dgrad-p-set-le d* ... {*f*, *p*}
  proof (*rule dgrad-p-set-leI-insert*)
    from *assms*(*1*) **have** *dgrad-set-le d* (*pp-of-term* ‘ *keys ?f*) (*insert t* (*pp-of-term* ‘ *keys f*))
      by (*rule dgrad-set-le-monom-mult*)
    also have *dgrad-set-le d* ... (*pp-of-term* ‘ (*keys f* ∪ *keys p*))
    proof (*rule dgrad-set-leI*, *simp*)
      fix *s*
      assume *s* = *t* ∨ *s* ∈ *pp-of-term* ‘ *keys f*
      thus ∃ *u*∈*keys f* ∪ *keys p*. *d s* ≤ *d* (*pp-of-term u*)
      proof
        assume *s* = *t*
        from *assms* **have** *dgrad-set-le d* {*s*} (*pp-of-term* ‘ *keys p*) **unfolding** ‹*s* =

t›
by (*rule dgrad-set-le-red-single*)
moreover have *s* ∈ {*s*} **..**
ultimately obtain *s0* **where** *s0* ∈ *pp-of-term* ' *keys p* **and** *d s* ≤ *d s0* **by** (*rule dgrad-set-leE*)
from *this*(*1*) **obtain** *u* **where** *u* ∈ *keys p* **and** *s0* = *pp-of-term u* **..**
from *this*(*1*) **have** *u* ∈ *keys f* ∪ *keys p* **by** *simp*
with ‹*d s* ≤ *d s0*› **show** *?thesis* **unfolding** ‹*s0* = *pp-of-term u*› **..**
next
assume *s* ∈ *pp-of-term* ' *keys f*
hence *s* ∈ *pp-of-term* ' (*keys f* ∪ *keys p*) **by** *blast*
then obtain *u* **where** *u* ∈ *keys f* ∪ *keys p* **and** *s* = *pp-of-term u* **..**
note *this*(*1*)
moreover have *d s* ≤ *d s* **..**
ultimately show *?thesis* **unfolding** ‹*s* = *pp-of-term u*› **..**
qed
qed
finally show *dgrad-p-set-le d* {*?f*} {*f*, *p*} **by** (*simp add*: *dgrad-p-set-le-def Keys-insert*)
next
show *dgrad-p-set-le d* {*p*} {*f*, *p*} **by** (*rule dgrad-p-set-le-subset*, *simp*)
qed
finally show *?thesis* **.**
qed

lemma *dgrad-p-set-le-red*:
assumes *dickson-grading d* **and** *red F p q*
shows *dgrad-p-set-le d* {*q*} (*insert p F*)
proof −
from *assms*(*2*) **obtain** *f t* **where** *f* ∈ *F* **and** *red-single p q f t* **by** (*rule red-setE*)
from *assms*(*1*) *this*(*2*) **have** *dgrad-p-set-le d* {*q*} {*f*, *p*} **by** (*rule dgrad-p-set-le-red-single*)
also have *dgrad-p-set-le d* ... (*insert p F*) **by** (*rule dgrad-p-set-le-subset*, *auto intro*: ‹*f* ∈ *F*›)
finally show *?thesis* **.**
qed

corollary *dgrad-p-set-le-red-rtrancl*:
assumes *dickson-grading d* **and** (*red F*)** *p q*
shows *dgrad-p-set-le d* {*q*} (*insert p F*)
using *assms*(*2*)
proof (*induct*)
case *base*
show *?case* **by** (*rule dgrad-p-set-le-subset*, *simp*)
next
case (*step y z*)
from *assms*(*1*) *step*(*2*) **have** *dgrad-p-set-le d* {*z*} (*insert y F*) **by** (*rule dgrad-p-set-le-red*)
also have *dgrad-p-set-le d* ... (*insert p F*)
proof (*rule dgrad-p-set-leI-insert*)
show *dgrad-p-set-le d F* (*insert p F*) **by** (*rule dgrad-p-set-le-subset*, *blast*)

qed *fact*
finally show *?case* **.**
qed

lemma *dgrad-p-set-red-single-pp*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-p-set d m* **and** *red-single p q f t*
shows $d\ t \le m$
proof −
from *assms*(*1*) *assms*(*3*) **have** *dgrad-set-le d* {*t*} (*pp-of-term* ‘ *keys p*) **by** (*rule dgrad-set-le-red-single*)
moreover have $t \in \{t\}$ **..**
ultimately obtain *s* **where** $s \in$ *pp-of-term* ‘ *keys p* **and** $d\ t \le d\ s$ **by** (*rule dgrad-set-leE*)
from *this*(*1*) **obtain** *u* **where** $u \in$ *keys p* **and** *s* = *pp-of-term u* **..**
from *assms*(*2*) *this*(*1*) **have** *d* (*pp-of-term u*) $\le m$ **by** (*rule dgrad-p-setD*)
with ‹$d\ t \le d\ s$› **show** *?thesis* **unfolding** ‹*s* = *pp-of-term u*› **by** (*rule le-trans*)
qed

lemma *dgrad-p-set-closed-red-single*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-p-set d m* **and** $f \in$ *dgrad-p-set d m*
and *red-single p q f t*
shows $q \in$ *dgrad-p-set d m*
proof −
from *dgrad-p-set-le-red-single*[*OF assms*(*1*, *4*)] **have** $\{q\} \subseteq$ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
from *assms*(*2*, *3*) **show** $\{f, p\} \subseteq$ *dgrad-p-set d m* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

lemma *dgrad-p-set-closed-red*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m* **and** $p \in$ *dgrad-p-set d m*
and *red F p q*
shows $q \in$ *dgrad-p-set d m*
proof −
from *assms*(*4*) **obtain** *f t* **where** $f \in F$ **and** ∗: *red-single p q f t* **by** (*rule red-setE*)
from *assms*(*2*) *this*(*1*) **have** $f \in$ *dgrad-p-set d m* **..**
from *assms*(*1*) *assms*(*3*) *this* ∗ **show** *?thesis* **by** (*rule dgrad-p-set-closed-red-single*)
qed

lemma *dgrad-p-set-closed-red-rtrancl*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m* **and** $p \in$ *dgrad-p-set d m*
and (*red F*)** *p q*
shows $q \in$ *dgrad-p-set d m*
using *assms*(*4*)
proof (*induct*)
case *base*
from *assms*(*3*) **show** *?case* **.**

next
  case (*step r q*)
  from *assms*(*1*) *assms*(*2*) *step*(*3*) *step*(*2*) **show** $q \in$ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-red*)
qed

lemma *red-rtrancl-repE*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m* **and** *finite G* **and** $p \in$ *dgrad-p-set d m*
    and (*red G*)** *p r*
  obtains *q* **where** $p = r + (\sum g{\in}G.\ q\ g \odot g)$ **and** $\bigwedge g.\ q\ g \in$ *punit.dgrad-p-set d m*
    and $\bigwedge g.$ *lt* $(q\ g \odot g) \preceq_t$ *lt p*
  using *assms*(*5*)
proof (*induct r arbitrary*: *thesis*)
  case *base*
  show *?case*
  proof (*rule base*)
    show $p = p + (\sum g{\in}G.\ 0 \odot g)$ **by** *simp*
  qed (*simp-all add*: *punit.zero-in-dgrad-p-set min-term-min*)
next
  case (*step r′ r*)
  from *step.hyps*(*2*) **obtain** *g t* **where** $g \in G$ **and** *rs*: *red-single r′ r g t* **by** (*rule red-setE*)
  from *this*(*2*) **have** $r' = r +$ *monomial* (*lookup r′* $(t \oplus$ *lt g*) / *lc g*) $t \odot g$
    by (*simp add*: *red-single-def mult-scalar-monomial*)
  moreover define *q0* **where** *q0* = *monomial* (*lookup r′* $(t \oplus$ *lt g*) / *lc g*) *t*
  ultimately have *r′*: $r' = r + q0 \odot g$ **by** *simp*
  obtain *q′* **where** *p*: $p = r' + (\sum g{\in}G.\ q'\ g \odot g)$ **and** *1*: $\bigwedge g.\ q'\ g \in$ *punit.dgrad-p-set d m*
    and *2*: $\bigwedge g.$ *lt* $(q'\ g \odot g) \preceq_t$ *lt p* **by** (*rule step.hyps*) *blast*
  define *q* **where** $q = q'(g := q0 + q'\ g)$
  show *?case*
  proof (*rule step.prems*)
    from *assms*(*3*) ‹$g \in G$› **have** $p = (r + q0 \odot g) + (q'\ g \odot g + (\sum g{\in}G - \{g\}.\ q'\ g \odot g))$
      by (*simp add*: *p r′ sum.remove*)
    also have $\ldots = r + (q\ g \odot g + (\sum g{\in}G - \{g\}.\ q'\ g \odot g))$
      by (*simp add*: *q-def mult-scalar-distrib-right*)
    also from *refl* **have** $(\sum g{\in}G - \{g\}.\ q'\ g \odot g) = (\sum g{\in}G - \{g\}.\ q\ g \odot g)$
      by (*rule sum.cong*) (*simp add*: *q-def*)
    finally show $p = r + (\sum g{\in}G.\ q\ g \odot g)$ **using** *assms*(*3*) ‹$g \in G$› **by** (*simp only*: *sum.remove*)
  next
    fix *g0*
    have $q\ g0 \in$ *punit.dgrad-p-set d m* $\wedge$ *lt* $(q\ g0 \odot g0) \preceq_t$ *lt p*
    proof (*cases g0 = g*)
      case *True*
      have *eq*: $q\ g = q0 + q'\ g$ **by** (*simp add*: *q-def*)

show *?thesis* **unfolding** *True eq*
proof
from *assms*(*1*, *2*, *4*) *step.hyps*(*1*) **have** $r' \in$ *dgrad-p-set d m*
by (*rule dgrad-p-set-closed-red-rtrancl*)
with *assms*(*1*) **have** $d\ t \leq m$ **using** *rs* **by** (*rule dgrad-p-set-red-single-pp*)
hence $q0 \in$ *punit.dgrad-p-set d m* **by** (*simp add*: *q0-def punit.dgrad-p-set-def dgrad-set-def*)
thus $q0 + q'\ g \in$ *punit.dgrad-p-set d m* **by** (*intro punit.dgrad-p-set-closed-plus 1*)
next
have *lt* $(q0 \odot g + q'\ g \odot g) \preceq_t$ *ord-term-lin.max* (*lt* $(q0 \odot g)$) (*lt* $(q'\ g \odot g)$)
by (*fact lt-plus-le-max*)
also have $\ldots \preceq_t$ *lt p*
proof (*intro ord-term-lin.max.boundedI 2*)
have *lt* $(q0 \odot g) \preceq_t t \oplus$ *lt g* **by** (*simp add*: *q0-def mult-scalar-monomial lt-monom-mult-le*)
also from *rs* **have** $\ldots \preceq_t$ *lt* r' **by** (*intro lt-max*) (*simp add*: *red-single-def*)
also from *step.hyps*(*1*) **have** $\ldots \preceq_t$ *lt p* **by** (*intro ord-p-lt red-rtrancl-ord*)
finally show *lt* $(q0 \odot g) \preceq_t$ *lt p* **.**
qed
finally show *lt* $((q0 + q'\ g) \odot g) \preceq_t$ *lt p* **by** (*simp only*: *mult-scalar-distrib-right*)
qed
next
case *False*
hence $q\ g0 = q'\ g0$ **by** (*simp add*: *q-def*)
thus *?thesis* **by** (*simp add*: *1 2*)
qed
thus $q\ g0 \in$ *punit.dgrad-p-set d m* **and** *lt* $(q\ g0 \odot g0) \preceq_t$ *lt p* **by** *simp-all*
qed
qed

lemma *is-relation-order-red*:
assumes *dickson-grading d*
shows *Confluence.relation-order* (*red F*) $(\prec_p)$ (*dgrad-p-set d m*)
proof
show *wfp-on* $(\prec_p)$ (*dgrad-p-set d m*)
proof (*rule wfp-onI-min*)
fix $x::'t \Rightarrow_0 'c$ **and** Q
assume $x \in Q$ **and** $Q \subseteq$ *dgrad-p-set d m*
with *assms* **obtain** q **where** $q \in Q$ **and** $*$: $\bigwedge y.\ y \prec_p q \Longrightarrow y \notin Q$
by (*rule ord-p-minimum-dgrad-p-set*, *auto*)
from *this*(*1*) **show** $\exists z \in Q.\ \forall y \in$ *dgrad-p-set d m*. $y \prec_p z \longrightarrow y \notin Q$
proof
from $*$ **show** $\forall y \in$ *dgrad-p-set d m*. $y \prec_p q \longrightarrow y \notin Q$ **by** *auto*
qed
qed
next
show *red F* $\leq (\prec_p)^{-1-1}$ **by** (*simp add*: *predicate2I red-ord*)

qed (*fact ord-strict-p-transitive*)

lemma *red-wf-dgrad-p-set-aux*:
  assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m*
  shows *wfp-on* (*red F*)$^{-1-1}$ (*dgrad-p-set d m*)
proof (*rule wfp-onI-min*)
  fix x::$'t \Rightarrow_0 'b$ **and** Q
  assume $x \in Q$ **and** $Q \subseteq$ *dgrad-p-set d m*
  with *assms*(*1*) **obtain** q **where** $q \in Q$ **and** $*$: $\bigwedge y.\ y \prec_p q \Longrightarrow y \notin Q$
    by (*rule ord-p-minimum-dgrad-p-set*, *auto*)
  from *this*(*1*) **show** $\exists z \in Q.\ \forall y \in$*dgrad-p-set d m*. (*red F*)$^{-1-1}$ $y\ z \longrightarrow y \notin Q$
  proof
    show $\forall y \in$*dgrad-p-set d m*. (*red F*)$^{-1-1}$ $y\ q \longrightarrow y \notin Q$
    proof (*intro ballI impI*, *simp*)
      fix y
      assume *red F q y*
      hence $y \prec_p q$ **by** (*rule red-ord*)
      thus $y \notin Q$ **by** (*rule* $*$)
    qed
  qed
qed

lemma *red-wf-dgrad-p-set*:
  assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m*
  shows *wfP* (*red F*)$^{-1-1}$
proof (*rule wfI-min*[*to-pred*])
  fix x::$'t \Rightarrow_0 'b$ **and** Q
  assume $x \in Q$
  from *assms*(*2*) **obtain** n **where** $m \le n$ **and** $x \in$ *dgrad-p-set d n* **and** $F \subseteq$ *dgrad-p-set d n*
    by (*rule dgrad-p-set-insert*)
  let *?Q* = $Q \cap$ *dgrad-p-set d n*
  from *assms*(*1*) ‹$F \subseteq$ *dgrad-p-set d n*› **have** *wfp-on* (*red F*)$^{-1-1}$ (*dgrad-p-set d n*)
    by (*rule red-wf-dgrad-p-set-aux*)
  moreover from ‹$x \in Q$› ‹$x \in$ *dgrad-p-set d n*› **have** $x \in$ *?Q* **..**
  moreover have *?Q* $\subseteq$ *dgrad-p-set d n* **by** *simp*
  ultimately obtain z **where** $z \in$ *?Q* **and** $*$: $\bigwedge y.$ (*red F*)$^{-1-1}$ $y\ z \Longrightarrow y \notin$ *?Q* **by** (*rule wfp-onE-min*) *blast*
  from *this*(*1*) **have** $z \in Q$ **and** $z \in$ *dgrad-p-set d n* **by** *simp-all*
  from *this*(*1*) **show** $\exists z \in Q.\ \forall y.$ (*red F*)$^{-1-1}$ $y\ z \longrightarrow y \notin Q$
  proof
    show $\forall y.$ (*red F*)$^{-1-1}$ $y\ z \longrightarrow y \notin Q$
    proof (*intro allI impI*)
      fix y
      assume (*red F*)$^{-1-1}$ $y\ z$
      hence *red F z y* **by** *simp*
      with *assms*(*1*) ‹$F \subseteq$ *dgrad-p-set d n*› ‹$z \in$ *dgrad-p-set d n*› **have** $y \in$ *dgrad-p-set d n*

    by (*rule dgrad-p-set-closed-red*)
   moreover from ‹(*red F*)$^{-1-1}$ *y z*› **have** $y \notin$ *?Q* **by** (*rule* ∗)
   ultimately show $y \notin Q$ **by** *blast*
  qed
 qed
qed

lemmas *red-wf-finite* = *red-wf-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemma *cbelow-on-monom-mult*:
 assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m* **and** $d\ t \leq m$ **and** $c \neq 0$
  and *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) *z* ($\lambda a\ b.$ *red F a b* ∨ *red F b a*) *p q*
 shows *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) (*monom-mult c t z*) ($\lambda a\ b.$ *red F a b* ∨ *red F b a*)
     (*monom-mult c t p*) (*monom-mult c t q*)
 using *assms*(*5*)
proof (*induct rule*: *cbelow-on-induct*)
 case *base*
 show *?case* **unfolding** *cbelow-on-def*
 proof (*rule disjI1*, *intro conjI*, *fact refl*)
  from *assms*(*5*) **have** $p \in$ *dgrad-p-set d m* **by** (*rule cbelow-on-first-in*)
  with *assms*(*1*) *assms*(*3*) **show** *monom-mult c t p* ∈ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-monom-mult*)
 next
  from *assms*(*5*) **have** $p \prec_p z$ **by** (*rule cbelow-on-first-below*)
  from *this assms*(*4*) **show** *monom-mult c t p* $\prec_p$ *monom-mult c t z* **by** (*rule ord-strict-p-monom-mult*)
 qed
next
 case (*step q′ q*)
 let *?R* = $\lambda a\ b.$ *red F a b* ∨ *red F b a*
 from *step*(*5*) **show** *?case*
 proof
  from *assms*(*1*) *assms*(*3*) *step*(*3*) **show** *monom-mult c t q* ∈ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-monom-mult*)
 next
  from *step*(*2*) *red-monom-mult*[*OF - assms*(*4*)] **show** *?R* (*monom-mult c t q′*) (*monom-mult c t q*) **by** *auto*
 next
  from *step*(*4*) *assms*(*4*) **show** *monom-mult c t q* $\prec_p$ *monom-mult c t z* **by** (*rule ord-strict-p-monom-mult*)
 qed
qed

lemma *cbelow-on-monom-mult-monomial*:
 assumes $c \neq 0$
  and *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) (*monomial c′ v*) ($\lambda a\ b.$ *red F a b* ∨ *red F b a*) *p q*
 shows *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) (*monomial c* ($t \oplus v$)) ($\lambda a\ b.$ *red F a b*

$\vee$ *red F b a*) *p q*
proof −
have ∗: $f \prec_p$ *monomial* c' $v \Longrightarrow f \prec_p$ *monomial c* $(t \oplus v)$ **for** *f*
proof (*simp add*: *ord-strict-p-monomial-iff assms*(*1*), *elim conjE disjE*, *erule disjI1*, *rule disjI2*)
assume *lt f* $\prec_t$ *v*
also have ... $\preceq_t$ $t \oplus v$ **using** *local.zero-min* **using** *splus-mono-left splus-zero* **by** *fastforce*
finally show *lt f* $\prec_t$ $t \oplus v$ **.**
qed
from *assms*(*2*) **show** *?thesis*
proof (*induct rule*: *cbelow-on-induct*)
case *base*
show *?case* **unfolding** *cbelow-on-def*
proof (*rule disjI1*, *intro conjI*, *fact refl*)
from *assms*(*2*) **show** $p \in$ *dgrad-p-set d m* **by** (*rule cbelow-on-first-in*)
next
from *assms*(*2*) **have** $p \prec_p$ *monomial* c' *v* **by** (*rule cbelow-on-first-below*)
thus $p \prec_p$ *monomial c* $(t \oplus v)$ **by** (*rule* ∗)
qed
next
case (*step* q' *q*)
let *?R* = λ*a b*. *red F a b* $\vee$ *red F b a*
from *step*(*5*) *step*(*3*) *step*(*2*) **show** *?case*
proof
from *step*(*4*) **show** $q \prec_p$ *monomial c* $(t \oplus v)$ **by** (*rule* ∗)
qed
qed
qed

lemma *cbelow-on-plus*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m* **and** $r \in$ *dgrad-p-set d m*
and *keys r* $\cap$ *keys z* = {}
and *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) *z* (λ*a b*. *red F a b* $\vee$ *red F b a*) *p q*
shows *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) $(z + r)$ (λ*a b*. *red F a b* $\vee$ *red F b a*) $(p + r)$ $(q + r)$
using *assms*(*5*)
proof (*induct rule*: *cbelow-on-induct*)
case *base*
show *?case* **unfolding** *cbelow-on-def*
proof (*rule disjI1*, *intro conjI*, *fact refl*)
from *assms*(*5*) **have** $p \in$ *dgrad-p-set d m* **by** (*rule cbelow-on-first-in*)
from *this assms*(*3*) **show** $p + r \in$ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-plus*)
next
from *assms*(*5*) **have** $p \prec_p z$ **by** (*rule cbelow-on-first-below*)
from *this assms*(*4*) **show** $p + r \prec_p z + r$ **by** (*rule ord-strict-p-plus*)
qed
next
case (*step* q' *q*)

```
  let ?RS = λa b. red F a b ∨ red F b a
  let ?A = dgrad-p-set d m
  let ?R = red F
  let ?ord = (≺p)
  from assms(1) have ro: relation-order ?R ?ord ?A
    by (rule is-relation-order-red)
  have dw: relation.dw-closed ?R ?A
     by (rule relation.dw-closedI, rule dgrad-p-set-closed-red, rule assms(1), rule
assms(2))
  from step(2) have relation.cs (red F) (q' + r) (q + r)
  proof
    assume red F q q'
    hence relation.cs (red F) (q + r) (q' + r) by (rule red-plus-cs)
    thus ?thesis by (rule relation.cs-sym)
  next
    assume red F q' q
    thus ?thesis by (rule red-plus-cs)
  qed
  with ro dw have cbelow-on ?A ?ord (z + r) ?RS (q' + r) (q + r)
  proof (rule relation-order.cs-implies-cbelow-on)
    from step(1) have q' ∈ ?A by (rule cbelow-on-second-in)
    from this assms(3) show q' + r ∈ ?A by (rule dgrad-p-set-closed-plus)
  next
    from step(3) assms(3) show q + r ∈ ?A by (rule dgrad-p-set-closed-plus)
  next
    from step(1) have q' ≺p z by (rule cbelow-on-second-below)
    from this assms(4) show q' + r ≺p z + r by (rule ord-strict-p-plus)
  next
    from step(4) assms(4) show q + r ≺p z + r by (rule ord-strict-p-plus)
  qed
  with step(5) show ?case by (rule cbelow-on-transitive)
qed

lemma is-full-pmdlI-lt-dgrad-p-set:
  assumes dickson-grading d and B ⊆ dgrad-p-set d m
  assumes ⋀k. k ∈ component-of-term ` Keys (B::('t ⇒0 'b::field) set) ⟹
          (∃ b∈B. b ≠ 0 ∧ component-of-term (lt b) = k ∧ lp b = 0)
  shows is-full-pmdl B
proof (rule is-full-pmdlI)
  fix p::'t ⇒0 'b
  from assms(1, 2) have wfP (red B)^-1^-1 by (rule red-wf-dgrad-p-set)
  moreover assume component-of-term ` keys p ⊆ component-of-term ` Keys B
  ultimately show p ∈ pmdl B
  proof (induct p)
    case (less p)
    show ?case
    proof (cases p = 0)
      case True
      show ?thesis by (simp add: True pmdl.span-zero)
```

next
case *False*
hence *lt p* ∈ *keys p* **by** (*rule lt-in-keys*)
hence *component-of-term* (*lt p*) ∈ *component-of-term* ' *keys p* **by** *simp*
also have ... ⊆ *component-of-term* ' *Keys B* **by** *fact*
finally have ∃ *b*∈*B*. *b* ≠ *0* ∧ *component-of-term* (*lt b*) = *component-of-term* (*lt p*) ∧ *lp b* = *0*
by (*rule assms*(*3*))
then obtain *b* **where** *b* ∈ *B* **and** *b* ≠ *0* **and** *component-of-term* (*lt b*) = *component-of-term* (*lt p*)
and *lp b* = *0* **by** *blast*
from *this*(*3*, *4*) **have** *eq*: *lp p* ⊕ *lt b* = *lt p* **by** (*simp add*: *splus-def term-of-pair-pair*)
define *q* **where** *q* = *p* − *monom-mult* (*lookup p* ((*lp p*) ⊕ *lt b*) / *lc b*) (*lp p*) *b*
have *red-single p q b* (*lp p*)
by (*auto simp*: *red-single-def* ‹*b* ≠ *0*› *q-def eq* ‹*lt p* ∈ *keys p*›)
with ‹*b* ∈ *B*› **have** *red B p q* **by** (*rule red-setI*)
hence (*red B*)$^{-1-1}$ *q p* **..**
moreover have *component-of-term* ' *keys q* ⊆ *component-of-term* ' *Keys B*
proof (*rule subset-trans*)
from ‹*red B p q*› **show** *component-of-term* ' *keys q* ⊆ *component-of-term* ' *keys p* ∪ *component-of-term* ' *Keys B*
by (*rule components-red-subset*)
next
from *less*(*2*) **show** *component-of-term* ' *keys p* ∪ *component-of-term* ' *Keys B* ⊆ *component-of-term* ' *Keys B*
by *blast*
qed
ultimately have *q* ∈ *pmdl B* **by** (*rule less.hyps*)
have *q* + *monom-mult* (*lookup p* ((*lp p*) ⊕ *lt b*) / *lc b*) (*lp p*) *b* ∈ *pmdl B*
by (*rule pmdl.span-add*, *fact*, *rule pmdl-closed-monom-mult*, *rule pmdl.span-base*, *fact*)
thus *?thesis* **by** (*simp add*: *q-def*)
qed
qed
qed

lemmas *is-full-pmdlI-lt-finite* = *is-full-pmdlI-lt-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

end

4.7 Algorithms

4.7.1 Function *find-adds*

context *ordered-term*
begin

```
primrec find-adds :: ('t ⇒₀ 'b) list ⇒ 't ⇒ ('t ⇒₀ 'b::zero) option where
  find-adds [] - = None|
  find-adds (f # fs) u = (if f ≠ 0 ∧ lt f adds_t u then Some f else find-adds fs u)

lemma find-adds-SomeD1:
  assumes find-adds fs u = Some f
  shows f ∈ set fs
  using assms by (induct fs, simp, simp split: if-splits)

lemma find-adds-SomeD2:
  assumes find-adds fs u = Some f
  shows f ≠ 0
  using assms by (induct fs, simp, simp split: if-splits)

lemma find-adds-SomeD3:
  assumes find-adds fs u = Some f
  shows lt f adds_t u
  using assms by (induct fs, simp, simp split: if-splits)

lemma find-adds-NoneE:
  assumes find-adds fs u = None and f ∈ set fs
  assumes f = 0 ⟹ thesis and f ≠ 0 ⟹ ¬ lt f adds_t u ⟹ thesis
  shows thesis
  using assms
proof (induct fs arbitrary: thesis)
  case Nil
  from Nil(2) show ?case by simp
next
  case (Cons a fs)
  from Cons(2) have 1: a = 0 ∨ ¬ lt a adds_t u and 2: find-adds fs u = None
    by (simp-all split: if-splits)
  from Cons(3) have f = a ∨ f ∈ set fs by simp
  thus ?case
  proof
    assume f = a
    show ?thesis
    proof (cases a = 0)
      case True
      show ?thesis by (rule Cons(4), simp add: ‹f = a› True)
    next
      case False
      with 1 have *: ¬ lt a adds_t u by simp
      show ?thesis by (rule Cons(5), simp-all add: ‹f = a› * False)
    qed
  next
    assume f ∈ set fs
    with 2 show ?thesis
    proof (rule Cons(1))
      assume f = 0
```

thus *?thesis* **by** (*rule Cons*(*4*))
next
assume $f \neq 0$ **and** ¬ *lt f* $adds_t$ *u*
thus *?thesis* **by** (*rule Cons*(*5*))
qed
qed
qed

lemma *find-adds-SomeD-red-single*:
assumes $p \neq 0$ **and** *find-adds fs* (*lt p*) = *Some f*
shows *red-single p* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *f* (*lp p* − *lp f*)
proof −
let *?f* = *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) *f*
from *assms*(*2*) **have** $f \neq 0$ **and** *lt f* $adds_t$ *lt p* **by** (*rule find-adds-SomeD2*, *rule find-adds-SomeD3*)
from *this*(*2*) **have** *eq*: (*lp p* − *lp f*) ⊕ *lt f* = *lt p*
by (*simp add*: *adds-minus-splus adds-term-def term-of-pair-pair*)
from *assms*(*1*) **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
moreover from ‹*f* ≠ *0*› **have** *lc f* ≠ *0* **by** (*rule lc-not-0*)
ultimately have *lc p* / *lc f* ≠ *0* **by** *simp*
hence *lt ?f* = (*lp p* − *lp f*) ⊕ *lt f* **by** (*simp add*: *lt-monom-mult* ‹*f* ≠ *0*›)
hence *lt-f*: *lt ?f* = *lt p* **by** (*simp only*: *eq*)
have *lookup ?f* (*lt p*) = *lookup ?f* ((*lp p* − *lp f*) ⊕ *lt f*) **by** (*simp only*: *eq*)
also have ... = (*lc p* / *lc f*) ∗ *lookup f* (*lt f*) **by** (*rule lookup-monom-mult-plus*)
also from ‹*lc f* ≠ *0*› **have** ... = *lookup p* (*lt p*) **by** (*simp add*: *lc-def*)
finally have *lc-f*: *lookup ?f* (*lt p*) = *lookup p* (*lt p*) **.**
have *red-single p* (*p* − *?f*) *f* (*lp p* − *lp f*)
by (*auto simp*: *red-single-def eq lc-def* ‹*f* ≠ *0*› *lt-in-keys assms*(*1*))
moreover have *p* − *?f* = *tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)
by (*rule poly-mapping-eqI*,
simp add: *tail-monom-mult*[*symmetric*] *lookup-minus lookup-tail-2 lt-f lc-f split*: *if-split*)
ultimately show *?thesis* **by** *simp*
qed

lemma *find-adds-SomeD-red*:
assumes $p \neq 0$ **and** *find-adds fs* (*lt p*) = *Some f*
shows *red* (*set fs*) *p* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*))
proof (*rule red-setI*)
from *assms*(*2*) **show** *f* ∈ *set fs* **by** (*rule find-adds-SomeD1*)
next
from *assms* **show** *red-single p* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *f* (*lp p* − *lp f*)
by (*rule find-adds-SomeD-red-single*)
qed

end

4.7.2 Function *trd*

context *gd-term*
begin

definition *trd-term* :: $('a \Rightarrow nat) \Rightarrow ((('t \Rightarrow_0 'b::field)\ list \times ('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) \times (('t \Rightarrow_0 'b)\ list \times ('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)))\ set$
  where $trd\text{-}term\ d = \{(x, y).\ dgrad\text{-}p\text{-}set\text{-}le\ d\ (set\ (fst\ (snd\ x)\ \#\ fst\ x))\ (set\ (fst\ (snd\ y)\ \#\ fst\ y)) \wedge fst\ (snd\ x) \prec_p fst\ (snd\ y)\}$

lemma *trd-term-wf*:
  assumes *dickson-grading d*
  shows *wf* (*trd-term d*)
proof (*rule wfI-min*)
  fix $x :: ('t \Rightarrow_0 'b::field)\ list \times ('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)$ **and** Q
  assume $x \in Q$
  let $?A = set\ (fst\ (snd\ x)\ \#\ fst\ x)$
  have *finite ?A* **..**
  then obtain m **where** A: $?A \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **by** (*rule dgrad-p-set-exhaust*)
  let $?B = dgrad\text{-}p\text{-}set\ d\ m$
  let $?Q = \{q \in Q.\ set\ (fst\ (snd\ q)\ \#\ fst\ q) \subseteq ?B\}$
  note *assms*
  moreover have $fst\ (snd\ x) \in fst\ `\ snd\ `\ ?Q$
    by (*rule*, *fact refl*, *rule*, *fact refl*, *simp only*: *mem-Collect-eq A* ‹$x \in Q$›)
  moreover have $fst\ `\ snd\ `\ ?Q \subseteq ?B$ **by** *auto*
  ultimately obtain *z0* **where** $z0 \in fst\ `\ snd\ `\ ?Q$
   and $*$: $\bigwedge y.\ y \prec_p z0 \Longrightarrow y \notin fst\ `\ snd\ `\ ?Q$ **by** (*rule ord-p-minimum-dgrad-p-set*, *blast*)
  from *this*(*1*) **obtain** z **where** $z \in \{q \in Q.\ set\ (fst\ (snd\ q)\ \#\ fst\ q) \subseteq ?B\}$ **and** *z0*: $z0 = fst\ (snd\ z)$
    by *fastforce*
  from *this*(*1*) **have** $z \in Q$ **and** a: $set\ (fst\ (snd\ z)\ \#\ fst\ z) \subseteq ?B$ **by** *simp-all*
  from *this*(*1*) **show** $\exists z \in Q.\ \forall y.\ (y, z) \in trd\text{-}term\ d \longrightarrow y \notin Q$
  proof
    show $\forall y.\ (y, z) \in trd\text{-}term\ d \longrightarrow y \notin Q$
    proof (*intro allI impI*)
      fix y
      assume $(y, z) \in trd\text{-}term\ d$
      hence b: $dgrad\text{-}p\text{-}set\text{-}le\ d\ (set\ (fst\ (snd\ y)\ \#\ fst\ y))\ (set\ (fst\ (snd\ z)\ \#\ fst\ z))$ **and** $fst\ (snd\ y) \prec_p z0$
        by (*simp-all add*: *trd-term-def z0*)
      from *this*(*2*) **have** $fst\ (snd\ y) \notin fst\ `\ snd\ `\ ?Q$ **by** (*rule* $*$)
      hence $y \notin Q \vee \neg\ set\ (fst\ (snd\ y)\ \#\ fst\ y) \subseteq ?B$ **by** *auto*
      moreover from *b a* **have** $set\ (fst\ (snd\ y)\ \#\ fst\ y) \subseteq ?B$ **by** (*rule dgrad-p-set-le-dgrad-p-set*)
      ultimately show $y \notin Q$ **by** *simp*
    qed
  qed
qed

```
function trd-aux :: ('t ⇒0 'b) list ⇒ ('t ⇒0 'b) ⇒ ('t ⇒0 'b) ⇒ ('t ⇒0 'b::field)
where
  trd-aux fs p r =
    (if p = 0 then
      r
    else
      case find-adds fs (lt p) of
        None   ⇒ trd-aux fs (tail p) (r + monomial (lc p) (lt p))
      | Some f ⇒ trd-aux fs (tail p − monom-mult (lc p / lc f) (lp p − lp f) (tail
f)) r
    )
  by auto
termination proof −
  from ex-dgrad obtain d::'a ⇒ nat where dg: dickson-grading d ..
  let ?R = trd-term d
  show ?thesis
  proof (rule, rule trd-term-wf, fact)
    fix fs and p r::'t ⇒0 'b
    assume p ≠ 0
    show ((fs, tail p, r + monomial (lc p) (lt p)), fs, p, r) ∈ trd-term d
    proof (simp add: trd-term-def, rule)
      show dgrad-p-set-le d (insert (tail p) (set fs)) (insert p (set fs))
       proof (rule dgrad-p-set-leI-insert-keys, rule dgrad-p-set-le-subset, rule sub-
set-insertI,
            rule dgrad-set-le-subset, simp add: Keys-insert image-Un)
        have keys (tail p) ⊆ keys p by (auto simp: keys-tail)
        hence pp-of-term ‘ keys (tail p) ⊆ pp-of-term ‘ keys p by (rule image-mono)
         thus pp-of-term ‘ keys (tail p) ⊆ pp-of-term ‘ keys p ∪ pp-of-term ‘ Keys
(set fs) by blast
      qed
    next
      from ‹p ≠ 0› show tail p ≺p p by (rule tail-ord-p)
    qed
  next
    fix fs::('t ⇒0 'b) list and p r f ::'t ⇒0 'b
    assume p ≠ 0 and find-adds fs (lt p) = Some f
    hence red (set fs) p (tail p − monom-mult (lc p / lc f) (lp p − lp f) (tail f))
      (is red - p ?q) by (rule find-adds-SomeD-red)
    show ((fs, ?q, r), fs, p, r) ∈ trd-term d
     by (simp add: trd-term-def, rule, rule dgrad-p-set-leI-insert, rule dgrad-p-set-le-subset,
rule subset-insertI,
           rule dgrad-p-set-le-red, fact dg, fact ‹red (set fs) p ?q›, rule red-ord, fact)
  qed
qed

definition trd :: ('t ⇒0 'b::field) list ⇒ ('t ⇒0 'b) ⇒ ('t ⇒0 'b)
  where trd fs p = trd-aux fs p 0

lemma trd-aux-red-rtrancl: (red (set fs))** p (trd-aux fs p r − r)
```

proof (*induct fs p r rule*: *trd-aux.induct*)
case (*1 fs p r*)
show *?case*
proof (*simp*, *split option.split*, *intro conjI impI allI*)
assume $p \neq 0$ **and** *find-adds fs* (*lt p*) = *None*
hence (*red* (*set fs*))** (*tail p*) (*trd-aux fs* (*tail p*) (*r* + *monomial* (*lc p*) (*lt p*)) − (*r* + *monomial* (*lc p*) (*lt p*)))
by (*rule 1*(*1*))
hence (*red* (*set fs*))** (*tail p* + *monomial* (*lc p*) (*lt p*))
(*trd-aux fs* (*tail p*) (*r* + *monomial* (*lc p*) (*lt p*)) − (*r* + *monomial* (*lc p*) (*lt p*)) + *monomial* (*lc p*) (*lt p*))
proof (*rule red-rtrancl-plus-higher*)
fix *u v*
assume $u \in$ *keys* (*tail p*)
assume $v \in$ *keys* (*monomial* (*lc p*) (*lt p*))
also have ... ⊆ {*lt p*} **by** (*simp add*: *keys-monomial*)
finally have *v* = *lt p* **by** *simp*
from ‹$u \in$ *keys* (*tail p*)› **show** $u \prec_t v$ **unfolding** ‹*v* = *lt p*› **by** (*rule keys-tail-less-lt*)
qed
thus (*red* (*set fs*))** *p* (*trd-aux fs* (*tail p*) (*r* + *monomial* (*lc p*) (*lt p*)) − *r*)
by (*simp only*: *leading-monomial-tail*[*symmetric*] *add.commute*[*of - monomial* (*lc p*) (*lt p*)], *simp*)
next
fix *f*
assume $p \neq 0$ **and** *find-adds fs* (*lt p*) = *Some f*
hence (*red* (*set fs*))** (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*))
(*trd-aux fs* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *r* − *r*)
and ∗: *red* (*set fs*) *p* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*))
by (*rule 1*(*2*), *rule find-adds-SomeD-red*)
let *?q* = *tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)
from ∗ **have** (*red* (*set fs*))** *p ?q* **..**
moreover have (*red* (*set fs*))** *?q* (*trd-aux fs ?q r* − *r*) **by** *fact*
ultimately show (*red* (*set fs*))** *p* (*trd-aux fs ?q r* − *r*) **by** (*rule rtranclp-trans*)
qed
qed

corollary *trd-red-rtrancl*: (*red* (*set fs*))** *p* (*trd fs p*)
proof −
have (*red* (*set fs*))** *p* (*trd fs p* − *0*) **unfolding** *trd-def* **by** (*rule trd-aux-red-rtrancl*)
thus *?thesis* **by** *simp*
qed

lemma *trd-aux-irred*:
assumes ¬ *is-red* (*set fs*) *r*
shows ¬ *is-red* (*set fs*) (*trd-aux fs p r*)
using *assms*
proof (*induct fs p r rule*: *trd-aux.induct*)

case (*1 fs p r*)
show *?case*
proof (*simp add*: *1*(*3*), *split option.split*, *intro impI conjI allI*)
assume $p \neq 0$ **and** ∗: *find-adds fs* (*lt p*) = *None*
thus ¬ *is-red* (*set fs*) (*trd-aux fs* (*tail p*) (*r* + *monomial* (*lc p*) (*lt p*)))
proof (*rule 1*(*1*))
show ¬ *is-red* (*set fs*) (*r* + *monomial* (*lc p*) (*lt p*))
proof
assume *is-red* (*set fs*) (*r* + *monomial* (*lc p*) (*lt p*))
then obtain *f u* **where** *f* ∈ *set fs* **and** $f \neq 0$ **and** *u* ∈ *keys* (*r* + *monomial* (*lc p*) (*lt p*))
and *lt f* $adds_t$ *u* **by** (*rule is-red-addsE*)
note *this*(*3*)
also have *keys* (*r* + *monomial* (*lc p*) (*lt p*)) ⊆ *keys r* ∪ *keys* (*monomial* (*lc p*) (*lt p*))
by (*rule Poly-Mapping.keys-add*)
also have ... ⊆ *insert* (*lt p*) (*keys r*) **by** *auto*
finally show *False*
proof
assume *u* = *lt p*
from ∗ ‹*f* ∈ *set fs*› **show** *?thesis*
proof (*rule find-adds-NoneE*)
assume *f* = *0*
with ‹$f \neq 0$› **show** *?thesis* **..**
next
assume ¬ *lt f* $adds_t$ *lt p*
from *this* ‹*lt f* $adds_t$ *u*› **show** *?thesis* **unfolding** ‹*u* = *lt p*› **..**
qed
next
assume *u* ∈ *keys r*
from ‹*f* ∈ *set fs*› ‹$f \neq 0$› *this* ‹*lt f* $adds_t$ *u*› **have** *is-red* (*set fs*) *r* **by** (*rule is-red-addsI*)
with *1*(*3*) **show** *?thesis* **..**
qed
qed
qed
next
fix *f*
assume $p \neq 0$ **and** *find-adds fs* (*lt p*) = *Some f*
from *this 1*(*3*) **show** ¬ *is-red* (*set fs*) (*trd-aux fs* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *r*)
by (*rule 1*(*2*))
qed
qed

corollary *trd-irred*: ¬ *is-red* (*set fs*) (*trd fs p*)
unfolding *trd-def* **using** *irred-0* **by** (*rule trd-aux-irred*)

lemma *trd-in-pmdl*: *p* − (*trd fs p*) ∈ *pmdl* (*set fs*)

using *trd-red-rtrancl* **by** (*rule red-rtranclp-diff-in-pmdl*)

lemma *pmdl-closed-trd*:
assumes $p \in$ *pmdl B* **and** *set fs* $\subseteq$ *pmdl B*
shows (*trd fs p*) $\in$ *pmdl B*
proof −
from *assms*(*2*) **have** *pmdl* (*set fs*) $\subseteq$ *pmdl B* **by** (*rule pmdl.span-subset-spanI*)
with *trd-in-pmdl* **have** $p -$ *trd fs p* $\in$ *pmdl B* **..**
with *assms*(*1*) **have** $p - (p -$ *trd fs p*) $\in$ *pmdl B* **by** (*rule pmdl.span-diff*)
thus *?thesis* **by** *simp*
qed

end

end

5 Gröbner Bases and Buchberger's Theorem

theory *Groebner-Bases*
imports *Reduction*
begin

This theory provides the main results about Gröbner bases for modules of multivariate polynomials.

context *gd-term*
begin

definition *crit-pair* :: $('t \Rightarrow_0 'b{::}field) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow (('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b))$
where *crit-pair p q* =
(*if component-of-term* (*lt p*) = *component-of-term* (*lt q*) *then*
(*monom-mult* (*1* / *lc p*) ((*lcs* (*lp p*) (*lp q*)) − (*lp p*)) (*tail p*),
monom-mult (*1* / *lc q*) ((*lcs* (*lp p*) (*lp q*)) − (*lp q*)) (*tail q*))
else (*0*, *0*))

definition *crit-pair-cbelow-on* :: $('a \Rightarrow nat) \Rightarrow nat \Rightarrow ('t \Rightarrow_0 'b{::}field)\ set \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
where *crit-pair-cbelow-on d m F p q* $\longleftrightarrow$
cbelow-on (*dgrad-p-set d m*) ($\prec_p$)
(*monomial 1* (*term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*))))
($\lambda a\ b.$ *red F a b* $\vee$ *red F b a*) (*fst* (*crit-pair p q*)) (*snd* (*crit-pair p q*))

definition *spoly* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b{::}field)$
where *spoly p q* = (*let v1* = *lt p*; *v2* = *lt q in*
if component-of-term v1 = *component-of-term v2 then*
let t1 = *pp-of-term v1*; *t2* = *pp-of-term v2*; *l* = *lcs t1 t2 in*
(*monom-mult* (*1* / *lookup p v1*) (*l* − *t1*) *p*) − (*monom-mult* (*1* / *lookup q v2*) (*l* − *t2*) *q*)

else 0)

definition (**in** *ordered-term*) *is-Groebner-basis* :: (*'t* $\Rightarrow_0$ *'b::field*) *set* ⇒ *bool*
where *is-Groebner-basis F* ≡ *relation.is-ChurchRosser* (*red F*)

5.1 Critical Pairs and S-Polynomials

lemma *crit-pair-same*: *fst* (*crit-pair p p*) = *snd* (*crit-pair p p*)
by (*simp add*: *crit-pair-def*)

lemma *crit-pair-swap*: *crit-pair p q* = (*snd* (*crit-pair q p*), *fst* (*crit-pair q p*))
by (*simp add*: *crit-pair-def lcs-comm*)

lemma *crit-pair-zero* [*simp*]: *fst* (*crit-pair 0 q*) = *0* **and** *snd* (*crit-pair p 0*) = *0*
by (*simp-all add*: *crit-pair-def*)

lemma *dgrad-p-set-le-crit-pair-zero*: *dgrad-p-set-le d* {*fst* (*crit-pair p 0*)} {*p*}
proof (*simp add*: *crit-pair-def lt-def*[*of 0*] *lcs-comm lcs-zero dgrad-p-set-le-def Keys-insert min-term-def term-simps*, *intro conjI impI dgrad-set-leI*)
fix *s*
assume *s* ∈ *pp-of-term* ‘ *keys* (*monom-mult* (*1* / *lc p*) *0* (*tail p*))
then obtain *v* **where** *v* ∈ *keys* (*monom-mult* (*1* / *lc p*) *0* (*tail p*)) **and** *s* = *pp-of-term v* **..**
from *this*(*1*) *keys-monom-mult-subset* **have** *v* ∈ (⊕) *0* ‘ *keys* (*tail p*) **..**
hence *v* ∈ *keys* (*tail p*) **by** (*simp add*: *image-iff term-simps*)
hence *v* ∈ *keys p* **by** (*simp add*: *keys-tail*)
hence *s* ∈ *pp-of-term* ‘ *keys p* **by** (*simp add*: ‹*s* = *pp-of-term v*›)
moreover have *d s* ≤ *d s* **..**
ultimately show ∃ *t*∈*pp-of-term* ‘ *keys p*. *d s* ≤ *d t* **..**
qed *simp*

lemma *dgrad-p-set-le-fst-crit-pair*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* {*fst* (*crit-pair p q*)} {*p*, *q*}
proof (*cases q* = *0*)
case *True*
have *dgrad-p-set-le d* {*fst* (*crit-pair p q*)} {*p*} **unfolding** *True*
by (*fact dgrad-p-set-le-crit-pair-zero*)
also have *dgrad-p-set-le d* ... {*p*, *q*} **by** (*rule dgrad-p-set-le-subset*, *simp*)
finally show *?thesis* **.**
next
case *False*
show *?thesis*
proof (*cases p* = *0*)
case *True*
have *dgrad-p-set-le d* {*fst* (*crit-pair p q*)} {*q*}
by (*simp add*: *True dgrad-p-set-le-def dgrad-set-le-def*)
also have *dgrad-p-set-le d* ... {*p*, *q*} **by** (*rule dgrad-p-set-le-subset*, *simp*)
finally show *?thesis* **.**

```
  next
    case False
    show ?thesis
    proof (simp add: dgrad-p-set-le-def Keys-insert crit-pair-def, intro conjI impI)
      define t where t = lcs (lp p) (lp q) − lp p
      let ?m = monom-mult (1 / lc p) t (tail p)
      from assms have dgrad-set-le d (pp-of-term ‘ keys ?m) (insert t (pp-of-term
‘ keys (tail p)))
        by (rule dgrad-set-le-monom-mult)
      also have dgrad-set-le d ... (pp-of-term ‘ (keys p ∪ keys q))
      proof (rule dgrad-set-leI, simp)
        fix s
        assume s = t ∨ s ∈ pp-of-term ‘ keys (tail p)
        thus ∃ v∈keys p ∪ keys q. d s ≤ d (pp-of-term v)
        proof
          assume s = t
          from assms have d s ≤ ord-class.max (d (lp p)) (d (lp q))
            unfolding ‹s = t› t-def by (rule dickson-grading-lcs-minus)
          hence d s ≤ d (lp p) ∨ d s ≤ d (lp q) by auto
          thus ?thesis
          proof
            from ‹p ≠ 0› have lt p ∈ keys p by (rule lt-in-keys)
            hence lt p ∈ keys p ∪ keys q by simp
            moreover assume d s ≤ d (lp p)
            ultimately show ?thesis ..
          next
            from ‹q ≠ 0› have lt q ∈ keys q by (rule lt-in-keys)
            hence lt q ∈ keys p ∪ keys q by simp
            moreover assume d s ≤ d (lp q)
            ultimately show ?thesis ..
          qed
        next
          assume s ∈ pp-of-term ‘ keys (tail p)
          hence s ∈ pp-of-term ‘ (keys p ∪ keys q) by (auto simp: keys-tail)
          then obtain v where v ∈ keys p ∪ keys q and s = pp-of-term v ..
          note this(1)
         moreover have d s ≤ d (pp-of-term v) by (simp add: ‹s = pp-of-term v›)
          ultimately show ?thesis ..
        qed
      qed
       finally show dgrad-set-le d (pp-of-term ‘ keys ?m) (pp-of-term ‘ (keys p ∪
keys q)) .
    qed (rule dgrad-set-leI, simp)
  qed
qed

lemma dgrad-p-set-le-snd-crit-pair:
  assumes dickson-grading d
  shows dgrad-p-set-le d {snd (crit-pair p q)} {p, q}
```

by (*simp add*: *crit-pair-swap*[*of p*] *insert-commute*[*of p q*], *rule dgrad-p-set-le-fst-crit-pair*, *fact*)

lemma *dgrad-p-set-closed-fst-crit-pair*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-p-set d m* **and** $q \in$ *dgrad-p-set d m*
shows *fst* (*crit-pair p q*) $\in$ *dgrad-p-set d m*
proof −
from *dgrad-p-set-le-fst-crit-pair*[*OF assms*(*1*)] **have** {*fst* (*crit-pair p q*)} $\subseteq$ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
from *assms*(*2*, *3*) **show** {*p*, *q*} $\subseteq$ *dgrad-p-set d m* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

lemma *dgrad-p-set-closed-snd-crit-pair*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-p-set d m* **and** $q \in$ *dgrad-p-set d m*
shows *snd* (*crit-pair p q*) $\in$ *dgrad-p-set d m*
by (*simp add*: *crit-pair-swap*[*of p q*], *rule dgrad-p-set-closed-fst-crit-pair*, *fact+*)

lemma *fst-crit-pair-below-lcs*:
fst (*crit-pair p q*) $\prec_p$ *monomial 1* (*term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*)))
proof (*cases tail p* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *crit-pair-def ord-strict-p-monomial-iff*)
next
case *False*
let *?t1* = *lp p*
let *?t2* = *lp q*
from *False* **have** $p \neq 0$ **by** *auto*
hence *lc p* $\neq 0$ **by** (*rule lc-not-0*)
hence *1* / *lc p* $\neq 0$ **by** *simp*
from *this False* **have** *lt* (*monom-mult* (*1* / *lc p*) (*lcs ?t1 ?t2* − *?t1*) (*tail p*)) =
(*lcs ?t1 ?t2* − *?t1*) $\oplus$ *lt* (*tail p*)
by (*rule lt-monom-mult*)
also from *lt-tail*[*OF False*] **have** ... $\prec_t$ (*lcs ?t1 ?t2* − *?t1*) $\oplus$ *lt p*
by (*rule splus-mono-strict*)
also from *adds-lcs* **have** ... = *term-of-pair* (*lcs ?t1 ?t2*, *component-of-term* (*lt p*))
by (*simp add*: *adds-lcs adds-minus splus-def*)
finally show *?thesis* **by** (*auto simp add*: *crit-pair-def ord-strict-p-monomial-iff*)
qed

lemma *snd-crit-pair-below-lcs*:
snd (*crit-pair p q*) $\prec_p$ *monomial 1* (*term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*)))
proof (*cases component-of-term* (*lt p*) = *component-of-term* (*lt q*))
case *True*

show *?thesis*
by (*simp add*: *True crit-pair-swap*[*of p*] *lcs-comm*[*of lp p*], *fact fst-crit-pair-below-lcs*)
next
case *False*
show *?thesis* **by** (*simp add*: *crit-pair-def False ord-strict-p-monomial-iff*)
qed

lemma *crit-pair-cbelow-same*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-p-set d m*
shows *crit-pair-cbelow-on d m F p p*
proof (*simp add*: *crit-pair-cbelow-on-def crit-pair-same cbelow-on-def term-simps*, *intro disjI1 conjI*)
from *assms*(*1*) *assms*(*2*) *assms*(*2*) **show** *snd* (*crit-pair p p*) $\in$ *dgrad-p-set d m*
by (*rule dgrad-p-set-closed-snd-crit-pair*)
next
from *snd-crit-pair-below-lcs*[*of p p*] **show** *snd* (*crit-pair p p*) $\prec_p$ *monomial 1* (*lt p*)
by (*simp add*: *term-simps*)
qed

lemma *crit-pair-cbelow-distinct-component*:
assumes *component-of-term* (*lt p*) $\neq$ *component-of-term* (*lt q*)
shows *crit-pair-cbelow-on d m F p q*
by (*simp add*: *crit-pair-cbelow-on-def crit-pair-def assms cbelow-on-def ord-strict-p-monomial-iff zero-in-dgrad-p-set*)

lemma *crit-pair-cbelow-sym*:
assumes *crit-pair-cbelow-on d m F p q*
shows *crit-pair-cbelow-on d m F q p*
proof (*cases component-of-term* (*lt q*) = *component-of-term* (*lt p*))
case *True*
from *assms* **show** *?thesis*
proof (*simp add*: *crit-pair-cbelow-on-def crit-pair-swap*[*of p q*] *lcs-comm True*, *elim cbelow-on-symmetric*)
show *symp* ($\lambda a\ b.$ *red F a b* $\vee$ *red F b a*) **by** (*simp add*: *symp-def*)
qed
next
case *False*
thus *?thesis* **by** (*rule crit-pair-cbelow-distinct-component*)
qed

lemma *crit-pair-cs-imp-crit-pair-cbelow-on*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m* **and** $p \in$ *dgrad-p-set d m*
and $q \in$ *dgrad-p-set d m*
and *relation.cs* (*red F*) (*fst* (*crit-pair p q*)) (*snd* (*crit-pair p q*))
shows *crit-pair-cbelow-on d m F p q*
proof −
from *assms*(*1*) **have** *relation-order* (*red F*) ($\prec_p$) (*dgrad-p-set d m*) **by** (*rule is-relation-order-red*)

moreover have *relation.dw-closed* (*red F*) (*dgrad-p-set d m*)
by (*rule relation.dw-closedI*, *rule dgrad-p-set-closed-red*, *rule assms*(*1*), *rule assms*(*2*))
moreover note *assms*(*5*)
moreover from *assms*(*1*) *assms*(*3*) *assms*(*4*) **have** *fst* (*crit-pair p q*) ∈ *dgrad-p-set d m*
by (*rule dgrad-p-set-closed-fst-crit-pair*)
moreover from *assms*(*1*) *assms*(*3*) *assms*(*4*) **have** *snd* (*crit-pair p q*) ∈ *dgrad-p-set d m*
by (*rule dgrad-p-set-closed-snd-crit-pair*)
moreover note *fst-crit-pair-below-lcs snd-crit-pair-below-lcs*
ultimately show *?thesis* **unfolding** *crit-pair-cbelow-on-def* **by** (*rule relation-order.cs-implies-cbelow-on*)
qed

lemma *crit-pair-cbelow-mono*:
assumes *crit-pair-cbelow-on d m F p q* **and** *F* ⊆ *G*
shows *crit-pair-cbelow-on d m G p q*
using *assms*(*1*) **unfolding** *crit-pair-cbelow-on-def*
proof (*induct rule*: *cbelow-on-induct*)
case *base*
show *?case* **by** (*simp add*: *cbelow-on-def*, *intro disjI1 conjI*, *fact*+)
next
case (*step b c*)
from *step*(*2*) **have** *red G b c* ∨ *red G c b* **using** *red-subset*[*OF - assms*(*2*)] **by** *blast*
from *step*(*5*) *step*(*3*) *this step*(*4*) **show** *?case* **..**
qed

lemma *lcs-red-single-fst-crit-pair*:
assumes *p* ≠ *0* **and** *component-of-term* (*lt p*) = *component-of-term* (*lt q*)
defines *t1* ≡ *lp p*
defines *t2* ≡ *lp q*
shows *red-single* (*monomial* (− *1*) (*term-of-pair* (*lcs t1 t2*, *component-of-term* (*lt p*))))
(*fst* (*crit-pair p q*)) *p* (*lcs t1 t2* − *t1*)
proof −
let *?l* = *term-of-pair* (*lcs t1 t2*, *component-of-term* (*lt p*))
from *assms*(*1*) **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
have *lt p* $adds_t$ *?l* **by** (*simp add*: *adds-lcs adds-term-def t1-def term-simps*)
hence *eq1*: (*lcs t1 t2* − *t1*) ⊕ *lt p* = *?l*
by (*simp add*: *adds-lcs adds-minus splus-def t1-def*)
with *assms*(*1*) **show** *?thesis*
proof (*simp add*: *crit-pair-def red-single-def assms*(*2*))
have *eq2*: *monomial* (− *1*) *?l* = *monom-mult* (− (*1* / *lc p*)) (*lcs t1 t2* − *t1*) (*monomial* (*lc p*) (*lt p*))
by (*simp add*: *monom-mult-monomial eq1* ‹*lc p* ≠ *0*›)
show *monom-mult* (*1* / *lc p*) (*lcs* (*lp p*) (*lp q*) − *lp p*) (*tail p*) =
monomial (− *1*) (*term-of-pair* (*lcs t1 t2*, *component-of-term* (*lt q*))) − *monom-mult* (− (*1* / *lc p*)) (*lcs t1 t2* − *t1*) *p*

apply (*simp add*: *t1-def t2-def monom-mult-dist-right-minus tail-alt-2 monom-mult-uminus-left*)
by (*metis assms*(*2*) *eq2 monom-mult-uminus-left t1-def t2-def*)
qed
qed

corollary *lcs-red-single-snd-crit-pair*:
assumes $q \neq 0$ **and** *component-of-term* (*lt p*) = *component-of-term* (*lt q*)
defines *t1* ≡ *lp p*
defines *t2* ≡ *lp q*
shows *red-single* (*monomial* (− *1*) (*term-of-pair* (*lcs t1 t2*, *component-of-term* (*lt p*))))
(*snd* (*crit-pair p q*)) *q* (*lcs t1 t2* − *t2*)
by (*simp add*: *crit-pair-swap*[*of p q*] *lcs-comm*[*of lp p*] *assms*(*2*) *t1-def t2-def*,
rule lcs-red-single-fst-crit-pair, *simp-all add*: *assms*(*1*, *2*))

lemma *GB-imp-crit-pair-cbelow-dgrad-p-set*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-p-set d m* **and** *is-Groebner-basis F*
assumes *p* ∈ *F* **and** *q* ∈ *F* **and** *p* ≠ *0* **and** *q* ≠ *0*
shows *crit-pair-cbelow-on d m F p q*
proof (*cases component-of-term* (*lt p*) = *component-of-term* (*lt q*))
case *True*
from *assms*(*1*, *2*) **show** *?thesis*
proof (*rule crit-pair-cs-imp-crit-pair-cbelow-on*)
from *assms*(*4*, *2*) **show** *p* ∈ *dgrad-p-set d m* **..**
next
from *assms*(*5*, *2*) **show** *q* ∈ *dgrad-p-set d m* **..**
next
let *?cp* = *crit-pair p q*
let *?l* = *monomial* (− *1*) (*term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*)))
from *assms*(*4*) *lcs-red-single-fst-crit-pair*[*OF assms*(*6*) *True*] **have** *red F ?l* (*fst ?cp*)
by (*rule red-setI*)
hence *1*: (*red F*)** *?l* (*fst ?cp*) **..**
from *assms*(*5*) *lcs-red-single-snd-crit-pair*[*OF assms*(*7*) *True*] **have** *red F ?l* (*snd ?cp*)
by (*rule red-setI*)
hence *2*: (*red F*)** *?l* (*snd ?cp*) **..**
from *assms*(*3*) **have** *relation.is-confluent-on* (*red F*) *UNIV*
by (*simp only*: *is-Groebner-basis-def relation.confluence-equiv-ChurchRosser*[*symmetric*]
relation.is-confluent-def)
from *this 1 2* **show** *relation.cs* (*red F*) (*fst ?cp*) (*snd ?cp*)
by (*simp add*: *relation.is-confluent-on-def*)
qed
next
case *False*
thus *?thesis* **by** (*rule crit-pair-cbelow-distinct-component*)
qed

```
lemma spoly-alt:
  assumes p ≠ 0 and q ≠ 0
  shows spoly p q = fst (crit-pair p q) − snd (crit-pair p q)
proof (cases component-of-term (lt p) = component-of-term (lt q))
  case ec: True
  show ?thesis
  proof (rule poly-mapping-eqI, simp only: lookup-minus)
    fix v
    define t1 where t1 = lp p
    define t2 where t2 = lp q
    let ?l = lcs t1 t2
    let ?lv = term-of-pair (?l, component-of-term (lt p))
    let ?cp = crit-pair p q
    let ?a = λx. monom-mult (1 / lc p) (?l − t1) x
    let ?b = λx. monom-mult (1 / lc q) (?l − t2) x
    have l-1: (?l − t1) ⊕ lt p = ?lv by (simp add: adds-lcs adds-minus splus-def
t1-def)
   have l-2: (?l − t2) ⊕ lt q = ?lv by (simp add: ec adds-lcs-2 adds-minus splus-def
t2-def)
    show lookup (spoly p q) v = lookup (fst ?cp) v − lookup (snd ?cp) v
    proof (cases v = ?lv)
      case True
      have v-1: v = (?l − t1) ⊕ lt p by (simp add: True l-1)
      from ‹p ≠ 0› have lt p ∈ keys p by (rule lt-in-keys)
      hence v-2: v = (?l − t2) ⊕ lt q by (simp add: True l-2)
      from ‹q ≠ 0› have lt q ∈ keys q by (rule lt-in-keys)
      from ‹lt p ∈ keys p› have lookup (?a p) v = 1
        by (simp add: in-keys-iff v-1 lookup-monom-mult lc-def term-simps)
      also from ‹lt q ∈ keys q› have ... = lookup (?b q) v
        by (simp add: in-keys-iff v-2 lookup-monom-mult lc-def term-simps)
      finally have lookup (spoly p q) v = 0
        by (simp add: spoly-def ec Let-def t1-def t2-def lookup-minus lc-def)
      moreover have lookup (fst ?cp) v = 0
      by (simp add: crit-pair-def ec v-1 lookup-monom-mult t1-def t2-def term-simps,
            simp only: not-in-keys-iff-lookup-eq-zero[symmetric] keys-tail, simp)
      moreover have lookup (snd ?cp) v = 0
      by (simp add: crit-pair-def ec v-2 lookup-monom-mult t1-def t2-def term-simps,
            simp only: not-in-keys-iff-lookup-eq-zero[symmetric] keys-tail, simp)
      ultimately show ?thesis by simp
    next
      case False
      have lookup (?a (tail p)) v = lookup (?a p) v
      proof (cases ?l − t1 adds_p v)
        case True
        then obtain u where v: v = (?l − t1) ⊕ u ..
        have u ≠ lt p
        proof
          assume u = lt p
          hence v = ?lv by (simp add: v l-1)
```

```
          with ‹v ≠ ?lv› show False ..
        qed
        thus ?thesis by (simp add: v lookup-monom-mult lookup-tail-2 term-simps)
      next
        case False
        thus ?thesis by (simp add: lookup-monom-mult)
      qed
      moreover have lookup (?b (tail q)) v = lookup (?b q) v
      proof (cases ?l − t2 adds_p v)
        case True
        then obtain u where v: v = (?l − t2) ⊕ u ..
        have u ≠ lt q
        proof
          assume u = lt q
          hence v = ?lv by (simp add: v l-2)
          with ‹v ≠ ?lv› show False ..
        qed
        thus ?thesis by (simp add: v lookup-monom-mult lookup-tail-2 term-simps)
      next
        case False
        thus ?thesis by (simp add: lookup-monom-mult)
      qed
      ultimately show ?thesis
         by (simp add: ec spoly-def crit-pair-def lookup-minus t1-def t2-def Let-def
lc-def)
    qed
  qed
next
  case False
  show ?thesis by (simp add: spoly-def crit-pair-def False)
qed

lemma spoly-same: spoly p p = 0
  by (simp add: spoly-def)

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

lemma spoly-red-zero-imp-crit-pair-cbelow-on:
  assumes dickson-grading d and F ⊆ dgrad-p-set d m and p ∈ dgrad-p-set d m
    and q ∈ dgrad-p-set d m and p ≠ 0 and q ≠ 0 and (red F)** (spoly p q) 0
  shows crit-pair-cbelow-on d m F p q
proof −
  from assms(7) have relation.cs (red F) (fst (crit-pair p q)) (snd (crit-pair p q))
    unfolding spoly-alt[OF assms(5) assms(6)] by (rule red-diff-rtrancl-cs)
  with assms(1) assms(2) assms(3) assms(4) show ?thesis by (rule crit-pair-cs-imp-crit-pair-cbelow-on)
qed

lemma dgrad-p-set-le-spoly-zero: dgrad-p-set-le d {spoly p 0} {p}
```

```
proof (simp add: term-simps spoly-def lt-def[of 0] lcs-comm lcs-zero dgrad-p-set-le-def
Keys-insert
      Let-def min-term-def lc-def[symmetric], intro conjI impI dgrad-set-leI)
  fix s
  assume s ∈ pp-of-term ‘ keys (monom-mult (1 / lc p) 0 p)
  then obtain u where u ∈ keys (monom-mult (1 / lc p) 0 p) and s = pp-of-term
u ..
  from this(1) keys-monom-mult-subset have u ∈ (⊕) 0 ‘ keys p ..
  hence u ∈ keys p by (simp add: image-iff term-simps)
  hence s ∈ pp-of-term ‘ keys p by (simp add: ‹s = pp-of-term u›)
  moreover have d s ≤ d s ..
  ultimately show ∃ t∈pp-of-term ‘ keys p. d s ≤ d t ..
qed simp

lemma dgrad-p-set-le-spoly:
  assumes dickson-grading d
  shows dgrad-p-set-le d {spoly p q} {p, q}
proof (cases p = 0)
  case True
  have dgrad-p-set-le d {spoly p q} {spoly q 0} unfolding True spoly-swap[of 0 q]
    by (fact dgrad-p-set-le-uminus)
  also have dgrad-p-set-le d ... {q} by (fact dgrad-p-set-le-spoly-zero)
  also have dgrad-p-set-le d ... {p, q} by (rule dgrad-p-set-le-subset, simp)
  finally show ?thesis .
next
  case False
  show ?thesis
  proof (cases q = 0)
    case True
   have dgrad-p-set-le d {spoly p q} {p} unfolding True by (fact dgrad-p-set-le-spoly-zero)
    also have dgrad-p-set-le d ... {p, q} by (rule dgrad-p-set-le-subset, simp)
    finally show ?thesis .
  next
    case False
    have dgrad-p-set-le d {spoly p q} {fst (crit-pair p q), snd (crit-pair p q)}
      unfolding spoly-alt[OF ‹p ≠ 0› False] by (rule dgrad-p-set-le-minus)
    also have dgrad-p-set-le d ... {p, q}
    proof (rule dgrad-p-set-leI-insert)
      from assms show dgrad-p-set-le d {fst (crit-pair p q)} {p, q}
        by (rule dgrad-p-set-le-fst-crit-pair)
    next
      from assms show dgrad-p-set-le d {snd (crit-pair p q)} {p, q}
        by (rule dgrad-p-set-le-snd-crit-pair)
    qed
    finally show ?thesis .
  qed
qed

lemma dgrad-p-set-closed-spoly:
```

assumes *dickson-grading d* **and** *p* ∈ *dgrad-p-set d m* **and** *q* ∈ *dgrad-p-set d m*
shows *spoly p q* ∈ *dgrad-p-set d m*
proof −
from *dgrad-p-set-le-spoly*[*OF assms*(*1*)] **have** {*spoly p q*} ⊆ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
from *assms*(*2*, *3*) **show** {*p*, *q*} ⊆ *dgrad-p-set d m* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

lemma *components-spoly-subset*: *component-of-term* ' *keys* (*spoly p q*) ⊆ *component-of-term* ' *Keys* {*p*, *q*}
unfolding *spoly-def Let-def*
proof (*split if-split*, *intro conjI impI*)
define *c* **where** *c* = (*1* / *lookup p* (*lt p*))
define *d* **where** *d* = (*1* / *lookup q* (*lt q*))
define *s* **where** *s* = *lcs* (*lp p*) (*lp q*) − *lp p*
define *t* **where** *t* = *lcs* (*lp p*) (*lp q*) − *lp q*
show *component-of-term* ' *keys* (*monom-mult c s p* − *monom-mult d t q*) ⊆ *component-of-term* ' *Keys* {*p*, *q*}
proof
fix *k*
assume *k* ∈ *component-of-term* ' *keys* (*monom-mult c s p* − *monom-mult d t q*)
then obtain *v* **where** *v* ∈ *keys* (*monom-mult c s p* − *monom-mult d t q*) **and** *k*: *k* = *component-of-term v* **..**
from *this*(*1*) *keys-minus* **have** *v* ∈ *keys* (*monom-mult c s p*) ∪ *keys* (*monom-mult d t q*) **..**
thus *k* ∈ *component-of-term* ' *Keys* {*p*, *q*}
proof
assume *v* ∈ *keys* (*monom-mult c s p*)
from *this keys-monom-mult-subset* **have** *v* ∈ (⊕) *s* ' *keys p* **..**
then obtain *u* **where** *u* ∈ *keys p* **and** *v*: *v* = *s* ⊕ *u* **..**
have *u* ∈ *Keys* {*p*, *q*} **by** (*rule in-KeysI*, *fact*, *simp*)
moreover have *k* = *component-of-term u* **by** (*simp add*: *v k term-simps*)
ultimately show *?thesis* **by** *simp*
next
assume *v* ∈ *keys* (*monom-mult d t q*)
from *this keys-monom-mult-subset* **have** *v* ∈ (⊕) *t* ' *keys q* **..**
then obtain *u* **where** *u* ∈ *keys q* **and** *v*: *v* = *t* ⊕ *u* **..**
have *u* ∈ *Keys* {*p*, *q*} **by** (*rule in-KeysI*, *fact*, *simp*)
moreover have *k* = *component-of-term u* **by** (*simp add*: *v k term-simps*)
ultimately show *?thesis* **by** *simp*
qed
qed
qed *simp*

lemma *pmdl-closed-spoly*:
assumes *p* ∈ *pmdl F* **and** *q* ∈ *pmdl F*

shows *spoly p q* ∈ *pmdl F*
proof (*cases component-of-term* (*lt p*) = *component-of-term* (*lt q*))
case *True*
show *?thesis*
by (*simp add*: *spoly-def True Let-def*, *rule pmdl.span-diff*,
(*rule pmdl-closed-monom-mult*, *fact*)+)
next
case *False*
show *?thesis* **by** (*simp add*: *spoly-def False pmdl.span-zero*)
qed

5.2 Buchberger's Theorem

Before proving the main theorem of Gröbner bases theory for S-polynomials, as is usually done in textbooks, we first prove it for critical pairs: a set F yields a confluent reduction relation if the critical pairs of all $p \in F$ and $q \in F$ can be connected below the least common sum of the leading power-products of p and q. The reason why we proceed in this way is that it becomes much easier to prove the correctness of Buchberger's second criterion for avoiding useless pairs.

lemma *crit-pair-cbelow-imp-confluent-dgrad-p-set*:
assumes *dg*: *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m*
assumes *main*: $\bigwedge p\ q.\ p \in F \Longrightarrow q \in F \Longrightarrow p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow$ *crit-pair-cbelow-on d m F p q*
shows *relation.is-confluent-on* (*red F*) (*dgrad-p-set d m*)
proof −
let *?A* = *dgrad-p-set d m*
let *?R* = *red F*
let *?RS* = λ*a b*. *red F a b* ∨ *red F b a*
let *?ord* = ($\prec_p$)
from *dg* **have** *ro*: *Confluence.relation-order ?R ?ord ?A*
by (*rule is-relation-order-red*)
have *dw*: *relation.dw-closed ?R ?A*
by (*rule relation.dw-closedI*, *rule dgrad-p-set-closed-red*, *rule dg*, *rule assms*(*2*))
show *?thesis*
proof (*rule relation-order.loc-connectivity-implies-confluence*, *fact ro*)
show *is-loc-connective-on ?A ?ord ?R* **unfolding** *is-loc-connective-on-def*
proof (*intro ballI allI impI*)
fix *a b1 b2* :: $'t \Rightarrow_0 'b$
assume *a* ∈ *?A*
assume *?R a b1* ∧ *?R a b2*
hence *?R a b1* **and** *?R a b2* **by** *simp-all*
hence *b1* ∈ *?A* **and** *b2* ∈ *?A* **and** *?ord b1 a* **and** *?ord b2 a*
using *red-ord dgrad-p-set-closed-red*[*OF dg assms*(*2*) ‹*a* ∈ *?A*›] **by** *blast*+
from *this*(*1*) *this*(*2*) **have** *b1* − *b2* ∈ *?A* **by** (*rule dgrad-p-set-closed-minus*)
from ‹*red F a b1*› **obtain** *f1* **and** *t1* **where** *f1* ∈ *F* **and** *r1*: *red-single a b1 f1 t1* **by** (*rule red-setE*)
from ‹*red F a b2*› **obtain** *f2* **and** *t2* **where** *f2* ∈ *F* **and** *r2*: *red-single a b2*

f2 t2 **by** (*rule red-setE*)

from *r1 r2* **have** *f1* ≠ *0* **and** *f2* ≠ *0* **by** (*simp-all add*: *red-single-def*)
hence *lc1*: *lc f1* ≠ *0* **and** *lc2*: *lc f2* ≠ *0* **using** *lc-not-0* **by** *auto*
show *cbelow-on ?A ?ord a* (λ*a b*. *?R a b* ∨ *?R b a*) *b1 b2*
proof (*cases t1* ⊕ *lt f1* = *t2* ⊕ *lt f2*)
case *False*
from *confluent-distinct*[*OF r1 r2 False* ‹*f1* ∈ *F*› ‹*f2* ∈ *F*›] **obtain** *s*
where *s1*: (*red F*)** *b1 s* **and** *s2*: (*red F*)** *b2 s* **.**
have *relation.cs ?R b1 b2* **unfolding** *relation.cs-def* **by** (*intro exI conjI*, *fact s1*, *fact s2*)
from *ro dw this* ‹*b1* ∈ *?A*› ‹*b2* ∈ *?A*› ‹*?ord b1 a*› ‹*?ord b2 a*› **show** *?thesis*
by (*rule relation-order.cs-implies-cbelow-on*)
next
case *True*
hence *ec*: *component-of-term* (*lt f1*) = *component-of-term* (*lt f2*)
by (*metis component-of-term-splus*)
let *?l1* = *lp f1*
let *?l2* = *lp f2*
define *v* **where** *v* ≡ *t2* ⊕ *lt f2*
define *l* **where** *l* ≡ *lcs ?l1 ?l2*
define *a'* **where** *a'* = *except a* {*v*}
define *ma* **where** *ma* = *monomial* (*lookup a v*) *v*
have *v-alt*: *v* = *t1* ⊕ *lt f1* **by** (*simp only*: *True v-def*)
have *a* = *ma* + *a'* **unfolding** *ma-def a'-def* **by** (*fact plus-except*)
have *comp-f1*: *component-of-term* (*lt f1*) = *component-of-term v* **by** (*simp add*: *v-alt term-simps*)

have *?l1 adds l* **unfolding** *l-def* **by** (*rule adds-lcs*)
have *?l2 adds l* **unfolding** *l-def* **by** (*rule adds-lcs-2*)
have *?l1* $adds_p$ (*t1* ⊕ *lt f1*) **by** (*simp add*: *adds-pp-splus term-simps*)
hence *?l1* $adds_p$ *v* **by** (*simp add*: *v-alt*)
have *?l2* $adds_p$ *v* **by** (*simp add*: *v-def adds-pp-splus term-simps*)
from ‹*?l1* $adds_p$ *v*› ‹*?l2* $adds_p$ *v*› **have** *l* $adds_p$ *v* **by** (*simp add*: *l-def adds-pp-def lcs-adds*)
have *pp-of-term* (*v* ⊖ *?l1*) = *t1* **by** (*simp add*: *v-alt term-simps*)
with ‹*l* $adds_p$ *v*› ‹*?l1 adds l*› **have** *tf1'*: *pp-of-term* ((*l* − *?l1*) ⊕ (*v* ⊖ *l*)) = *t1*
by (*simp add*: *minus-splus-sminus-cancel*)
hence *tf1*: ((*pp-of-term v*) − *l*) + (*l* − *?l1*) = *t1* **by** (*simp add*: *add.commute term-simps*)
have *pp-of-term* (*v* ⊖ *?l2*) = *t2* **by** (*simp add*: *v-def term-simps*)
with ‹*l* $adds_p$ *v*› ‹*?l2 adds l*› **have** *tf2'*: *pp-of-term* ((*l* − *?l2*) ⊕ (*v* ⊖ *l*)) = *t2*
by (*simp add*: *minus-splus-sminus-cancel*)
hence *tf2*: ((*pp-of-term v*) − *l*) + (*l* − *?l2*) = *t2* **by** (*simp add*: *add.commute term-simps*)
let *?ca* = *lookup a v*
let *?v* = *pp-of-term v* − *l*
have *?v* + *l* = *pp-of-term v* **using** ‹*l* $adds_p$ *v*› *adds-minus adds-pp-def* **by**

blast

from *tf1'* **have** *?v adds t1* **unfolding** *pp-of-term-splus add.commute*[*of l* − *?l1*] *pp-of-term-sminus*

using *addsI* **by** *blast*

with *dg* **have** *d ?v* ≤ *d t1* **by** (*rule dickson-grading-adds-imp-le*)

also from *dg* ‹*a* ∈ *?A*› *r1* **have** ... ≤ *m* **by** (*rule dgrad-p-set-red-single-pp*)

finally have *d ?v* ≤ *m* **.**

from *r2* **have** *?ca* ≠ *0* **by** (*simp add*: *red-single-def v-def*)

hence − *?ca* ≠ *0* **by** *simp*

from *r1* **have** *b1* = *a* − *monom-mult* (*?ca* / *lc f1*) *t1 f1* **by** (*simp add*: *red-single-def v-alt*)

also have ... = *monom-mult* (− *?ca*) *?v* (*fst* (*crit-pair f1 f2*)) + *a'*

proof (*simp add*: *a'-def ec crit-pair-def l-def*[*symmetric*] *monom-mult-assoc tf1*,

rule poly-mapping-eqI, *simp add*: *lookup-add lookup-minus*)

fix *u*

show *lookup a u* − *lookup* (*monom-mult* (*?ca* / *lc f1*) *t1 f1*) *u* =

lookup (*monom-mult* (− (*?ca* / *lc f1*)) *t1* (*tail f1*)) *u* + *lookup* (*except a* {*v*}) *u*

proof (*cases u* = *v*)

case *True*

show *?thesis*

by (*simp add*: *True lookup-except v-alt lookup-monom-mult lookup-tail-2 lc-def*[*symmetric*] *lc1 term-simps*)

next

case *False*

hence *u* ∉ {*v*} **by** *simp*

moreover

{

assume *t1* $adds_p$ *u*

hence *t1* ⊕ (*u* ⊖ *t1*) = *u* **by** (*simp add*: *adds-pp-sminus*)

hence *u* ⊖ *t1* ≠ *lt f1* **using** *False v-alt* **by** *auto*

hence *lookup f1* (*u* ⊖ *t1*) = *lookup* (*tail f1*) (*u* ⊖ *t1*) **by** (*simp add*: *lookup-tail-2*)

}

ultimately show *?thesis* **using** *False* **by** (*simp add*: *lookup-except lookup-monom-mult*)

qed

qed

finally have *b1*: *b1* = *monom-mult* (− *?ca*) *?v* (*fst* (*crit-pair f1 f2*)) + *a'* **.**

from *r2* **have** *b2* = *a* − *monom-mult* (*?ca* / *lc f2*) *t2 f2*

by (*simp add*: *red-single-def v-def True*)

also have ... = *monom-mult* (− *?ca*) *?v* (*snd* (*crit-pair f1 f2*)) + *a'*

proof (*simp add*: *a'-def ec crit-pair-def l-def*[*symmetric*] *monom-mult-assoc tf2*,

rule poly-mapping-eqI, *simp add*: *lookup-add lookup-minus*)
fix *u*
show *lookup a u* − *lookup* (*monom-mult* (*?ca* / *lc f2*) *t2 f2*) *u* =
lookup (*monom-mult* (− (*?ca* / *lc f2*)) *t2* (*tail f2*)) *u* + *lookup* (*except a* {*v*}) *u*
proof (*cases u* = *v*)
case *True*
show *?thesis*
by (*simp add*: *True lookup-except v-def lookup-monom-mult lookup-tail-2 lc-def*[*symmetric*] *lc2 term-simps*)
next
case *False*
hence *u* ∉ {*v*} **by** *simp*
moreover
{
assume *t2* $adds_p$ *u*
hence *t2* ⊕ (*u* ⊖ *t2*) = *u* **by** (*simp add*: *adds-pp-sminus*)
hence *u* ⊖ *t2* ≠ *lt f2* **using** *False v-def* **by** *auto*
hence *lookup f2* (*u* ⊖ *t2*) = *lookup* (*tail f2*) (*u* ⊖ *t2*) **by** (*simp add*: *lookup-tail-2*)
}
ultimately show *?thesis* **using** *False* **by** (*simp add*: *lookup-except lookup-monom-mult*)
qed
qed
finally have *b2*: *b2* = *monom-mult* (− *?ca*) *?v* (*snd* (*crit-pair f1 f2*)) + *a′*
.

let *?lv* = *term-of-pair* (*l*, *component-of-term* (*lt f1*))
from ‹*f1* ∈ *F*› ‹*f2* ∈ *F*› ‹*f1* ≠ *0*› ‹*f2* ≠ *0*› **have** *crit-pair-cbelow-on d m F f1 f2* **by** (*rule main*)
hence *cbelow-on ?A ?ord* (*monomial 1 ?lv*) *?RS* (*fst* (*crit-pair f1 f2*)) (*snd* (*crit-pair f1 f2*))
by (*simp only*: *crit-pair-cbelow-on-def l-def*)
with *dg assms* (*2*) ‹*d ?v* ≤ *m*› ‹− *?ca* ≠ *0*›
have *cbelow-on ?A ?ord* (*monom-mult* (− *?ca*) *?v* (*monomial 1 ?lv*)) *?RS*
(*monom-mult* (− *?ca*) *?v* (*fst* (*crit-pair f1 f2*)))
(*monom-mult* (− *?ca*) *?v* (*snd* (*crit-pair f1 f2*)))
by (*rule cbelow-on-monom-mult*)
hence *cbelow-on ?A ?ord* (*monomial* (− *?ca*) *v*) *?RS*
(*monom-mult* (− *?ca*) *?v* (*fst* (*crit-pair f1 f2*)))
(*monom-mult* (− *?ca*) *?v* (*snd* (*crit-pair f1 f2*)))
by (*simp add*: *monom-mult-monomial* ‹(*pp-of-term v* − *l*) + *l* = *pp-of-term v*› *splus-def comp-f1 term-simps*)
with ‹*?ca* ≠ *0*› **have** *cbelow-on ?A ?ord* (*monomial ?ca* (*0* ⊕ *v*)) *?RS*
(*monom-mult* (−*?ca*) *?v* (*fst* (*crit-pair f1 f2*))) (*monom-mult* (−*?ca*) *?v* (*snd* (*crit-pair f1 f2*)))
by (*rule cbelow-on-monom-mult-monomial*)
hence *cbelow-on ?A ?ord ma ?RS*

(*monom-mult* (−*?ca*) *?v* (*fst* (*crit-pair f1 f2*))) (*monom-mult* (−*?ca*) *?v* (*snd* (*crit-pair f1 f2*)))
by (*simp add*: *ma-def term-simps*)
with *dg assms*(*2*) - -
show *cbelow-on ?A ?ord a ?RS b1 b2* **unfolding** ‹*a* = *ma* + *a'*› *b1 b2*
proof (*rule cbelow-on-plus*)
show *a'* ∈ *?A*
by (*rule*, *simp add*: *a'-def keys-except*, *erule conjE*, *intro dgrad-p-setD*, *rule* ‹*a* ∈ *dgrad-p-set d m*›)
next
show *keys a'* ∩ *keys ma* = {} **by** (*simp add*: *ma-def a'-def keys-except*)
qed
qed
qed
qed *fact*
qed

corollary *crit-pair-cbelow-imp-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-p-set d m*
assumes ⋀*p q*. *p* ∈ *F* ⟹ *q* ∈ *F* ⟹ *p* ≠ *0* ⟹ *q* ≠ *0* ⟹ *crit-pair-cbelow-on d m F p q*
shows *is-Groebner-basis F*
unfolding *is-Groebner-basis-def*
proof (*rule relation.confluence-implies-ChurchRosser*, *simp only*: *relation.is-confluent-def relation.is-confluent-on-def*, *intro ballI allI impI*)
fix *a b1 b2*
assume *a*: (*red F*)** *a b1* ∧ (*red F*)** *a b2*
from *assms*(*2*) **obtain** *n* **where** *m* ≤ *n* **and** *a* ∈ *dgrad-p-set d n* **and** *F* ⊆ *dgrad-p-set d n*
by (*rule dgrad-p-set-insert*)
{
fix *p q*
assume *p* ∈ *F* **and** *q* ∈ *F* **and** *p* ≠ *0* **and** *q* ≠ *0*
hence *crit-pair-cbelow-on d m F p q* **by** (*rule assms*(*3*))
from *this dgrad-p-set-subset*[*OF* ‹*m* ≤ *n*›] **have** *crit-pair-cbelow-on d n F p q*
unfolding *crit-pair-cbelow-on-def* **by** (*rule cbelow-on-mono*)
}
with *assms*(*1*) ‹*F* ⊆ *dgrad-p-set d n*› **have** *relation.is-confluent-on* (*red F*) (*dgrad-p-set d n*)
by (*rule crit-pair-cbelow-imp-confluent-dgrad-p-set*)
from *this* ‹*a* ∈ *dgrad-p-set d n*› **have** ∀ *b1 b2*. (*red F*)** *a b1* ∧ (*red F*)** *a b2* ⟶ *relation.cs* (*red F*) *b1 b2*
unfolding *relation.is-confluent-on-def* **..**
with *a* **show** *relation.cs* (*red F*) *b1 b2* **by** *blast*
qed

corollary *Buchberger-criterion-dgrad-p-set*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-p-set d m*

assumes $\bigwedge p\ q.\ p \in F \Longrightarrow q \in F \Longrightarrow p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow p \neq q \Longrightarrow$ *component-of-term* (*lt p*) = *component-of-term* (*lt q*) $\Longrightarrow$ (*red F*)** (*spoly p q*) *0*
shows *is-Groebner-basis F*
using *assms*(*1*) *assms*(*2*)
proof (*rule crit-pair-cbelow-imp-GB-dgrad-p-set*)
fix *p q*
assume $p \in F$ **and** $q \in F$ **and** $p \neq 0$ **and** $q \neq 0$
from *this*(*1*, *2*) *assms*(*2*) **have** *p*: $p \in$ *dgrad-p-set d m* **and** *q*: $q \in$ *dgrad-p-set d m* **by** *auto*
show *crit-pair-cbelow-on d m F p q*
proof (*cases p = q*)
case *True*
from *assms*(*1*) *q* **show** *?thesis* **unfolding** *True* **by** (*rule crit-pair-cbelow-same*)
next
case *False*
show *?thesis*
proof (*cases component-of-term* (*lt p*) = *component-of-term* (*lt q*))
case *True*
from *assms*(*1*) *assms*(*2*) *p q* ‹$p \neq 0$› ‹$q \neq 0$› **show** *crit-pair-cbelow-on d m F p q*
proof (*rule spoly-red-zero-imp-crit-pair-cbelow-on*)
from ‹$p \in F$› ‹$q \in F$› ‹$p \neq 0$› ‹$q \neq 0$› ‹$p \neq q$› *True* **show** (*red F*)** (*spoly p q*) *0*
by (*rule assms*(*3*))
qed
next
case *False*
thus *?thesis* **by** (*rule crit-pair-cbelow-distinct-component*)
qed
qed
qed

lemmas *Buchberger-criterion-finite* = *Buchberger-criterion-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemma (**in** *ordered-term*) *GB-imp-zero-reducibility*:
assumes *is-Groebner-basis G* **and** $f \in$ *pmdl G*
shows (*red G*)** *f 0*
proof –
from *in-pmdl-srtc*[*OF* ‹$f \in$ *pmdl G*›] ‹*is-Groebner-basis G*› **have** *relation.cs* (*red G*) *f 0*
unfolding *is-Groebner-basis-def relation.is-ChurchRosser-def* **by** *simp*
then obtain *s* **where** *rfs*: (*red G*)** *f s* **and** *r0s*: (*red G*)** *0 s* **unfolding** *relation.cs-def* **by** *auto*
from *rtrancl-0*[*OF r0s*] **and** *rfs* **show** *?thesis* **by** *simp*
qed

lemma (**in** *ordered-term*) *GB-imp-reducibility*:

assumes *is-Groebner-basis G* **and** $f \neq 0$ **and** $f \in$ *pmdl G*
shows *is-red G f*
using *assms* **by** (*meson GB-imp-zero-reducibility is-red-def relation.rtrancl-is-final*)

lemma *is-Groebner-basis-empty*: *is-Groebner-basis* {}
by (*rule Buchberger-criterion-finite*, *rule*, *simp*)

lemma *is-Groebner-basis-singleton*: *is-Groebner-basis* {*f*}
by (*rule Buchberger-criterion-finite*, *simp*, *simp add*: *spoly-same*)

5.3 Buchberger's Criteria for Avoiding Useless Pairs

Unfortunately, the product criterion is only applicable to scalar polynomials.

lemma (**in** *gd-powerprod*) *product-criterion*:
assumes *dickson-grading d* **and** $F \subseteq$ *punit.dgrad-p-set d m* **and** $p \in F$ **and** $q \in F$
and $p \neq 0$ **and** $q \neq 0$ **and** *gcs* (*punit.lt p*) (*punit.lt q*) = *0*
shows *punit.crit-pair-cbelow-on d m F p q*
proof −
let *?lt* = *punit.lt p*
let *?lq* = *punit.lt q*
let *?l* = *lcs ?lt ?lq*
define *s* **where** *s* = *punit.monom-mult* (− *1* / (*punit.lc p* ∗ *punit.lc q*)) *0* (*punit.tail p* ∗ *punit.tail q*)
from *assms*(*7*) **have** *?l* = *?lt* + *?lq* **by** (*metis add-cancel-left-left gcs-plus-lcs*)
hence *?l* − *?lt* = *?lq* **and** *?l* − *?lq* = *?lt* **by** *simp-all*

have (*punit.red* {*q*})** (*punit.tail p* ∗ (*monomial* (*1* / *punit.lc p*) (*punit.lt q*)))
(*punit.monom-mult* (− (*1* / *punit.lc p*) / *punit.lc q*) *0* (*punit.tail p* ∗ *punit.tail q*))
unfolding *punit-mult-scalar*[*symmetric*] **using** ‹$q \neq 0$› **by** (*rule punit.red-mult-scalar-lt*)
moreover have *punit.monom-mult* (*1* / *punit.lc p*) (*punit.lt q*) (*punit.tail p*) =
punit.tail p ∗ (*monomial* (*1* / *punit.lc p*) (*punit.lt q*))
by (*simp add*: *times-monomial-left*[*symmetric*])
ultimately have (*punit.red* {*q*})** (*fst* (*punit.crit-pair p q*)) *s*
by (*simp add*: *punit.crit-pair-def* ‹*?l* − *?lt* = *?lq*› *s-def*)
moreover from ‹$q \in F$› **have** {*q*} ⊆ *F* **by** *simp*
ultimately have *1*: (*punit.red F*)** (*fst* (*punit.crit-pair p q*)) *s* **by** (*rule punit.red-rtrancl-subset*)

have (*punit.red* {*p*})** (*punit.tail q* ∗ (*monomial* (*1* / *punit.lc q*) (*punit.lt p*)))
(*punit.monom-mult* (− (*1* / *punit.lc q*) / *punit.lc p*) *0* (*punit.tail q* ∗ *punit.tail p*))
unfolding *punit-mult-scalar*[*symmetric*] **using** ‹$p \neq 0$› **by** (*rule punit.red-mult-scalar-lt*)
hence (*punit.red* {*p*})** (*snd* (*punit.crit-pair p q*)) *s*
by (*simp add*: *punit.crit-pair-def* ‹*?l* − *?lq* = *?lt*› *s-def mult.commute flip*: *times-monomial-left*)
moreover from ‹$p \in F$› **have** {*p*} ⊆ *F* **by** *simp*
ultimately have *2*: (*punit.red F*)** (*snd* (*punit.crit-pair p q*)) *s* **by** (*rule punit.red-rtrancl-subset*)

note *assms*(*1*) *assms*(*2*)
moreover from ‹*p* ∈ *F*› ‹*F* ⊆ *punit.dgrad-p-set d m*› **have** *p* ∈ *punit.dgrad-p-set d m* **..**
moreover from ‹*q* ∈ *F*› ‹*F* ⊆ *punit.dgrad-p-set d m*› **have** *q* ∈ *punit.dgrad-p-set d m* **..**
moreover from *1 2* **have** *relation.cs* (*punit.red F*) (*fst* (*punit.crit-pair p q*)) (*snd* (*punit.crit-pair p q*))
unfolding *relation.cs-def* **by** *blast*
ultimately show *?thesis* **by** (*rule punit.crit-pair-cs-imp-crit-pair-cbelow-on*)
qed

lemma *chain-criterion*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-p-set d m* **and** *p* ∈ *F* **and** *q* ∈ *F*
and *p* ≠ *0* **and** *q* ≠ *0* **and** *lp r adds lcs* (*lp p*) (*lp q*)
and *component-of-term* (*lt r*) = *component-of-term* (*lt p*)
and *pr*: *crit-pair-cbelow-on d m F p r* **and** *rq*: *crit-pair-cbelow-on d m F r q*
shows *crit-pair-cbelow-on d m F p q*
proof (*cases component-of-term* (*lt p*) = *component-of-term* (*lt q*))
case *True*
with *assms*(*8*) **have** *comp-r*: *component-of-term* (*lt r*) = *component-of-term* (*lt q*) **by** *simp*
let *?A* = *dgrad-p-set d m*
let *?RS* = λ*a b*. *red F a b* ∨ *red F b a*
let *?lt* = *lp p*
let *?lq* = *lp q*
let *?lr* = *lp r*
let *?ltr* = *lcs ?lt ?lr*
let *?lrq* = *lcs ?lr ?lq*
let *?ltq* = *lcs ?lt ?lq*

from ‹*p* ∈ *F*› ‹*F* ⊆ *dgrad-p-set d m*› **have** *p* ∈ *dgrad-p-set d m* **..**
from *this* ‹*p* ≠ *0*› **have** *d ?lt* ≤ *m* **by** (*rule dgrad-p-setD-lp*)
from ‹*q* ∈ *F*› ‹*F* ⊆ *dgrad-p-set d m*› **have** *q* ∈ *dgrad-p-set d m* **..**
from *this* ‹*q* ≠ *0*› **have** *d ?lq* ≤ *m* **by** (*rule dgrad-p-setD-lp*)
from *assms*(*1*) **have** *d ?ltq* ≤ *ord-class.max* (*d ?lt*) (*d ?lq*) **by** (*rule dickson-grading-lcs*)
also from ‹*d ?lt* ≤ *m*› ‹*d ?lq* ≤ *m*› **have** ... ≤ *m* **by** *simp*
finally have *d ?ltq* ≤ *m* **.**

from *adds-lcs* ‹*?lr adds ?ltq*› **have** *?ltr adds ?ltq* **by** (*rule lcs-adds*)
then obtain *up* **where** *?ltq* = *?ltr* + *up* **..**
hence *up1*: *?ltq* − *?lt* = *up* + (*?ltr* − *?lt*) **and** *up2*: *up* + (*?ltr* − *?lr*) = *?ltq* − *?lr*
by (*metis add.commute adds-lcs minus-plus*, *metis add.commute adds-lcs-2 minus-plus*)
have *fst-pq*: *fst* (*crit-pair p q*) = *monom-mult 1 up* (*fst* (*crit-pair p r*))
by (*simp add*: *crit-pair-def monom-mult-assoc up1 True comp-r*)
from *assms*(*1*) *assms*(*2*) *- - pr*
have *cbelow-on ?A* ($\prec_p$) (*monom-mult 1 up* (*monomial 1* (*term-of-pair* (*?ltr*,

component-of-term (*lt p*))))) *?RS*
(*fst* (*crit-pair p q*)) (*monom-mult 1 up* (*snd* (*crit-pair p r*)))
unfolding *fst-pq crit-pair-cbelow-on-def*
proof (*rule cbelow-on-monom-mult*)
from ‹*d ?ltq* ≤ *m*› **show** *d up* ≤ *m* **by** (*simp add*: ‹*?ltq* = *?ltr* + *up*› *dickson-gradingD1*[*OF assms*(*1*)])
qed *simp*
hence *1*: *cbelow-on ?A* (≺*p*) (*monomial 1* (*term-of-pair* (*?ltq*, *component-of-term* (*lt p*)))) *?RS*
(*fst* (*crit-pair p q*)) (*monom-mult 1 up* (*snd* (*crit-pair p r*)))
by (*simp add*: *monom-mult-monomial* ‹*?ltq* = *?ltr* + *up*› *add.commute splus-def term-simps*)

from ‹*?lr adds ?ltq*› *adds-lcs-2* **have** *?lrq adds ?ltq* **by** (*rule lcs-adds*)
then obtain *uq* **where** *?ltq* = *?lrq* + *uq* **..**
hence *uq1*: *?ltq* − *?lq* = *uq* + (*?lrq* − *?lq*) **and** *uq2*: *uq* + (*?lrq* − *?lr*) = *?ltq* − *?lr*
by (*metis add.commute adds-lcs-2 minus-plus*, *metis add.commute adds-lcs minus-plus*)
have *eq*: *monom-mult 1 uq* (*fst* (*crit-pair r q*)) = *monom-mult 1 up* (*snd* (*crit-pair p r*))
by (*simp add*: *crit-pair-def monom-mult-assoc up2 uq2 True comp-r*)
have *snd-pq*: *snd* (*crit-pair p q*) = *monom-mult 1 uq* (*snd* (*crit-pair r q*))
by (*simp add*: *crit-pair-def monom-mult-assoc uq1 True comp-r*)
from *assms*(*1*) *assms*(*2*) - - *rq*
have *cbelow-on ?A* (≺*p*) (*monom-mult 1 uq* (*monomial 1* (*term-of-pair* (*?lrq*, *component-of-term* (*lt p*))))) *?RS*
(*monom-mult 1 uq* (*fst* (*crit-pair r q*))) (*snd* (*crit-pair p q*))
unfolding *snd-pq crit-pair-cbelow-on-def assms*(*8*)
proof (*rule cbelow-on-monom-mult*)
from ‹*d ?ltq* ≤ *m*› **show** *d uq* ≤ *m* **by** (*simp add*: ‹*?ltq* = *?lrq* + *uq*› *dickson-gradingD1*[*OF assms*(*1*)])
qed *simp*
hence *cbelow-on ?A* (≺*p*) (*monomial 1* (*term-of-pair* (*?ltq*, *component-of-term* (*lt p*)))) *?RS*
(*monom-mult 1 uq* (*fst* (*crit-pair r q*))) (*snd* (*crit-pair p q*))
by (*simp add*: *monom-mult-monomial* ‹*?ltq* = *?lrq* + *uq*› *add.commute splus-def term-simps*)
hence *cbelow-on ?A* (≺*p*) (*monomial 1* (*term-of-pair* (*?ltq*, *component-of-term* (*lt p*)))) *?RS*
(*monom-mult 1 up* (*snd* (*crit-pair p r*))) (*snd* (*crit-pair p q*))
by (*simp only*: *eq*)

with *1* **show** *?thesis* **unfolding** *crit-pair-cbelow-on-def* **by** (*rule cbelow-on-transitive*)
next
case *False*
thus *?thesis* **by** (*rule crit-pair-cbelow-distinct-component*)
qed

5.4 Weak and Strong Gröbner Bases

lemma *ord-p-wf-on*:
  assumes *dickson-grading d*
  shows *wfp-on* $(\prec_p)$ (*dgrad-p-set d m*)
proof (*rule wfp-onI-min*)
  fix $x::'t \Rightarrow_0 'b$ **and** Q
  assume $x \in Q$ **and** $Q \subseteq$ *dgrad-p-set d m*
  with *assms* **obtain** z **where** $z \in Q$ **and** $*$: $\bigwedge y.\ y \prec_p z \Longrightarrow y \notin Q$
    by (*rule ord-p-minimum-dgrad-p-set*, *blast*)
  from *this*(*1*) **show** $\exists z{\in}Q.\ \forall y{\in}$*dgrad-p-set d m*. $y \prec_p z \longrightarrow y \notin Q$
  proof
    show $\forall y{\in}$*dgrad-p-set d m*. $y \prec_p z \longrightarrow y \notin Q$ **by** (*intro ballI impI* $*$)
  qed
qed

lemma *is-red-implies-0-red-dgrad-p-set*:
  assumes *dickson-grading d* **and** $B \subseteq$ *dgrad-p-set d m*
  assumes *pmdl B* $\subseteq$ *pmdl A* **and** $\bigwedge q.\ q \in$ *pmdl A* $\Longrightarrow q \in$ *dgrad-p-set d m* $\Longrightarrow$
$q \neq 0 \Longrightarrow$ *is-red B q*
    and $p \in$ *pmdl A* **and** $p \in$ *dgrad-p-set d m*
  shows $($*red B*$)^{**}$ *p 0*
proof −
  from *ord-p-wf-on*[*OF assms*(*1*)] *assms*(*6*, *5*) **show** *?thesis*
  proof (*induction p rule*: *wfp-on-induct*)
    case (*less p*)
    show *?case*
    proof (*cases p* = *0*)
      case *True*
      thus *?thesis* **by** *simp*
    next
      case *False*
      from *assms*(*4*)[*OF less*(*3*, *1*) *False*] **obtain** *q* **where** *redpq*: *red B p q* **unfolding** *is-red-alt* **..**
      with *assms*(*1*) *assms*(*2*) *less*(*1*) **have** $q \in$ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-red*)
      moreover from *redpq* **have** $q \prec_p p$ **by** (*rule red-ord*)
      moreover from ‹*pmdl B* $\subseteq$ *pmdl A*› ‹$p \in$ *pmdl A*› ‹*red B p q*› **have** $q \in$
pmdl A
        by (*rule pmdl-closed-red*)
      ultimately have $($*red B*$)^{**}$ *q 0* **by** (*rule less*(*2*))
      show *?thesis* **by** (*rule converse-rtranclp-into-rtranclp*, *rule redpq*, *fact*)
    qed
  qed
qed

lemma *is-red-implies-0-red-dgrad-p-set'*:
  assumes *dickson-grading d* **and** $B \subseteq$ *dgrad-p-set d m*
  assumes *pmdl B* $\subseteq$ *pmdl A* **and** $\bigwedge q.\ q \in$ *pmdl A* $\Longrightarrow q \neq 0 \Longrightarrow$ *is-red B q*
    and $p \in$ *pmdl A*

shows (*red B*)** *p 0*
proof –
from *assms*(*2*) **obtain** *n* **where** $m \leq n$ **and** $p \in$ *dgrad-p-set d n* **and** *B*: $B \subseteq$ *dgrad-p-set d n*
by (*rule dgrad-p-set-insert*)
from *ord-p-wf-on*[*OF assms*(*1*)] *this*(*2*) *assms*(*5*) **show** *?thesis*
proof (*induction p rule*: *wfp-on-induct*)
case (*less p*)
show *?case*
proof (*cases p = 0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
from *assms*(*4*)[*OF* ‹$p \in$ (*pmdl A*)› *False*] **obtain** *q* **where** *redpq*: *red B p q* **unfolding** *is-red-alt* **..**
with *assms*(*1*) *B* ‹$p \in$ *dgrad-p-set d n*› **have** $q \in$ *dgrad-p-set d n* **by** (*rule dgrad-p-set-closed-red*)
moreover from *redpq* **have** $q \prec_p p$ **by** (*rule red-ord*)
moreover from ‹*pmdl B* $\subseteq$ *pmdl A*› ‹$p \in$ *pmdl A*› ‹*red B p q*› **have** $q \in$ *pmdl A*
by (*rule pmdl-closed-red*)
ultimately have (*red B*)** *q 0* **by** (*rule less*(*2*))
show *?thesis* **by** (*rule converse-rtranclp-into-rtranclp*, *rule redpq*, *fact*)
qed
qed
qed

lemma *pmdl-eqI-adds-lt-dgrad-p-set*:
fixes *G*::(*'t* $\Rightarrow_0$ *'b*::*field*) *set*
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m* **and** $B \subseteq$ *dgrad-p-set d m* **and** *pmdl G* $\subseteq$ *pmdl B*
assumes $\bigwedge f.\ f \in$ *pmdl B* $\Longrightarrow f \in$ *dgrad-p-set d m* $\Longrightarrow f \neq 0 \Longrightarrow (\exists g \in G.\ g \neq 0 \wedge$ *lt g adds*$_t$ *lt f*)
shows *pmdl G* = *pmdl B*
proof
show *pmdl B* $\subseteq$ *pmdl G*
proof (*rule pmdl.span-subset-spanI*, *rule*)
fix *p*
assume $p \in B$
hence $p \in$ *pmdl B* **and** $p \in$ *dgrad-p-set d m* **by** (*rule pmdl.span-base*, *rule*, *intro assms*(*3*))
with *assms*(*1*, *2*, *4*) - **have** (*red G*)** *p 0*
proof (*rule is-red-implies-0-red-dgrad-p-set*)
fix *f*
assume $f \in$ *pmdl B* **and** $f \in$ *dgrad-p-set d m* **and** $f \neq 0$
hence ($\exists g \in G.\ g \neq 0 \wedge$ *lt g adds*$_t$ *lt f*) **by** (*rule assms*(*5*))
then obtain *g* **where** $g \in G$ **and** $g \neq 0$ **and** *lt g adds*$_t$ *lt f* **by** *blast*
thus *is-red G f* **using** ‹$f \neq 0$› *is-red-indI1* **by** *blast*

```
    qed
    thus p ∈ pmdl G by (rule red-rtranclp-0-in-pmdl)
  qed
qed fact

lemma pmdl-eqI-adds-lt-dgrad-p-set':
  fixes G::('t ⇒_0 'b::field) set
  assumes dickson-grading d and G ⊆ dgrad-p-set d m and pmdl G ⊆ pmdl B
  assumes ⋀f. f ∈ pmdl B ⟹ f ≠ 0 ⟹ (∃ g ∈ G. g ≠ 0 ∧ lt g adds_t lt f)
  shows pmdl G = pmdl B
proof
  show pmdl B ⊆ pmdl G
  proof
    fix p
    assume p ∈ pmdl B
    with assms(1, 2, 3) - have (red G)** p 0
    proof (rule is-red-implies-0-red-dgrad-p-set')
      fix f
      assume f ∈ pmdl B and f ≠ 0
      hence (∃ g ∈ G. g ≠ 0 ∧ lt g adds_t lt f) by (rule assms(4))
      then obtain g where g ∈ G and g ≠ 0 and lt g adds_t lt f by blast
      thus is-red G f using ‹f ≠ 0› is-red-indI1 by blast
    qed
    thus p ∈ pmdl G by (rule red-rtranclp-0-in-pmdl)
  qed
qed fact

lemma GB-implies-unique-nf-dgrad-p-set:
  assumes dickson-grading d and G ⊆ dgrad-p-set d m
  assumes isGB: is-Groebner-basis G
  shows ∃! h. (red G)** f h ∧ ¬ is-red G h
proof −
  from assms(1) assms(2) have wfP (red G)^-1-1 by (rule red-wf-dgrad-p-set)
  then obtain h where ftoh: (red G)** f h and irredh: relation.is-final (red G) h
    by (rule relation.wf-imp-nf-ex)
  show ?thesis
  proof
     from ftoh and irredh show (red G)** f h ∧ ¬ is-red G h by (simp add:
is-red-def)
  next
    fix h'
    assume (red G)** f h' ∧ ¬ is-red G h'
   hence ftoh': (red G)** f h' and irredh': relation.is-final (red G) h' by (simp-all
add: is-red-def)
    show h' = h
    proof (rule relation.ChurchRosser-unique-final)
     from isGB show relation.is-ChurchRosser (red G) by (simp only: is-Groebner-basis-def)
    qed fact+
  qed
```

qed

lemma *translation-property'*:
  assumes $p \neq 0$ **and** *red-p-0*: $(red\ F)^{**}\ p\ 0$
  shows *is-red F* $(p + q) \vee$ *is-red F q*
proof (*rule disjCI*)
  assume *not-red*: ¬ *is-red F q*
  from *red-p-0* ‹$p \neq 0$› **obtain** *f* **where** $f \in F$ **and** $f \neq 0$ **and** *lt-adds*: *lt f* $adds_t$ *lt p*
    by (*rule zero-reducibility-implies-lt-divisibility*)
  show *is-red F* $(p + q)$
  proof (*cases* $q = 0$)
    case *True*
    with *is-red-indI1*[*OF* ‹$f \in F$› ‹$f \neq 0$› ‹$p \neq 0$› *lt-adds*] **show** *?thesis* **by** *simp*
  next
    case *False*
    from *not-red is-red-addsI*[*OF* ‹$f \in F$› ‹$f \neq 0$› - *lt-adds*, *of q*] **have** ¬ *lt p* ∈ (*keys q*) **by** *blast*
    hence *lookup q* (*lt p*) = *0* **by** (*simp add*: *in-keys-iff*)
    with *lt-in-keys*[*OF* ‹$p \neq 0$›] **have** *lt p* ∈ (*keys* $(p + q)$) **unfolding** *in-keys-iff* **by** (*simp add*: *lookup-add*)
    from *is-red-addsI*[*OF* ‹$f \in F$› ‹$f \neq 0$› *this lt-adds*] **show** *?thesis* **.**
  qed
qed

lemma *translation-property*:
  assumes $p \neq q$ **and** *red-0*: $(red\ F)^{**}\ (p - q)\ 0$
  shows *is-red F p* ∨ *is-red F q*
proof −
  from ‹$p \neq q$› **have** $p - q \neq 0$ **by** *simp*
  from *translation-property'*[*OF this red-0*, *of q*] **show** *?thesis* **by** *simp*
qed

lemma *weak-GB-is-strong-GB-dgrad-p-set*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
  assumes $\bigwedge f.\ f \in pmdl\ G \Longrightarrow f \in dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow (red\ G)^{**}\ f\ 0$
  shows *is-Groebner-basis G*
  using *assms*(*1*, *2*)
proof (*rule Buchberger-criterion-dgrad-p-set*)
  fix *p q*
  assume $p \in G$ **and** $q \in G$
  hence $p \in pmdl\ G$ **and** $q \in pmdl\ G$ **by** (*auto intro*: *pmdl.span-base*)
  hence *spoly p q* ∈ *pmdl G* **by** (*rule pmdl-closed-spoly*)
  thus $(red\ G)^{**}$ (*spoly p q*) *0*
  proof (*rule assms*(*3*))
    note *assms*(*1*)
    moreover from ‹$p \in G$› *assms*(*2*) **have** $p \in$ *dgrad-p-set d m* **..**
    moreover from ‹$q \in G$› *assms*(*2*) **have** $q \in$ *dgrad-p-set d m* **..**
    ultimately show *spoly p q* ∈ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-spoly*)

qed
qed

lemma *weak-GB-is-strong-GB*:
assumes $\bigwedge f.\ f \in (pmdl\ G) \Longrightarrow (red\ G)^{**}\ f\ 0$
shows *is-Groebner-basis G*
unfolding *is-Groebner-basis-def*
proof (*rule relation.confluence-implies-ChurchRosser*,
simp add: *relation.is-confluent-def relation.is-confluent-on-def*, *intro allI impI*, *erule conjE*)
fix *f p q*
assume $(red\ G)^{**}\ f\ p$ **and** $(red\ G)^{**}\ f\ q$
hence *relation.srtc* (*red G*) *p q*
by (*meson relation.rtc-implies-srtc relation.srtc-symmetric relation.srtc-transitive*)
hence $p - q \in pmdl\ G$ **by** (*rule srtc-in-pmdl*)
hence $(red\ G)^{**}\ (p - q)\ 0$ **by** (*rule assms*)
thus *relation.cs* (*red G*) *p q* **by** (*rule red-diff-rtrancl-cs*)
qed

corollary *GB-alt-1-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq dgrad\text{-}p\text{-}set\ d\ m$
shows $is\text{-}Groebner\text{-}basis\ G \longleftrightarrow (\forall f \in pmdl\ G.\ f \in dgrad\text{-}p\text{-}set\ d\ m \longrightarrow (red\ G)^{**}\ f\ 0)$
using *weak-GB-is-strong-GB-dgrad-p-set*[*OF assms*] *GB-imp-zero-reducibility* **by** *blast*

corollary *GB-alt-1*: $is\text{-}Groebner\text{-}basis\ G \longleftrightarrow (\forall f \in pmdl\ G.\ (red\ G)^{**}\ f\ 0)$
using *weak-GB-is-strong-GB GB-imp-zero-reducibility* **by** *blast*

lemma *isGB-I-is-red*:
assumes *dickson-grading d* **and** $G \subseteq dgrad\text{-}p\text{-}set\ d\ m$
assumes $\bigwedge f.\ f \in pmdl\ G \Longrightarrow f \in dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow f \neq 0 \Longrightarrow is\text{-}red\ G\ f$
shows *is-Groebner-basis G*
unfolding *GB-alt-1-dgrad-p-set*[*OF assms*(*1*, *2*)]
proof (*intro ballI impI*)
fix *f*
assume $f \in pmdl\ G$ **and** $f \in dgrad\text{-}p\text{-}set\ d\ m$
with *assms*(*1*, *2*) *subset-refl assms*(*3*) **show** $(red\ G)^{**}\ f\ 0$
by (*rule is-red-implies-0-red-dgrad-p-set*)
qed

lemma *GB-alt-2-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq dgrad\text{-}p\text{-}set\ d\ m$
shows $is\text{-}Groebner\text{-}basis\ G \longleftrightarrow (\forall f \in pmdl\ G.\ f \neq 0 \longrightarrow is\text{-}red\ G\ f)$
proof
assume *is-Groebner-basis G*
show $\forall f \in pmdl\ G.\ f \neq 0 \longrightarrow is\text{-}red\ G\ f$
proof (*intro ballI*, *intro impI*)
fix *f*

```
    assume f ∈ (pmdl G) and f ≠ 0
    show is-red G f by (rule GB-imp-reducibility, fact+)
  qed
next
  assume a2: ∀f∈pmdl G. f ≠ 0 ⟶ is-red G f
  show is-Groebner-basis G unfolding GB-alt-1
  proof
    fix f
    assume f ∈ pmdl G
    from assms show (red G)** f 0
    proof (rule is-red-implies-0-red-dgrad-p-set')
      fix q
      assume q ∈ pmdl G and q ≠ 0
      thus is-red G q by (rule a2[rule-format])
    qed (fact subset-refl, fact)
  qed
qed

lemma GB-adds-lt:
  assumes is-Groebner-basis G and f ∈ pmdl G and f ≠ 0
  obtains g where g ∈ G and g ≠ 0 and lt g adds_t lt f
proof −
  from assms(1) assms(2) have (red G)** f 0 by (rule GB-imp-zero-reducibility)
  show ?thesis by (rule zero-reducibility-implies-lt-divisibility, fact+)
qed

lemma isGB-I-adds-lt:
  assumes dickson-grading d and G ⊆ dgrad-p-set d m
  assumes ⋀f. f ∈ pmdl G ⟹ f ∈ dgrad-p-set d m ⟹ f ≠ 0 ⟹ (∃ g ∈ G. g
≠ 0 ∧ lt g adds_t lt f)
  shows is-Groebner-basis G
  using assms(1, 2)
proof (rule isGB-I-is-red)
  fix f
  assume f ∈ pmdl G and f ∈ dgrad-p-set d m and f ≠ 0
  hence (∃ g ∈ G. g ≠ 0 ∧ lt g adds_t lt f) by (rule assms(3))
  then obtain g where g ∈ G and g ≠ 0 and lt g adds_t lt f by blast
  thus is-red G f using ‹f ≠ 0› is-red-indI1 by blast
qed

lemma GB-alt-3-dgrad-p-set:
  assumes dickson-grading d and G ⊆ dgrad-p-set d m
  shows is-Groebner-basis G ⟷ (∀f ∈ pmdl G. f ≠ 0 ⟶ (∃ g ∈ G. g ≠ 0 ∧ lt
g adds_t lt f))
    (is ?L ⟷ ?R)
proof
  assume ?L
  show ?R
  proof (intro ballI impI)
```

fix *f*
assume $f \in$ *pmdl G* **and** $f \neq 0$
with ‹*?L*› **obtain** *g* **where** $g \in G$ **and** $g \neq 0$ **and** *lt g* $adds_t$ *lt f* **by** (*rule GB-adds-lt*)
thus $\exists g \in G.\ g \neq 0 \wedge$ *lt g* $adds_t$ *lt f* **by** *blast*
qed
next
assume *?R*
show *?L* **unfolding** *GB-alt-2-dgrad-p-set*[*OF assms*]
proof (*intro ballI impI*)
fix *f*
assume $f \in$ *pmdl G* **and** $f \neq 0$
with ‹*?R*› **have** ($\exists g \in G.\ g \neq 0 \wedge$ *lt g* $adds_t$ *lt f*) **by** *blast*
then obtain *g* **where** $g \in G$ **and** $g \neq 0$ **and** *lt g* $adds_t$ *lt f* **by** *blast*
thus *is-red G f* **using** ‹$f \neq 0$› *is-red-indI1* **by** *blast*
qed
qed

lemma *GB-insert*:
assumes *is-Groebner-basis G* **and** $f \in$ *pmdl G*
shows *is-Groebner-basis* (*insert f G*)
using *assms* **unfolding** *GB-alt-1*
by (*metis insert-subset pmdl.span-insert-idI red-rtrancl-subset subsetI*)

lemma *GB-subset*:
assumes *is-Groebner-basis G* **and** $G \subseteq G'$ **and** *pmdl* G' = *pmdl G*
shows *is-Groebner-basis* G'
using *assms*(*1*) **unfolding** *GB-alt-1* **using** *assms*(*2*) *assms*(*3*) *red-rtrancl-subset*
by *blast*

lemma (**in** *ordered-term*) *GB-remove-0-stable-GB*:
assumes *is-Groebner-basis G*
shows *is-Groebner-basis* ($G - \{0\}$)
using *assms* **by** (*simp only*: *is-Groebner-basis-def red-minus-singleton-zero*)

lemmas *is-red-implies-0-red-finite* = *is-red-implies-0-red-dgrad-p-set'*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-implies-unique-nf-finite* = *GB-implies-unique-nf-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-alt-2-finite* = *GB-alt-2-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-alt-3-finite* = *GB-alt-3-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *pmdl-eqI-adds-lt-finite* = *pmdl-eqI-adds-lt-dgrad-p-set'*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

5.5 Alternative Characterization of Gröbner Bases via Representations of S-Polynomials

definition *spoly-rep* :: $('a \Rightarrow nat) \Rightarrow nat \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0$ *'b::field*$) \Rightarrow$ *bool*
  where *spoly-rep d m G g1 g2* $\longleftrightarrow$ ($\exists$ *q. spoly g1 g2* = ($\sum$ *g*$\in$*G. q g* $\odot$ *g*) $\wedge$
    ($\forall$ *g. q g* $\in$ *punit.dgrad-p-set d m* $\wedge$
      (*q g* $\odot$ *g* $\neq$ *0* $\longrightarrow$ *lt* (*q g* $\odot$ *g*) $\prec_t$ *term-of-pair* (*lcs* (*lp g1*) (*lp g2*),
        component-of-term (*lt g2*)))))

lemma *spoly-repI*:
  spoly g1 g2 = ($\sum$ *g*$\in$*G. q g* $\odot$ *g*) $\Longrightarrow$ ($\bigwedge$*g. q g* $\in$ *punit.dgrad-p-set d m*) $\Longrightarrow$
    ($\bigwedge$*g. q g* $\odot$ *g* $\neq$ *0* $\Longrightarrow$ *lt* (*q g* $\odot$ *g*) $\prec_t$ *term-of-pair* (*lcs* (*lp g1*) (*lp g2*),
      component-of-term (*lt g2*))) $\Longrightarrow$
    spoly-rep d m G g1 g2
  by (*auto simp*: *spoly-rep-def*)

lemma *spoly-repI-zero*:
  assumes *spoly g1 g2* = *0*
  shows *spoly-rep d m G g1 g2*
proof (*rule spoly-repI*)
  show *spoly g1 g2* = ($\sum$ *g*$\in$*G. 0* $\odot$ *g*) **by** (*simp add*: *assms*)
qed (*simp-all add*: *punit.zero-in-dgrad-p-set*)

lemma *spoly-repE*:
  assumes *spoly-rep d m G g1 g2*
  obtains *q* **where** *spoly g1 g2* = ($\sum$ *g*$\in$*G. q g* $\odot$ *g*) **and** $\bigwedge$*g. q g* $\in$ *punit.dgrad-p-set d m*
    and $\bigwedge$*g. q g* $\odot$ *g* $\neq$ *0* $\Longrightarrow$ *lt* (*q g* $\odot$ *g*) $\prec_t$ *term-of-pair* (*lcs* (*lp g1*) (*lp g2*),
      component-of-term (*lt g2*))
  using *assms* **by** (*auto simp*: *spoly-rep-def*)

corollary *isGB-D-spoly-rep*:
  assumes *dickson-grading d* **and** *is-Groebner-basis G* **and** *G* $\subseteq$ *dgrad-p-set d m* **and** *finite G*
    and *g1* $\in$ *G* **and** *g2* $\in$ *G* **and** *g1* $\neq$ *0* **and** *g2* $\neq$ *0*
  shows *spoly-rep d m G g1 g2*
proof (*cases spoly g1 g2* = *0*)
  case *True*
  thus *?thesis* **by** (*rule spoly-repI-zero*)
next
  case *False*
  let *?v* = *term-of-pair* (*lcs* (*lp g1*) (*lp g2*), *component-of-term* (*lt g1*))
  let *?h* = *crit-pair g1 g2*
  from *assms*(*7*, *8*) **have** *eq*: *spoly g1 g2* = *fst ?h* + (− *snd ?h*) **by** (*simp add*: *spoly-alt*)
  have *fst ?h* $\prec_p$ *monomial 1 ?v* **by** (*fact fst-crit-pair-below-lcs*)
  hence *d1*: *fst ?h* = *0* $\vee$ *lt* (*fst ?h*) $\prec_t$ *?v* **by** (*simp only*: *ord-strict-p-monomial-iff*)
  have *snd ?h* $\prec_p$ *monomial 1 ?v* **by** (*fact snd-crit-pair-below-lcs*)

```
 hence d2: snd ?h = 0 ∨ lt (− snd ?h) ≺t ?v by (simp only: ord-strict-p-monomial-iff
lt-uminus)
  note assms(1)
  moreover from assms(5, 3) have g1 ∈ dgrad-p-set d m ..
  moreover from assms(6, 3) have g2 ∈ dgrad-p-set d m ..
  ultimately have spoly g1 g2 ∈ dgrad-p-set d m by (rule dgrad-p-set-closed-spoly)
  from assms(5) have g1 ∈ pmdl G by (rule pmdl.span-base)
  moreover from assms(6) have g2 ∈ pmdl G by (rule pmdl.span-base)
  ultimately have spoly g1 g2 ∈ pmdl G by (rule pmdl-closed-spoly)
  with assms(2) have (red G)** (spoly g1 g2) 0 by (rule GB-imp-zero-reducibility)
  with assms(1, 3, 4) ‹spoly - - ∈ dgrad-p-set - -› obtain q
   where 1: spoly g1 g2 = 0 + (∑ g∈G. q g ⊙ g) and 2: ⋀g. q g ∈ punit.dgrad-p-set
d m
     and ⋀g. lt (q g ⊙ g) ⪯t lt (spoly g1 g2) by (rule red-rtrancl-repE) blast
  show ?thesis
  proof (rule spoly-repI)
    fix g
    note ‹lt (q g ⊙ g) ⪯t lt (spoly g1 g2)›
    also from d1 have lt (spoly g1 g2) ≺t ?v
    proof
      assume fst ?h = 0
      hence eq: spoly g1 g2 = − snd ?h by (simp add: eq)
      also from d2 have lt ... ≺t ?v
      proof
        assume snd ?h = 0
        with False show ?thesis by (simp add: eq)
      qed
      finally show ?thesis .
    next
      assume *: lt (fst ?h) ≺t ?v
      from d2 show ?thesis
      proof
        assume snd ?h = 0
        with * show ?thesis by (simp add: eq)
      next
        assume **: lt (− snd ?h) ≺t ?v
          have lt (spoly g1 g2) ⪯t ord-term-lin.max (lt (fst ?h)) (lt (− snd ?h))
unfolding eq
          by (fact lt-plus-le-max)
       also from * ** have ... ≺t ?v by (simp only: ord-term-lin.max-less-iff-conj)
        finally show ?thesis .
      qed
    qed
   also from False have ... = term-of-pair (lcs (lp g1) (lp g2), component-of-term
(lt g2))
      by (simp add: spoly-def Let-def split: if-split-asm)
   finally show lt (q g ⊙ g) ≺t term-of-pair (lcs (lp g1) (lp g2), component-of-term
(lt g2)) .
  qed (simp-all add: 1 2)
```

qed

The finiteness assumption on G in the following theorem could be dropped, but it makes the proof a lot easier (although it is still fairly complicated).

lemma *isGB-I-spoly-rep*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m* **and** *finite G*
    and $\bigwedge$*g1 g2*. *g1* $\in G \Longrightarrow$ *g2* $\in G \Longrightarrow$ *g1* $\neq 0 \Longrightarrow$ *g2* $\neq 0 \Longrightarrow$ *spoly g1 g2* $\neq 0 \Longrightarrow$ *spoly-rep d m G g1 g2*
  shows *is-Groebner-basis G*
proof (*rule ccontr*)
  assume ¬ *is-Groebner-basis G*
  then obtain *p* **where** $p \in$ *pmdl G* **and** *p-in*: $p \in$ *dgrad-p-set d m* **and** ¬ (*red G*)** *p 0*
    by (*auto simp*: *GB-alt-1-dgrad-p-set*[*OF assms*(*1*, *2*)])
  from ‹¬ *is-Groebner-basis G*› **have** $G \neq \{\}$ **by** (*auto simp*: *is-Groebner-basis-empty*)

  obtain *r* **where** *p-red*: (*red G*)** *p r* **and** *r-irred*: ¬ *is-red G r*
  proof −
    define *A* **where** $A = \{q.$ (*red G*)** *p q*$\}$
    from *assms*(*1*, *2*) **have** *wfP* (*red G*)$^{-1-1}$ **by** (*rule red-wf-dgrad-p-set*)
    moreover have $p \in A$ **by** (*simp add*: *A-def*)
    ultimately obtain *r* **where** $r \in A$ **and** *r-min*: $\bigwedge$*z*. (*red G*)$^{-1-1}$ *z r* $\Longrightarrow z \notin A$
      by (*rule wfE-min*[*to-pred*]) *blast*
    show *?thesis*
    proof
      from ‹$r \in A$› **show** ∗: (*red G*)** *p r* **by** (*simp add*: *A-def*)

      show ¬ *is-red G r*
      proof
        assume *is-red G r*
        then obtain *z* **where** (*red G*) *r z* **by** (*rule is-redE*)
        hence (*red G*)$^{-1-1}$ *z r* **by** *simp*
        hence $z \notin A$ **by** (*rule r-min*)
        hence ¬ (*red G*)** *p z* **by** (*simp add*: *A-def*)
        moreover from ∗ ‹(*red G*) *r z*› **have** (*red G*)** *p z* **..**
        ultimately show *False* **..**
      qed
    qed
  qed
  from *assms*(*1*, *2*) *p-in p-red* **have** *r-in*: $r \in$ *dgrad-p-set d m* **by** (*rule dgrad-p-set-closed-red-rtrancl*)
  from *p-red* ‹¬ (*red G*)** *p 0*› **have** $r \neq 0$ **by** *blast*
  from *p-red* **have** $p - r \in$ *pmdl G* **by** (*rule red-rtranclp-diff-in-pmdl*)
  with ‹$p \in$ *pmdl G*› **have** $p - (p - r) \in$ *pmdl G* **by** (*rule pmdl.span-diff*)
  hence $r \in$ *pmdl G* **by** *simp*
  with *assms*(*3*) **obtain** *q0* **where** *r*: $r = (\sum g \in G.$ *q0 g* $\odot$ *g*$)$ **by** (*rule pmdl.span-finiteE*)
  from *assms*(*3*) **have** *finite* (*q0* ‘ *G*) **by** (*rule finite-imageI*)
  then obtain *m0* **where** *q0* ‘ $G \subseteq$ *punit.dgrad-p-set d m0* **by** (*rule punit.dgrad-p-set-exhaust*)
  define m' **where** $m' =$ *ord-class.max m m0*

have *dgrad-p-set d m* $\subseteq$ *dgrad-p-set d m'* **by** (*rule dgrad-p-set-subset*) (*simp add*: *m'-def*)
with *assms*(*2*) **have** *G-sub*: $G \subseteq$ *dgrad-p-set d m'* **by** (*rule subset-trans*)
have *punit.dgrad-p-set d m0* $\subseteq$ *punit.dgrad-p-set d m'*
by (*rule punit.dgrad-p-set-subset*) (*simp add*: *m'-def*)
with ‹*q0* ‘ $G \subseteq$ -› **have** *q0* ‘ $G \subseteq$ *punit.dgrad-p-set d m'* **by** (*rule subset-trans*)

define *mlt* **where** *mlt* = (λq. *ord-term-lin.Max* (*lt* ‘ $\{q\ g \odot g \mid g.\ g \in G \wedge q\ g \odot g \neq 0\}$))
define *mnum* **where** *mnum* = (λq. *card* $\{g{\in}G.\ q\ g \odot g \neq 0 \wedge$ *lt* $(q\ g \odot g) =$ *mlt q*$\}$)
define *rel* **where** *rel* = ($\lambda q1\ q2$. *mlt q1* $\prec_t$ *mlt q2* $\vee$ (*mlt q1* = *mlt q2* $\wedge$ *mnum q1* < *mnum q2*))
define *rel-dom* **where** *rel-dom* = $\{q.\ q$ ‘ $G \subseteq$ *punit.dgrad-p-set d m'* $\wedge\ r = (\sum g{\in}G.\ q\ g \odot g)\}$

have *mlt-in*: *mlt q* $\in$ *lt* ‘ $\{q\ g \odot g \mid g.\ g \in G \wedge q\ g \odot g \neq 0\}$ **if** $q \in$ *rel-dom* **for** *q*
unfolding *mlt-def*
proof (*rule ord-term-lin.Max-in*, *simp-all add*: *assms*(*3*), *rule ccontr*)
assume $\nexists g.\ g \in G \wedge q\ g \odot g \neq 0$
hence $q\ g \odot g = 0$ **if** $g \in G$ **for** *g* **using** *that* **by** *simp*
with *that* **have** $r = 0$ **by** (*simp add*: *rel-dom-def*)
with ‹$r \neq 0$› **show** *False* **..**
qed

have *rel-dom-dgrad-set*: *pp-of-term* ‘ *mlt* ‘ *rel-dom* $\subseteq$ *dgrad-set d m'*
proof (*rule subsetI*, *elim imageE*)
fix *q v t*
assume $q \in$ *rel-dom* **and** *v*: $v =$ *mlt q* **and** *t*: $t =$ *pp-of-term v*
from *this*(*1*) **have** $v \in$ *lt* ‘ $\{q\ g \odot g \mid g.\ g \in G \wedge q\ g \odot g \neq 0\}$ **unfolding** *v* **by** (*rule mlt-in*)
then obtain *g* **where** $g \in G$ **and** $q\ g \odot g \neq 0$ **and** *v*: $v =$ *lt* $(q\ g \odot g)$ **by** *blast*
from *this*(*2*) **have** $q\ g \neq 0$ **and** $g \neq 0$ **by** *auto*
hence $v =$ *punit.lt* $(q\ g) \oplus$ *lt g* **unfolding** *v* **by** (*rule lt-mult-scalar*)
hence $t =$ *punit.lt* $(q\ g)$ + *lp g* **by** (*simp add*: *t pp-of-term-splus*)
also from *assms*(*1*) **have** *d* ... = *ord-class.max* (*d* (*punit.lt* (*q g*))) (*d* (*lp g*))
by (*rule dickson-gradingD1*)
also have ... $\leq m'$
proof (*rule max.boundedI*)
from ‹$g \in G$› ‹$q \in$ *rel-dom*› **have** $q\ g \in$ *punit.dgrad-p-set d m'* **by** (*auto simp*: *rel-dom-def*)
moreover from ‹$q\ g \neq 0$› **have** *punit.lt* $(q\ g) \in$ *keys* $(q\ g)$ **by** (*rule punit.lt-in-keys*)
ultimately show *d* (*punit.lt* (*q g*)) $\leq m'$ **by** (*rule punit.dgrad-p-setD*[*simplified*])
next
from ‹$g \in G$› *G-sub* **have** $g \in$ *dgrad-p-set d m'* **..**
moreover from ‹$g \neq 0$› **have** *lt g* $\in$ *keys g* **by** (*rule lt-in-keys*)

```
      ultimately show d (lp g) ≤ m′ by (rule dgrad-p-setD)
    qed
    finally show t ∈ dgrad-set d m′ by (simp add: dgrad-set-def)
  qed

  obtain q where q ∈ rel-dom and q-min: ⋀q′. rel q′ q ⟹ q′ ∉ rel-dom
  proof −
    from ‹q0 ‘ G ⊆ punit.dgrad-p-set d m′› have q0 ∈ rel-dom by (simp add: rel-dom-def r)
    hence mlt q0 ∈ mlt ‘ rel-dom by (rule imageI)
    with assms(1) obtain u where u ∈ mlt ‘ rel-dom and u-min: ⋀w. w ≺t u ⟹ w ∉ mlt ‘ rel-dom
      using rel-dom-dgrad-set by (rule ord-term-minimum-dgrad-set) blast
    from this(1) obtain q′ where q′ ∈ rel-dom and u: u = mlt q′ ..
    hence q′ ∈ rel-dom ∩ {q. mlt q = u} (is - ∈ ?A) by simp
    hence mnum q′ ∈ mnum ‘ ?A by (rule imageI)
    with wf[to-pred] obtain k where k ∈ mnum ‘ ?A and k-min: ⋀l. l < k ⟹ l ∉ mnum ‘ ?A
      by (rule wfE-min[to-pred]) blast
    from this(1) obtain q′′ where q′′ ∈ rel-dom and mlt′′: mlt q′′ = u and k: k = mnum q′′
      by blast
    from this(1) show ?thesis
    proof
      fix q0
      assume rel q0 q′′
      show q0 ∉ rel-dom
      proof
        assume q0 ∈ rel-dom
        from ‹rel q0 q′′› show False unfolding rel-def
        proof (elim disjE conjE)
          assume mlt q0 ≺t mlt q′′
          hence mlt q0 ∉ mlt ‘ rel-dom unfolding mlt′′ by (rule u-min)
          moreover from ‹q0 ∈ rel-dom› have mlt q0 ∈ mlt ‘ rel-dom by (rule imageI)
          ultimately show ?thesis ..
        next
          assume mlt q0 = mlt q′′
          with ‹q0 ∈ rel-dom› have q0 ∈ ?A by (simp add: mlt′′)
          assume mnum q0 < mnum q′′
          hence mnum q0 ∉ mnum ‘ ?A unfolding k[symmetric] by (rule k-min)
          with ‹q0 ∈ ?A› show ?thesis by blast
        qed
      qed
    qed
  qed
  from this(1) have q-in: ⋀g. g ∈ G ⟹ q g ∈ punit.dgrad-p-set d m′
    and r: r = (∑g∈G. q g ⊙ g) by (auto simp: rel-dom-def)
```

define *v* **where** $v = mlt\ q$
from ‹$q \in rel\text{-}dom$› **have** $v \in lt$ ‘ $\{q\ g \odot g \mid g.\ g \in G \wedge q\ g \odot g \neq 0\}$ **unfolding** *v-def*
by (*rule mlt-in*)
then obtain *g1* **where** $g1 \in G$ **and** $q\ g1 \odot g1 \neq 0$ **and** *v1*: $v = lt\ (q\ g1 \odot g1)$ **by** *blast*
moreover define *M* **where** $M = \{g{\in}G.\ q\ g \odot g \neq 0 \wedge lt\ (q\ g \odot g) = v\}$
ultimately have $g1 \in M$ **by** *simp*
have *v-max*: $lt\ (q\ g \odot g) \prec_t v$ **if** $g \in G$ **and** $g \notin M$ **and** $q\ g \odot g \neq 0$ **for** *g*
proof −
from *that* **have** $lt\ (q\ g \odot g) \neq v$ **by** (*auto simp*: *M-def*)
moreover have $lt\ (q\ g \odot g) \preceq_t v$ **unfolding** *v-def mlt-def*
by (*rule ord-term-lin.Max-ge*) (*auto simp*: *assms*(*3*) ‹$q\ g \odot g \neq 0$› *intro*!: *imageI* ‹$g \in G$›)
ultimately show *?thesis* **by** *simp*
qed
from ‹$q\ g1 \odot g1 \neq 0$› **have** $q\ g1 \neq 0$ **and** $g1 \neq 0$ **by** *auto*
hence *v1'*: $v = punit.lt\ (q\ g1) \oplus lt\ g1$ **unfolding** *v1* **by** (*rule lt-mult-scalar*)
have $M - \{g1\} \neq \{\}$
proof
assume $M - \{g1\} = \{\}$
have $v \in keys\ (q\ g1 \odot g1)$ **unfolding** *v1* **using** ‹$q\ g1 \odot g1 \neq 0$› **by** (*rule lt-in-keys*)
moreover have $v \notin keys\ (\sum g{\in}G{-}\{g1\}.\ q\ g \odot g)$
proof
assume $v \in keys\ (\sum g{\in}G{-}\{g1\}.\ q\ g \odot g)$
also have $\ldots \subseteq (\bigcup g{\in}G{-}\{g1\}.\ keys\ (q\ g \odot g))$ **by** (*fact keys-sum-subset*)
finally obtain *g* **where** $g \in G - \{g1\}$ **and** $v \in keys\ (q\ g \odot g)$ **..**
from *this*(*2*) **have** $q\ g \odot g \neq 0$ **and** $v \preceq_t lt\ (q\ g \odot g)$ **by** (*auto intro*: *lt-max-keys*)
from ‹$g \in G - \{g1\}$› ‹$M - \{g1\} = \{\}$› **have** $g \in G$ **and** $g \notin M$ **by** *blast+*
hence $lt\ (q\ g \odot g) \prec_t v$ **by** (*rule v-max*) *fact*
with ‹$v \preceq_t$ -› **show** *False* **by** *simp*
qed
ultimately have $v \in keys\ (q\ g1 \odot g1 + (\sum g{\in}G{-}\{g1\}.\ q\ g \odot g))$ **by** (*rule in-keys-plusI1*)
also from ‹$g1 \in G$› *assms*(*3*) **have** $\ldots = keys\ r$ **by** (*simp add*: *r sum.remove*)
finally have $v \in keys\ r$ **.**
with ‹$g1 \in G$› ‹$g1 \neq 0$› **have** *is-red G r* **by** (*rule is-red-addsI*) (*simp add*: *v1' term-simps*)
with *r-irred* **show** *False* **..**
qed
then obtain *g2* **where** $g2 \in M$ **and** $g1 \neq g2$ **by** *blast*
from *this*(*1*) **have** $g2 \in G$ **and** $q\ g2 \odot g2 \neq 0$ **and** *v2*: $v = lt\ (q\ g2 \odot g2)$ **by** (*simp-all add*: *M-def*)
from *this*(*2*) **have** $q\ g2 \neq 0$ **and** $g2 \neq 0$ **by** *auto*
hence *v2'*: $v = punit.lt\ (q\ g2) \oplus lt\ g2$ **unfolding** *v2* **by** (*rule lt-mult-scalar*)
hence *component-of-term* $(punit.lt\ (q\ g1) \oplus lt\ g1) =$ *component-of-term* $(punit.lt\ (q\ g2) \oplus lt\ g2)$

by (*simp only*: *v1′ flip*: *v2′*)
hence *cmp-eq*: *component-of-term* (*lt g1*) = *component-of-term* (*lt g2*) **by** (*simp add*: *term-simps*)

have *M* ⊆ *G* **by** (*simp add*: *M-def*)
have *r* = *q g1* ⊙ *g1* + (∑ *g*∈*G* − {*g1*}. *q g* ⊙ *g*)
using *assms*(*3*) ‹*g1* ∈ *G*› **by** (*simp add*: *r sum.remove*)
also have ... = *q g1* ⊙ *g1* + *q g2* ⊙ *g2* + (∑ *g*∈*G* − {*g1*} − {*g2*}. *q g* ⊙ *g*)
using *assms*(*3*) ‹*g2* ∈ *G*› ‹*g1* ≠ *g2*›
by (*metis* (*no-types*, *lifting*) *add.assoc finite-Diff insert-Diff insert-Diff-single insert-iff*
sum.insert-remove)
finally have *r*: *r* = *q g1* ⊙ *g1* + *q g2* ⊙ *g2* + (∑ *g*∈*G* − {*g1*, *g2*}. *q g* ⊙ *g*)
by (*simp flip*: *Diff-insert2*)

let *?l* = *lcs* (*lp g1*) (*lp g2*)
let *?v* = *term-of-pair* (*?l*, *component-of-term* (*lt g2*))
have *lp g1 adds lp* (*q g1* ⊙ *g1*) **by** (*simp add*: *v1′ pp-of-term-splus flip*: *v1*)
moreover have *lp g2 adds lp* (*q g1* ⊙ *g1*) **by** (*simp add*: *v2′ pp-of-term-splus flip*: *v1*)
ultimately have *l-adds*: *?l adds lp* (*q g1* ⊙ *g1*) **by** (*rule lcs-adds*)

have *spoly-rep d m G g1 g2*
proof (*cases spoly g1 g2* = *0*)
case *True*
thus *?thesis* **by** (*rule spoly-repI-zero*)
next
case *False*
with ‹*g1* ∈ *G*› ‹*g2* ∈ *G*› ‹*g1* ≠ *0*› ‹*g2* ≠ *0*› **show** *?thesis* **by** (*rule assms*(*4*))
qed
then obtain *q′* **where** *spoly*: *spoly g1 g2* = (∑ *g*∈*G*. *q′ g* ⊙ *g*)
and ⋀*g*. *q′ g* ∈ *punit.dgrad-p-set d m* **and** ⋀*g*. *q′ g* ⊙ *g* ≠ *0* ⟹ *lt* (*q′ g* ⊙ *g*) $\prec_t$ *?v*
by (*rule spoly-repE*) *blast*
note *this*(*2*)
also have *punit.dgrad-p-set d m* ⊆ *punit.dgrad-p-set d m′*
by (*rule punit.dgrad-p-set-subset*) (*simp add*: *m′-def*)
finally have *q′-in*: ⋀*g*. *q′ g* ∈ *punit.dgrad-p-set d m′* **.**

define *mu* **where** *mu* = *monomial* (*lc* (*q g1* ⊙ *g1*)) (*lp* (*q g1* ⊙ *g1*) − *?l*)
define *mu1* **where** *mu1* = *monomial* (*1* / *lc g1*) (*?l* − *lp g1*)
define *mu2* **where** *mu2* = *monomial* (*1* / *lc g2*) (*?l* − *lp g2*)
define *q″* **where** *q″* = (λ*g*. *q g* + *mu* ∗ *q′ g*)
(*g1*:=*punit.tail* (*q g1*) + *mu* ∗ *q′ g1*, *g2*:=*q g2* + *mu* ∗ *q′ g2* + *mu* ∗ *mu2*)
from ‹*q g1* ⊙ *g1* ≠ *0*› **have** *mu* ≠ *0* **by** (*simp add*: *mu-def monomial-0-iff lc-eq-zero-iff*)
from ‹*g1* ≠ *0*› *l-adds* **have** *mu-times-mu1*: *mu* ∗ *mu1* = *monomial* (*punit.lc* (*q g1*)) (*punit.lt* (*q g1*))

by (*simp add*: *mu-def mu1-def times-monomial-monomial lc-mult-scalar lc-eq-zero-iff minus-plus-minus-cancel adds-lcs v1*′ *pp-of-term-splus flip*: *v1*)
from *l-adds* **have** *mu-times-mu2*: $mu * mu2 = monomial\ (lc\ (q\ g1 \odot g1)\ /\ lc\ g2)\ (punit.lt\ (q\ g2))$
by (*simp add*: *mu-def mu2-def times-monomial-monomial lc-mult-scalar minus-plus-minus-cancel adds-lcs-2 v2*′ *pp-of-term-splus flip*: *v1*)
have $mu1 \odot g1 - mu2 \odot g2 = spoly\ g1\ g2$
by (*simp add*: *spoly-def Let-def cmp-eq lc-def mult-scalar-monomial mu1-def mu2-def*)
also have $\ldots = q'\ g1 \odot g1 + (\sum g\in G - \{g1\}.\ q'\ g \odot g)$
using *assms*(*3*) ‹$g1 \in G$› **by** (*simp add*: *spoly sum.remove*)
also have $\ldots = q'\ g1 \odot g1 + q'\ g2 \odot g2 + (\sum g\in G - \{g1\} - \{g2\}.\ q'\ g \odot g)$
using *assms*(*3*) ‹$g2 \in G$› ‹$g1 \neq g2$›
by (*metis* (*no-types*, *lifting*) *add.assoc finite-Diff insert-Diff insert-Diff-single insert-iff sum.insert-remove*)
finally have $(q'\ g1 - mu1) \odot g1 + (q'\ g2 + mu2) \odot g2 + (\sum g\in G - \{g1, g2\}.\ q'\ g \odot g) = 0$
by (*simp add*: *algebra-simps flip*: *Diff-insert2*)
hence $0 = mu \odot ((q'\ g1 - mu1) \odot g1 + (q'\ g2 + mu2) \odot g2 + (\sum g\in G - \{g1, g2\}.\ q'\ g \odot g))$ **by** *simp*
also have $\ldots = (mu * q'\ g1 - mu * mu1) \odot g1 + (mu * q'\ g2 + mu * mu2) \odot g2 + (\sum g\in G - \{g1, g2\}.\ (mu * q'\ g) \odot g)$
by (*simp add*: *mult-scalar-distrib-left sum-mult-scalar-distrib-left distrib-left right-diff-distrib flip*: *mult-scalar-assoc*)
finally have $r = r + (mu * q'\ g1 - mu * mu1) \odot g1 + (mu * q'\ g2 + mu * mu2) \odot g2 + (\sum g\in G - \{g1, g2\}.\ (mu * q'\ g) \odot g)$ **by** *simp*
also have $\ldots = (q\ g1 - mu * mu1 + mu * q'\ g1) \odot g1 + (q\ g2 + mu * q'\ g2 + mu * mu2) \odot g2 + (\sum g\in G - \{g1, g2\}.\ (q\ g + mu * q'\ g) \odot g)$
by (*simp add*: *r algebra-simps flip*: *sum.distrib*)
also have $q\ g1 - mu * mu1 = punit.tail\ (q\ g1)$
by (*simp only*: *mu-times-mu1 punit.leading-monomial-tail diff-eq-eq add.commute*[*of punit.tail* (*q g1*)])
finally have $r = q''\ g1 \odot g1 + q''\ g2 \odot g2 + (\sum g\in G - \{g1\} - \{g2\}.\ q''\ g \odot g)$
using ‹$g1 \neq g2$› **by** (*simp add*: *q*′′*-def flip*: *Diff-insert2*)
also from ‹*finite G*› ‹$g1 \neq g2$› ‹$g1 \in G$› ‹$g2 \in G$› **have** $\ldots = (\sum g\in G.\ q''\ g \odot g)$
by (*simp add*: *sum.remove*) (*metis* (*no-types*, *lifting*) *finite-Diff insert-Diff insert-iff sum.remove*)
finally have *r*: $r = (\sum g\in G.\ q''\ g \odot g)$ **.**

have *1*: $lt\ ((mu * q'\ g) \odot g) \prec_t v$ **if** $(mu * q'\ g) \odot g \neq 0$ **for** *g*
proof −

from *that* **have** $q'\ g \odot g \neq 0$ **by** (*auto simp*: *mult-scalar-assoc*)
hence ∗: *lt* $(q'\ g \odot g) \prec_t$ *?v* **by** *fact*
from ‹$q'\ g \odot g \neq 0$› ‹$mu \neq 0$› **have** *lt* $((mu * q'\ g) \odot g) = (lp\ (q\ g1 \odot g1) - ?l) \oplus lt\ (q'\ g \odot g)$
by (*simp add*: *mult-scalar-assoc lt-mult-scalar*) (*simp add*: *mu-def punit.lt-monomial monomial-0-iff*)
also from ∗ **have** ... $\prec_t (lp\ (q\ g1 \odot g1) - ?l) \oplus ?v$ **by** (*rule splus-mono-strict*)
also from *l-adds* **have** ... $= v$ **by** (*simp add*: *splus-def minus-plus term-simps v1' flip*: *cmp-eq v1*)
finally show *?thesis* **.**
qed

have *2*: *lt* $(q''\ g1 \odot g1) \prec_t v$ **if** $q''\ g1 \odot g1 \neq 0$ **using** *that*
proof (*rule lt-less*)
fix *u*
assume $v \preceq_t u$
have $u \notin$ *keys* $(q''\ g1 \odot g1)$
proof
assume $u \in$ *keys* $(q''\ g1 \odot g1)$
also from ‹$g1 \neq g2$› **have** ... = *keys* $((punit.tail\ (q\ g1) + mu * q'\ g1) \odot g1)$
by (*simp add*: *q''-def*)
also have ... $\subseteq$ *keys* $(punit.tail\ (q\ g1) \odot g1) \cup$ *keys* $((mu * q'\ g1) \odot g1)$
unfolding *mult-scalar-distrib-right* **by** (*fact Poly-Mapping.keys-add*)
finally show *False*
proof
assume $u \in$ *keys* $(punit.tail\ (q\ g1) \odot g1)$
hence $u \preceq_t lt\ (punit.tail\ (q\ g1) \odot g1)$ **by** (*rule lt-max-keys*)
also have ... $\preceq_t punit.lt\ (punit.tail\ (q\ g1)) \oplus lt\ g1$
by (*metis in-keys-mult-scalar-le lt-def lt-in-keys min-term-min*)
also have ... $\prec_t punit.lt\ (q\ g1) \oplus lt\ g1$
proof (*intro splus-mono-strict-left punit.lt-tail notI*)
assume $punit.tail\ (q\ g1) = 0$
with ‹$u \in$ *keys* $(punit.tail\ (q\ g1) \odot g1)$› **show** *False* **by** *simp*
qed
also have ... $= v$ **by** (*simp only*: *v1'*)
finally show *?thesis* **using** ‹$v \preceq_t u$› **by** *simp*
next
assume $u \in$ *keys* $((mu * q'\ g1) \odot g1)$
hence $(mu * q'\ g1) \odot g1 \neq 0$ **and** $u \preceq_t lt\ ((mu * q'\ g1) \odot g1)$ **by** (*auto intro*: *lt-max-keys*)
note *this*(*2*)
also from ‹$(mu * q'\ g1) \odot g1 \neq 0$› **have** $lt\ ((mu * q'\ g1) \odot g1) \prec_t v$ **by** (*rule 1*)
finally show *?thesis* **using** ‹$v \preceq_t u$› **by** *simp*
qed
qed
thus *lookup* $(q''\ g1 \odot g1)\ u = 0$ **by** (*simp add*: *in-keys-iff*)
qed

have *3*: $lt\ (q''\ g2 \odot g2) \preceq_t v$
proof (*rule lt-le*)
fix *u*
assume $v \prec_t u$
have $u \notin keys\ (q''\ g2 \odot g2)$
proof
assume $u \in keys\ (q''\ g2 \odot g2)$
also have $\ldots = keys\ ((q\ g2 + mu * q'\ g2 + mu * mu2) \odot g2)$ **by** (*simp add*: *q''-def*)
also have $\ldots \subseteq keys\ (q\ g2 \odot g2 + (mu * q'\ g2) \odot g2) \cup keys\ ((mu * mu2) \odot g2)$
unfolding *mult-scalar-distrib-right* **by** (*fact Poly-Mapping.keys-add*)
finally show *False*
proof
assume $u \in keys\ (q\ g2 \odot g2 + (mu * q'\ g2) \odot g2)$
also have $\ldots \subseteq keys\ (q\ g2 \odot g2) \cup keys\ ((mu * q'\ g2) \odot g2)$ **by** (*fact Poly-Mapping.keys-add*)
finally show *?thesis*
proof
assume $u \in keys\ (q\ g2 \odot g2)$
hence $u \preceq_t lt\ (q\ g2 \odot g2)$ **by** (*rule lt-max-keys*)
with ‹$v \prec_t u$› **show** *?thesis* **by** (*simp add*: *v2*)
next
assume $u \in keys\ ((mu * q'\ g2) \odot g2)$
hence $(mu * q'\ g2) \odot g2 \neq 0$ **and** $u \preceq_t lt\ ((mu * q'\ g2) \odot g2)$ **by** (*auto intro*: *lt-max-keys*)
note *this*(*2*)
also from ‹$(mu * q'\ g2) \odot g2 \neq 0$› **have** $lt\ ((mu * q'\ g2) \odot g2) \prec_t v$ **by** (*rule 1*)
finally show *?thesis* **using** ‹$v \prec_t u$› **by** *simp*
qed
next
assume $u \in keys\ ((mu * mu2) \odot g2)$
hence $(mu * mu2) \odot g2 \neq 0$ **and** $u \preceq_t lt\ ((mu * mu2) \odot g2)$ **by** (*auto intro*: *lt-max-keys*)
from *this*(*1*) **have** $(mu * mu2) \neq 0$ **by** *auto*
note ‹$u \preceq_t$ -›
also from ‹$mu * mu2 \neq 0$› ‹$g2 \neq 0$› **have** $lt\ ((mu * mu2) \odot g2) = punit.lt\ (q\ g2) \oplus lt\ g2$
by (*simp add*: *lt-mult-scalar*) (*simp add*: *mu-times-mu2 punit.lt-monomial monomial-0-iff*)
finally show *?thesis* **using** ‹$v \prec_t u$› **by** (*simp add*: *v2'*)
qed
qed
thus $lookup\ (q''\ g2 \odot g2)\ u = 0$ **by** (*simp add*: *in-keys-iff*)
qed

have *4*: $lt\ (q''\ g \odot g) \preceq_t v$ **if** $g \in M$ **for** *g*

proof (*cases* $g \in \{g1, g2\}$)
case *True*
hence $g = g1 \lor g = g2$ **by** *simp*
thus *?thesis*
proof
assume $g = g1$
show *?thesis*
proof (*cases* $q'' \ g1 \odot g1 = 0$)
case *True*
thus *?thesis* **by** (*simp add*: ‹$g = g1$› *min-term-min*)
next
case *False*
hence *lt* $(q'' \ g \odot g) \prec_t v$ **unfolding** ‹$g = g1$› **by** (*rule 2*)
thus *?thesis* **by** *simp*
qed
next
assume $g = g2$
with *3* **show** *?thesis* **by** *simp*
qed
next
case *False*
hence q'': $q'' \ g = q \ g + mu * q' \ g$ **by** (*simp add*: q''-*def*)
show *?thesis*
proof (*rule lt-le*)
fix u
assume $v \prec_t u$
have $u \notin$ *keys* $(q'' \ g \odot g)$
proof
assume $u \in$ *keys* $(q'' \ g \odot g)$
also have ... $\subseteq$ *keys* $(q \ g \odot g) \cup$ *keys* $((mu * q' \ g) \odot g)$
unfolding q'' *mult-scalar-distrib-right* **by** (*fact Poly-Mapping.keys-add*)
finally show *False*
proof
assume $u \in$ *keys* $(q \ g \odot g)$
hence $u \preceq_t$ *lt* $(q \ g \odot g)$ **by** (*rule lt-max-keys*)
with ‹$g \in M$› ‹$v \prec_t u$› **show** *?thesis* **by** (*simp add*: *M-def*)
next
assume $u \in$ *keys* $((mu * q' \ g) \odot g)$
hence $(mu * q' \ g) \odot g \neq 0$ **and** $u \preceq_t$ *lt* $((mu * q' \ g) \odot g)$ **by** (*auto intro*: *lt-max-keys*)
note *this*(*2*)
also from ‹$(mu * q' \ g) \odot g \neq 0$› **have** *lt* $((mu * q' \ g) \odot g) \prec_t v$ **by** (*rule 1*)
finally show *?thesis* **using** ‹$v \prec_t u$› **by** *simp*
qed
qed
thus *lookup* $(q'' \ g \odot g) \ u = 0$ **by** (*simp add*: *in-keys-iff*)
qed
qed

have *5*: $lt\ (q''\ g \odot g) \prec_t v$ **if** $g \in G$ **and** $g \notin M$ **and** $q''\ g \odot g \neq 0$ **for** g **using** *that*(*3*)
proof (*rule lt-less*)
fix u
assume $v \preceq_t u$
from *that*(*2*) ‹$g1 \in M$› ‹$g2 \in M$› **have** $g \neq g1$ **and** $g \neq g2$ **by** *blast+*
hence q'': $q''\ g = q\ g + mu * q'\ g$ **by** (*simp add*: q''-*def*)
have $u \notin keys\ (q''\ g \odot g)$
proof
assume $u \in keys\ (q''\ g \odot g)$
also have $\ldots \subseteq keys\ (q\ g \odot g) \cup keys\ ((mu * q'\ g) \odot g)$
unfolding q'' *mult-scalar-distrib-right* **by** (*fact Poly-Mapping.keys-add*)
finally show *False*
proof
assume $u \in keys\ (q\ g \odot g)$
hence $q\ g \odot g \neq 0$ **and** $u \preceq_t lt\ (q\ g \odot g)$ **by** (*auto intro*: *lt-max-keys*)
note *this*(*2*)
also from *that*(*1*, *2*) ‹$q\ g \odot g \neq 0$› **have** $\ldots \prec_t v$ **by** (*rule v-max*)
finally show *?thesis* **using** ‹$v \preceq_t u$› **by** *simp*
next
assume $u \in keys\ ((mu * q'\ g) \odot g)$
hence $(mu * q'\ g) \odot g \neq 0$ **and** $u \preceq_t lt\ ((mu * q'\ g) \odot g)$ **by** (*auto intro*: *lt-max-keys*)
note *this*(*2*)
also from ‹$(mu * q'\ g) \odot g \neq 0$› **have** $lt\ ((mu * q'\ g) \odot g) \prec_t v$ **by** (*rule 1*)
finally show *?thesis* **using** ‹$v \preceq_t u$› **by** *simp*
qed
qed
thus $lookup\ (q''\ g \odot g)\ u = 0$ **by** (*simp add*: *in-keys-iff*)
qed

define u **where** $u = mlt\ q''$
have *u-in*: $u \in lt$ ‘ $\{q''\ g \odot g \mid g.\ g \in G \wedge q''\ g \odot g \neq 0\}$ **unfolding** *u-def mlt-def*
proof (*rule ord-term-lin.Max-in*, *simp-all add*: *assms*(*3*), *rule ccontr*)
assume $\nexists g.\ g \in G \wedge q''\ g \odot g \neq 0$
hence $q''\ g \odot g = 0$ **if** $g \in G$ **for** g **using** *that* **by** *simp*
hence $r = 0$ **by** (*simp add*: r)
with ‹$r \neq 0$› **show** *False* **..**
qed
have *u-max*: $lt\ (q''\ g \odot g) \preceq_t u$ **if** $g \in G$ **for** g
proof (*cases* $q''\ g \odot g = 0$)
case *True*
thus *?thesis* **by** (*simp add*: *min-term-min*)
next
case *False*
show *?thesis* **unfolding** *u-def mlt-def*

by (*rule ord-term-lin.Max-ge*) (*auto simp*: *assms*(*3*) *False intro*!: *imageI* ‹*g* ∈ *G*›)
qed
have *q''* ∈ *rel-dom*
proof (*simp add*: *rel-dom-def r*, *intro subsetI*, *elim imageE*)
fix *g*
assume *g* ∈ *G*
from *assms*(*1*) *l-adds* **have** *d* (*lp* (*q g1* ⊙ *g1*) − *?l*) ≤ *d* (*lp* (*q g1* ⊙ *g1*))
by (*rule dickson-grading-minus*)
also have ... = *d* (*punit.lt* (*q g1*) + *lp g1*) **by** (*simp add*: *v1' term-simps flip*: *v1*)
also from *assms*(*1*) **have** ... = *ord-class.max* (*d* (*punit.lt* (*q g1*))) (*d* (*lp g1*))
by (*rule dickson-gradingD1*)
also have ... ≤ *m'*
proof (*rule max.boundedI*)
from ‹*g1* ∈ *G*› **have** *q g1* ∈ *punit.dgrad-p-set d m'* **by** (*rule q-in*)
moreover from ‹*q g1* ≠ *0*› **have** *punit.lt* (*q g1*) ∈ *keys* (*q g1*) **by** (*rule punit.lt-in-keys*)
ultimately show *d* (*punit.lt* (*q g1*)) ≤ *m'* **by** (*rule punit.dgrad-p-setD*[*simplified*])
next
from ‹*g1* ∈ *G*› *G-sub* **have** *g1* ∈ *dgrad-p-set d m'* **..**
moreover from ‹*g1* ≠ *0*› **have** *lt g1* ∈ *keys g1* **by** (*rule lt-in-keys*)
ultimately show *d* (*lp g1*) ≤ *m'* **by** (*rule dgrad-p-setD*)
qed
finally have *d1*: *d* (*lp* (*q g1* ⊙ *g1*) − *?l*) ≤ *m'* **.**
have *d* (*?l* − *lp g2*) ≤ *ord-class.max* (*d* (*lp g2*)) (*d* (*lp g1*))
unfolding *lcs-comm*[*of lp g1*] **using** *assms*(*1*) **by** (*rule dickson-grading-lcs-minus*)
also have ... ≤ *m'*
proof (*rule max.boundedI*)
from ‹*g2* ∈ *G*› *G-sub* **have** *g2* ∈ *dgrad-p-set d m'* **..**
moreover from ‹*g2* ≠ *0*› **have** *lt g2* ∈ *keys g2* **by** (*rule lt-in-keys*)
ultimately show *d* (*lp g2*) ≤ *m'* **by** (*rule dgrad-p-setD*)
next
from ‹*g1* ∈ *G*› *G-sub* **have** *g1* ∈ *dgrad-p-set d m'* **..**
moreover from ‹*g1* ≠ *0*› **have** *lt g1* ∈ *keys g1* **by** (*rule lt-in-keys*)
ultimately show *d* (*lp g1*) ≤ *m'* **by** (*rule dgrad-p-setD*)
qed
finally have *mu2*: *mu2* ∈ *punit.dgrad-p-set d m'*
by (*simp add*: *mu2-def punit.dgrad-p-set-def dgrad-set-def*)
fix *z*
assume *z*: *z* = *q'' g*
have *g* = *g1* ∨ *g* = *g2* ∨ (*g* ≠ *g1* ∧ *g* ≠ *g2*) **by** *blast*
thus *z* ∈ *punit.dgrad-p-set d m'*
proof (*elim disjE conjE*)
assume *g* = *g1*
with ‹*g1* ≠ *g2*› **have** *q'' g* = *punit.tail* (*q g1*) + *mu* ∗ *q' g1* **by** (*simp add*: *q''-def*)
also have ... ∈ *punit.dgrad-p-set d m'* **unfolding** *mu-def times-monomial-left*
by (*intro punit.dgrad-p-set-closed-plus punit.dgrad-p-set-closed-tail*

punit.dgrad-p-set-closed-monom-mult d1 assms(*1*) *q-in q′-in* ‹*g1* ∈ *G*›)
finally show *?thesis* **by** (*simp only*: *z*)
next
assume $g = g2$
hence $q'' \ g = q \ g2 + mu * q' \ g2 + mu * mu2$ **by** (*simp add*: *q′′-def*)
also have ... ∈ *punit.dgrad-p-set d m′* **unfolding** *mu-def times-monomial-left*
by (*intro punit.dgrad-p-set-closed-plus punit.dgrad-p-set-closed-monom-mult d1 mu2 q-in q′-in assms*(*1*) ‹*g2* ∈ *G*›)
finally show *?thesis* **by** (*simp only*: *z*)
next
assume $g \neq g1$ **and** $g \neq g2$
hence $q'' \ g = q \ g + mu * q' \ g$ **by** (*simp add*: *q′′-def*)
also have ... ∈ *punit.dgrad-p-set d m′* **unfolding** *mu-def times-monomial-left*
by (*intro punit.dgrad-p-set-closed-plus punit.dgrad-p-set-closed-monom-mult d1 assms*(*1*) *q-in q′-in* ‹*g* ∈ *G*›)
finally show *?thesis* **by** (*simp only*: *z*)
qed
qed
with *q-min* **have** ¬ *rel q′′ q* **by** *blast*
hence $v \preceq_t u$ **and** $u \neq v \lor mnum \ q \leq mnum \ q''$ **by** (*auto simp*: *v-def u-def rel-def*)
moreover have $u \preceq_t v$
proof −
from *u-in* **obtain** *g* **where** *g* ∈ *G* **and** $q'' \ g \odot g \neq 0$ **and** *u*: $u = lt \ (q'' \ g \odot g)$ **by** *blast*
show *?thesis*
proof (*cases g* ∈ *M*)
case *True*
thus *?thesis* **unfolding** *u* **by** (*rule 4*)
next
case *False*
with ‹*g* ∈ *G*› **have** $lt \ (q'' \ g \odot g) \prec_t v$ **using** ‹$q'' \ g \odot g \neq 0$› **by** (*rule 5*)
thus *?thesis* **by** (*simp add*: *u*)
qed
qed
ultimately have *u-v*: $u = v$ **and** $mnum \ q \leq mnum \ q''$ **by** *simp-all*
note *this*(*2*)
also have $mnum \ q'' < card \ M$ **unfolding** *mnum-def*
proof (*rule psubset-card-mono*)
from ‹*M* ⊆ *G*› ‹*finite G*› **show** *finite M* **by** (*rule finite-subset*)
next
have $\{g{\in}G. \ q'' \ g \odot g \neq 0 \land lt \ (q'' \ g \odot g) = v\} \subseteq M - \{g1\}$
proof
fix *g*
assume $g \in \{g{\in}G. \ q'' \ g \odot g \neq 0 \land lt \ (q'' \ g \odot g) = v\}$
hence *g* ∈ *G* **and** $q'' \ g \odot g \neq 0$ **and** $lt \ (q'' \ g \odot g) = v$ **by** *simp-all*
with *2 5* **show** $g \in M - \{g1\}$ **by** *blast*
qed

also from ‹*g1* ∈ *M*› **have** ... ⊂ *M* **by** *blast*
finally show {*g*∈*G*. *q''* *g* ⊙ *g* ≠ *0* ∧ *lt* (*q''* *g* ⊙ *g*) = *mlt q''*} ⊂ *M*
by (*simp only*: *u-v flip*: *u-def*)
qed
also have ... = *mnum q* **by** (*simp only*: *M-def mnum-def v-def*)
finally show *False* **..**
qed

5.6 Replacing Elements in Gröbner Bases

lemma *replace-in-dgrad-p-set*:
assumes *G* ⊆ *dgrad-p-set d m*
obtains *n* **where** *q* ∈ *dgrad-p-set d n* **and** *G* ⊆ *dgrad-p-set d n*
and *insert q* (*G* − {*p*}) ⊆ *dgrad-p-set d n*
proof −
from *assms* **obtain** *n* **where** *m* ≤ *n* **and** *1*: *q* ∈ *dgrad-p-set d n* **and** *2*: *G* ⊆ *dgrad-p-set d n*
by (*rule dgrad-p-set-insert*)
from *this*(*2*, *3*) **have** *insert q* (*G* − {*p*}) ⊆ *dgrad-p-set d n* **by** *auto*
with *1 2* **show** *?thesis* **..**
qed

lemma *GB-replace-lt-adds-stable-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *G* ⊆ *dgrad-p-set d m*
assumes *isGB*: *is-Groebner-basis G* **and** *q* ≠ *0* **and** *q*: *q* ∈ (*pmdl G*) **and** *lt q* $adds_t$ *lt p*
shows *is-Groebner-basis* (*insert q* (*G* − {*p*})) (**is** *is-Groebner-basis ?G'*)
proof −
from *assms*(*2*) **obtain** *n* **where** *1*: *G* ⊆ *dgrad-p-set d n* **and** *2*: *?G'* ⊆ *dgrad-p-set d n*
by (*rule replace-in-dgrad-p-set*)
from *isGB* **show** *?thesis* **unfolding** *GB-alt-3-dgrad-p-set*[*OF assms*(*1*) *1*] *GB-alt-3-dgrad-p-set*[*OF assms*(*1*) *2*]
proof (*intro ballI impI*)
fix *f*
assume *f1*: *f* ∈ (*pmdl ?G'*) **and** *f* ≠ *0*
and *a1*: ∀ *f*∈*pmdl G*. *f* ≠ *0* ⟶ (∃ *g*∈*G*. *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*)
from *f1 pmdl.replace-span*[*OF q*, *of p*] **have** *f* ∈ *pmdl G* **..**
from *a1*[*rule-format*, *OF this* ‹*f* ≠ *0*›] **obtain** *g* **where** *g* ∈ *G* **and** *g* ≠ *0* **and** *lt g* $adds_t$ *lt f* **by** *auto*
show ∃ *g*∈*?G'*. *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
proof (*cases g* = *p*)
case *True*
show *?thesis*
proof
from ‹*lt q* $adds_t$ *lt p*› **have** *lt q* $adds_t$ *lt g* **unfolding** *True* **.**
also have ... $adds_t$ *lt f* **by** *fact*
finally have *lt q* $adds_t$ *lt f* **.**
with ‹*q* ≠ *0*› **show** *q* ≠ *0* ∧ *lt q* $adds_t$ *lt f* **..**

    next
      show $q \in ?G'$ **by** *simp*
    qed
  next
    case *False*
    show *?thesis*
    proof
      show $g \neq 0 \wedge lt\ g\ adds_t\ lt\ f$ **by** (*rule*, *fact+*)
    next
      from ‹$g \in G$› *False* **show** $g \in ?G'$ **by** *blast*
    qed
  qed
qed
qed

lemma *GB-replace-lt-adds-stable-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
assumes *isGB*: *is-Groebner-basis G* **and** $q \neq 0$ **and** $q \in$ *pmdl G* **and** *lt q* $adds_t$ *lt p*
shows *pmdl* (*insert q* ($G - \{p\}$)) = *pmdl G* (**is** *pmdl ?G'* = *pmdl G*)
proof (*rule*, *rule pmdl.replace-span*, *fact*, *rule*)
fix *f*
assume $f \in$ *pmdl G*
note *assms*(*1*)
moreover from *assms*(*2*) **obtain** *n* **where** $?G' \subseteq$ *dgrad-p-set d n* **by** (*rule replace-in-dgrad-p-set*)
moreover have *is-Groebner-basis ?G'* **by** (*rule GB-replace-lt-adds-stable-GB-dgrad-p-set*, *fact+*)
ultimately have $\exists!\ h.\ (red\ ?G')^{**}\ f\ h \wedge \neg$ *is-red ?G' h* **by** (*rule GB-implies-unique-nf-dgrad-p-set*)
then obtain *h* **where** *ftoh*: $(red\ ?G')^{**}\ f\ h$ **and** *irredh*: ¬ *is-red ?G' h* **by** *auto*
have ¬ *is-red G h*
proof
  assume *is-red G h*
  have *is-red ?G' h* **by** (*rule replace-lt-adds-stable-is-red*, *fact+*)
  with *irredh* **show** *False* **..**
qed
have $f - h \in$ *pmdl ?G'* **by** (*rule red-rtranclp-diff-in-pmdl*, *rule ftoh*)
have $f - h \in$ *pmdl G* **by** (*rule*, *fact*, *rule pmdl.replace-span*, *fact*)
from *pmdl.span-diff*[*OF this* ‹$f \in$ *pmdl G*›] **have** $-h \in$ *pmdl G* **by** *simp*
from *pmdl.span-neg*[*OF this*] **have** $h \in$ *pmdl G* **by** *simp*
with *isGB* ‹¬ *is-red G h*› **have** $h = 0$ **using** *GB-imp-reducibility* **by** *auto*
with *ftoh* **have** $(red\ ?G')^{**}\ f\ 0$ **by** *simp*
thus $f \in$ *pmdl ?G'* **by** (*simp add*: *red-rtranclp-0-in-pmdl*)
qed

lemma *GB-replace-red-stable-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
assumes *isGB*: *is-Groebner-basis G* **and** $p \in G$ **and** *q*: *red* ($G - \{p\}$) *p q*
shows *is-Groebner-basis* (*insert q* ($G - \{p\}$)) (**is** *is-Groebner-basis ?G'*)

proof −
from *assms*(*2*) **obtain** *n* **where** *1*: $G \subseteq$ *dgrad-p-set d n* **and** *2*: *?G'* $\subseteq$ *dgrad-p-set d n*
by (*rule replace-in-dgrad-p-set*)
from *isGB* **show** *?thesis* **unfolding** *GB-alt-2-dgrad-p-set*[*OF assms*(*1*) *1*] *GB-alt-2-dgrad-p-set*[*OF assms*(*1*) *2*]
proof (*intro ballI impI*)
fix *f*
assume *f1*: $f \in$ (*pmdl ?G'*) **and** $f \neq 0$
and *a1*: $\forall f \in$*pmdl G*. $f \neq 0 \longrightarrow$ *is-red G f*
have $q \in$ *pmdl G*
proof (*rule pmdl-closed-red*, *rule pmdl.span-mono*)
from *pmdl.span-superset* ‹$p \in G$› **show** $p \in$ *pmdl G* **..**
next
show $G - \{p\} \subseteq G$ **by** (*rule Diff-subset*)
qed (*rule q*)
from *f1 pmdl.replace-span*[*OF this*, *of p*] **have** $f \in$ *pmdl G* **..**
have *is-red G f* **by** (*rule a1*[*rule-format*], *fact+*)
show *is-red ?G' f* **by** (*rule replace-red-stable-is-red*, *fact+*)
qed
qed

lemma *GB-replace-red-stable-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
assumes *isGB*: *is-Groebner-basis G* **and** $p \in G$ **and** *ptoq*: *red* ($G - \{p\}$) *p q*
shows *pmdl* (*insert q* ($G - \{p\}$)) = *pmdl G* (**is** *pmdl ?G'* = -)
proof −
from ‹$p \in G$› *pmdl.span-superset* **have** $p \in$ *pmdl G* **..**
have $q \in$ *pmdl G*
by (*rule pmdl-closed-red*, *rule pmdl.span-mono*, *rule Diff-subset*, *rule* ‹$p \in$ *pmdl G*›, *rule ptoq*)
show *?thesis*
proof (*rule*, *rule pmdl.replace-span*, *fact*, *rule*)
fix *f*
assume $f \in$ *pmdl G*
note *assms*(*1*)
moreover from *assms*(*2*) **obtain** *n* **where** *?G'* $\subseteq$ *dgrad-p-set d n* **by** (*rule replace-in-dgrad-p-set*)
moreover have *is-Groebner-basis ?G'* **by** (*rule GB-replace-red-stable-GB-dgrad-p-set*, *fact+*)
ultimately have $\exists!$ *h*. (*red ?G'*)** *f h* $\wedge \neg$ *is-red ?G' h* **by** (*rule GB-implies-unique-nf-dgrad-p-set*)
then obtain *h* **where** *ftoh*: (*red ?G'*)** *f h* **and** *irredh*: $\neg$ *is-red ?G' h* **by** *auto*
have $\neg$ *is-red G h*
proof
assume *is-red G h*
have *is-red ?G' h* **by** (*rule replace-red-stable-is-red*, *fact+*)
with *irredh* **show** *False* **..**
qed
have $f - h \in$ *pmdl ?G'* **by** (*rule red-rtranclp-diff-in-pmdl*, *rule ftoh*)

```
    have f − h ∈ pmdl G by (rule, fact, rule pmdl.replace-span, fact)
    from pmdl.span-diff[OF this ‹f ∈ pmdl G›] have −h ∈ pmdl G by simp
    from pmdl.span-neg[OF this] have h ∈ pmdl G by simp
    with isGB ‹¬ is-red G h› have h = 0 using GB-imp-reducibility by auto
    with ftoh have (red ?G′)** f 0 by simp
    thus f ∈ pmdl ?G′ by (simp add: red-rtranclp-0-in-pmdl)
  qed
qed

lemma GB-replace-red-rtranclp-stable-GB-dgrad-p-set:
  assumes dickson-grading d and G ⊆ dgrad-p-set d m
  assumes isGB: is-Groebner-basis G and p ∈ G and ptoq: (red (G − {p}))** p
q
  shows is-Groebner-basis (insert q (G − {p}))
  using ptoq
proof (induct q rule: rtranclp-induct)
  case base
  from isGB ‹p ∈ G› show ?case by (simp add: insert-absorb)
next
  case (step y z)
  show ?case
  proof (cases y = p)
    case True
    from assms(1) assms(2) isGB ‹p ∈ G› show ?thesis
    proof (rule GB-replace-red-stable-GB-dgrad-p-set)
      from ‹red (G − {p}) y z› show red (G − {p}) p z unfolding True .
    qed
  next
    case False
    show ?thesis
      proof (cases y ∈ G)
        case True
        with ‹y ≠ p› have y ∈ G − {p} (is - ∈ ?G′) by blast
        hence insert y (G − {p}) = ?G′ by auto
        with step(3) have is-Groebner-basis ?G′ by simp
        from ‹y ∈ ?G′› pmdl.span-superset have y ∈ pmdl ?G′ ..
        have z ∈ pmdl ?G′ by (rule pmdl-closed-red, rule subset-refl, fact+)
        show is-Groebner-basis (insert z ?G′) by (rule GB-insert, fact+)
      next
        case False
        from assms(2) obtain n where insert y (G − {p}) ⊆ dgrad-p-set d n
            by (rule replace-in-dgrad-p-set)
       from assms(1) this step(3) have is-Groebner-basis (insert z (insert y (G −
{p}) − {y}))
        proof (rule GB-replace-red-stable-GB-dgrad-p-set)
          from ‹red (G − {p}) y z› False show red ((insert y (G − {p})) − {y})
y z by simp
        qed simp
        moreover from False have ... = (insert z (G − {p})) by simp
```

```
      ultimately show ?thesis by simp
    qed
  qed
qed

lemma GB-replace-red-rtranclp-stable-pmdl-dgrad-p-set:
  assumes dickson-grading d and G ⊆ dgrad-p-set d m
  assumes isGB: is-Groebner-basis G and p ∈ G and ptoq: (red (G − {p}))** p
q
  shows pmdl (insert q (G − {p})) = pmdl G
  using ptoq
proof (induct q rule: rtranclp-induct)
  case base
  from ‹p ∈ G› show ?case by (simp add: insert-absorb)
next
  case (step y z)
  show ?case
  proof (cases y = p)
    case True
    from assms(1) assms(2) isGB ‹p ∈ G› step(2) show ?thesis unfolding True
      by (rule GB-replace-red-stable-pmdl-dgrad-p-set)
  next
    case False
    have gb: is-Groebner-basis (insert y (G − {p}))
      by (rule GB-replace-red-rtranclp-stable-GB-dgrad-p-set, fact+)
    show ?thesis
    proof (cases y ∈ G)
      case True
      with ‹y ≠ p› have y ∈ G − {p} (is - ∈ ?G') by blast
      hence eq: insert y ?G' = ?G' by auto
      from ‹y ∈ ?G'› have y ∈ pmdl ?G' by (rule pmdl.span-base)
      have z ∈ pmdl ?G' by (rule pmdl-closed-red, rule subset-refl, fact+)
      hence pmdl (insert z ?G') = pmdl ?G' by (rule pmdl.span-insert-idI)
      also from step(3) have ... = pmdl G by (simp only: eq)
      finally show ?thesis .
    next
      case False
      from assms(2) obtain n where 1: insert y (G − {p}) ⊆ dgrad-p-set d n
        by (rule replace-in-dgrad-p-set)
     from False have pmdl (insert z (G − {p})) = pmdl (insert z (insert y (G −
{p}) − {y}))
        by auto
      also from assms(1) 1 gb have ... = pmdl (insert y (G − {p}))
      proof (rule GB-replace-red-stable-pmdl-dgrad-p-set)
        from step(2) False show red ((insert y (G − {p})) − {y}) y z by simp
      qed simp
      also have ... = pmdl G by fact
      finally show ?thesis .
    qed
```

qed
qed

lemmas *GB-replace-lt-adds-stable-GB-finite* =
GB-replace-lt-adds-stable-GB-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-replace-lt-adds-stable-pmdl-finite* =
GB-replace-lt-adds-stable-pmdl-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-replace-red-stable-GB-finite* =
GB-replace-red-stable-GB-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-replace-red-stable-pmdl-finite* =
GB-replace-red-stable-pmdl-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-replace-red-rtranclp-stable-GB-finite* =
GB-replace-red-rtranclp-stable-GB-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-replace-red-rtranclp-stable-pmdl-finite* =
GB-replace-red-rtranclp-stable-pmdl-dgrad-p-set[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

5.7 An Inconstructive Proof of the Existence of Finite Gröbner Bases

lemma *ex-finite-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
obtains *G* **where** *G* ⊆ *dgrad-p-set d m* **and** *finite G* **and** *is-Groebner-basis G* **and** *pmdl G* = *pmdl F*
proof −
define *S* **where** *S* = {*lt f* | *f*. *f* ∈ *pmdl F* ∧ *f* ∈ *dgrad-p-set d m* ∧ *f* ≠ *0*}
note *assms*(*1*)
moreover from - *assms*(*2*) **have** *finite* (*component-of-term* ‘ *S*)
proof (*rule finite-subset*)
have *component-of-term* ‘ *S* ⊆ *component-of-term* ‘ *Keys* (*pmdl F*)
by (*rule image-mono*, *rule*, *auto simp add*: *S-def intro*!: *in-KeysI lt-in-keys*)
thus *component-of-term* ‘ *S* ⊆ *component-of-term* ‘ *Keys F* **by** (*simp only*: *components-pmdl*)
qed
moreover have *pp-of-term* ‘ *S* ⊆ *dgrad-set d m*
proof
fix *s*
assume *s* ∈ *pp-of-term* ‘ *S*
then obtain *u* **where** *u* ∈ *S* **and** *s* = *pp-of-term u* **..**
from *this*(*1*) **obtain** *f* **where** *f* ∈ *pmdl F* ∧ *f* ∈ *dgrad-p-set d m* ∧ *f* ≠ *0* **and** *u*: *u* = *lt f*
unfolding *S-def* **by** *blast*
from *this*(*1*) **have** *f* ∈ *dgrad-p-set d m* **and** *f* ≠ *0* **by** *simp-all*
have *u* ∈ *keys f* **unfolding** *u* **by** (*rule lt-in-keys*, *fact*)
with ‹*f* ∈ *dgrad-p-set d m*› **have** *d* (*pp-of-term u*) ≤ *m* **unfolding** *u* **by** (*rule dgrad-p-setD*)
thus *s* ∈ *dgrad-set d m* **by** (*simp add*: ‹*s* = *pp-of-term u*› *dgrad-set-def*)

qed
ultimately obtain *T* **where** *finite T* **and** *T* ⊆ *S* **and** ∗: ⋀*s*. *s* ∈ *S* ⟹ (∃ *t*∈*T*. *t* $adds_t$ *s*)
by (*rule ex-finite-adds-term*, *blast*)
define *crit* **where** *crit* = (λ*t f*. *f* ∈ *pmdl F* ∧ *f* ∈ *dgrad-p-set d m* ∧ *f* ≠ *0* ∧ *t* = *lt f*)
have *ex-crit*: *t* ∈ *T* ⟹ (∃ *f*. *crit t f*) **for** *t*
proof −
assume *t* ∈ *T*
from *this* ‹*T* ⊆ *S*› **have** *t* ∈ *S* **..**
then obtain *f* **where** *f* ∈ *pmdl F* ∧ *f* ∈ *dgrad-p-set d m* ∧ *f* ≠ *0* **and** *t* = *lt f*
unfolding *S-def* **by** *blast*
thus ∃ *f*. *crit t f* **unfolding** *crit-def* **by** *blast*
qed
define *G* **where** *G* = (λ*t*. *SOME g*. *crit t g*) ‘ *T*
have *G*: *g* ∈ *G* ⟹ *g* ∈ *pmdl F* ∧ *g* ∈ *dgrad-p-set d m* ∧ *g* ≠ *0* **for** *g*
proof −
assume *g* ∈ *G*
then obtain *t* **where** *t* ∈ *T* **and** *g*: *g* = (*SOME h*. *crit t h*) **unfolding** *G-def* **..**
have *crit t g* **unfolding** *g* **by** (*rule someI-ex*, *rule ex-crit*, *fact*)
thus *g* ∈ *pmdl F* ∧ *g* ∈ *dgrad-p-set d m* ∧ *g* ≠ *0* **by** (*simp add*: *crit-def*)
qed
have ∗∗: *t* ∈ *T* ⟹ (∃ *g*∈*G*. *lt g* = *t*) **for** *t*
proof −
assume *t* ∈ *T*
define *g* **where** *g* = (*SOME h*. *crit t h*)
from ‹*t* ∈ *T*› **have** *g* ∈ *G* **unfolding** *g-def G-def* **by** *blast*
thus ∃ *g*∈*G*. *lt g* = *t*
proof
have *crit t g* **unfolding** *g-def* **by** (*rule someI-ex*, *rule ex-crit*, *fact*)
thus *lt g* = *t* **by** (*simp add*: *crit-def*)
qed
qed
have *adds*: *f* ∈ *pmdl F* ⟹ *f* ∈ *dgrad-p-set d m* ⟹ *f* ≠ *0* ⟹ (∃ *g*∈*G*. *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*) **for** *f*
proof −
assume *f* ∈ *pmdl F* **and** *f* ∈ *dgrad-p-set d m* **and** *f* ≠ *0*
hence *lt f* ∈ *S* **unfolding** *S-def* **by** *blast*
hence ∃ *t*∈*T*. *t* $adds_t$ (*lt f*) **by** (*rule* ∗)
then obtain *t* **where** *t* ∈ *T* **and** *t* $adds_t$ (*lt f*) **..**
from *this*(*1*) **have** ∃ *g*∈*G*. *lt g* = *t* **by** (*rule* ∗∗)
then obtain *g* **where** *g* ∈ *G* **and** *lt g* = *t* **..**
show ∃ *g*∈*G*. *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
proof (*intro bexI conjI*)
from *G*[*OF* ‹*g* ∈ *G*›] **show** *g* ≠ *0* **by** (*elim conjE*)
next
from ‹*t* $adds_t$ *lt f*› **show** *lt g* $adds_t$ *lt f* **by** (*simp only*: ‹*lt g* = *t*›)
qed *fact*

```
qed
have sub1: pmdl G ⊆ pmdl F
proof (rule pmdl.span-subset-spanI, rule)
  fix g
  assume g ∈ G
  from G[OF this] show g ∈ pmdl F ..
qed
have sub2: G ⊆ dgrad-p-set d m
proof
  fix g
  assume g ∈ G
  from G[OF this] show g ∈ dgrad-p-set d m by (elim conjE)
qed
show ?thesis
proof
  from ‹finite T› show finite G unfolding G-def ..
next
  from assms(1) sub2 adds show is-Groebner-basis G
  proof (rule isGB-I-adds-lt)
    fix f
    assume f ∈ pmdl G
    from this sub1 show f ∈ pmdl F ..
  qed
next
  show pmdl G = pmdl F
  proof
    show pmdl F ⊆ pmdl G
    proof (rule pmdl.span-subset-spanI, rule)
      fix f
      assume f ∈ F
      hence f ∈ pmdl F by (rule pmdl.span-base)
      from ‹f ∈ F› assms(3) have f ∈ dgrad-p-set d m ..
      with assms(1) sub2 sub1 - ‹f ∈ pmdl F› have (red G)** f 0
      proof (rule is-red-implies-0-red-dgrad-p-set)
        fix q
        assume q ∈ pmdl F and q ∈ dgrad-p-set d m and q ≠ 0
        hence (∃ g ∈ G. g ≠ 0 ∧ lt g adds_t lt q) by (rule adds)
        then obtain g where g ∈ G and g ≠ 0 and lt g adds_t lt q by blast
        thus is-red G q using ‹q ≠ 0› is-red-indI1 by blast
      qed
      thus f ∈ pmdl G by (rule red-rtranclp-0-in-pmdl)
    qed
  qed fact
next
  show G ⊆ dgrad-p-set d m
  proof
    fix g
    assume g ∈ G
    hence g ∈ pmdl F ∧ g ∈ dgrad-p-set d m ∧ g ≠ 0 by (rule G)
```

thus *g* ∈ *dgrad-p-set d m* **by** (*elim conjE*)
qed
qed
qed

The preceding lemma justifies the following definition.

definition *some-GB* :: ($'t \Rightarrow_0 'b$) *set* ⇒ ($'t \Rightarrow_0 'b$::*field*) *set*
where *some-GB F* = (*SOME G. finite G* ∧ *is-Groebner-basis G* ∧ *pmdl G* = *pmdl F*)

lemma *some-GB-props-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *finite* (*some-GB F*) ∧ *is-Groebner-basis* (*some-GB F*) ∧ *pmdl* (*some-GB F*) = *pmdl F*
proof −
from *assms* **obtain** *G* **where** *finite G* **and** *is-Groebner-basis G* **and** *pmdl G* = *pmdl F*
by (*rule ex-finite-GB-dgrad-p-set*)
hence *finite G* ∧ *is-Groebner-basis G* ∧ *pmdl G* = *pmdl F* **by** *simp*
thus *finite* (*some-GB F*) ∧ *is-Groebner-basis* (*some-GB F*) ∧ *pmdl* (*some-GB F*) = *pmdl F*
unfolding *some-GB-def* **by** (*rule someI*)
qed

lemma *finite-some-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *finite* (*some-GB F*)
using *some-GB-props-dgrad-p-set*[*OF assms*] **..**

lemma *some-GB-isGB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *is-Groebner-basis* (*some-GB F*)
using *some-GB-props-dgrad-p-set*[*OF assms*] **by** (*elim conjE*)

lemma *some-GB-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *pmdl* (*some-GB F*) = *pmdl F*
using *some-GB-props-dgrad-p-set*[*OF assms*] **by** (*elim conjE*)

lemma *finite-imp-finite-component-Keys*:
assumes *finite F*
shows *finite* (*component-of-term* ' *Keys F*)
by (*rule finite-imageI*, *rule finite-Keys*, *fact*)

lemma *finite-some-GB-finite*: *finite F* ⟹ *finite* (*some-GB F*)

by (*rule finite-some-GB-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*, *erule finite-imp-finite-component-Keys*, *erule dgrad-p-set-exhaust-expl*)

lemma *some-GB-isGB-finite*: *finite F* ⟹ *is-Groebner-basis* (*some-GB F*)
by (*rule some-GB-isGB-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*, *erule finite-imp-finite-component-Keys*, *erule dgrad-p-set-exhaust-expl*)

lemma *some-GB-pmdl-finite*: *finite F* ⟹ *pmdl* (*some-GB F*) = *pmdl F*
by (*rule some-GB-pmdl-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*, *erule finite-imp-finite-component-Keys*, *erule dgrad-p-set-exhaust-expl*)

Theory *Buchberger* implements an algorithm for effectively computing Gröbner bases.

5.8 Relation *red-supset*

The following relation is needed for proving the termination of Buchberger's algorithm (i. e. function *gb-schema-aux*).

definition *red-supset*::($'t \Rightarrow_0 'b$::*field*) *set* ⇒ ($'t \Rightarrow_0 'b$) *set* ⇒ *bool* (**infixl** ‹⊐p› *50*)
where *red-supset A B* ≡ (∃ *p*. *is-red A p* ∧ ¬ *is-red B p*) ∧ (∀ *p*. *is-red B p* ⟶ *is-red A p*)

lemma *red-supsetE*:
assumes *A* ⊐*p* *B*
obtains *p* **where** *is-red A p* **and** ¬ *is-red B p*
proof −
from *assms* **have** ∃ *p*. *is-red A p* ∧ ¬ *is-red B p* **by** (*simp add*: *red-supset-def*)
from *this* **obtain** *p* **where** *is-red A p* **and** ¬ *is-red B p* **by** *auto*
thus *?thesis* **..**
qed

lemma *red-supsetD*:
assumes *a1*: *A* ⊐*p* *B* **and** *a2*: *is-red B p*
shows *is-red A p*
proof −
from *assms* **have** ∀ *p*. *is-red B p* ⟶ *is-red A p* **by** (*simp add*: *red-supset-def*)
hence *is-red B p* ⟶ *is-red A p* **..**
from *a2 this* **show** *?thesis* **by** *simp*
qed

lemma *red-supsetI* [*intro*]:
assumes ⋀*q*. *is-red B q* ⟹ *is-red A q* **and** *is-red A p* **and** ¬ *is-red B p*
shows *A* ⊐*p* *B*
unfolding *red-supset-def* **using** *assms* **by** *auto*

lemma *red-supset-insertI*:
assumes *x* ≠ *0* **and** ¬ *is-red A x*

shows (*insert x A*) ⊐*p* *A*
proof
  fix *q*
  assume *is-red A q*
  thus *is-red* (*insert x A*) *q* **unfolding** *is-red-alt*
  proof
    fix *a*
    assume *red A q a*
    from *red-unionI2*[*OF this*, *of* {*x*}] **have** *red* (*insert x A*) *q a* **by** *simp*
    show ∃ *qa*. *red* (*insert x A*) *q qa*
    proof
      show *red* (*insert x A*) *q a* **by** *fact*
    qed
  qed
next
  show *is-red* (*insert x A*) *x* **unfolding** *is-red-alt*
  proof
    from *red-unionI1*[*OF red-self*[*OF* ‹$x \neq 0$›], *of A*] **show** *red* (*insert x A*) *x 0*
by *simp*
  qed
next
  show ¬ *is-red A x* **by** *fact*
qed

lemma *red-supset-transitive*:
  assumes *A* ⊐*p* *B* **and** *B* ⊐*p* *C*
  shows *A* ⊐*p* *C*
proof −
  from *assms*(*2*) **obtain** *p* **where** *is-red B p* **and** ¬ *is-red C p* **by** (*rule red-supsetE*)
  show *?thesis*
  proof
    fix *q*
    assume *is-red C q*
    with *assms*(*2*) **have** *is-red B q* **by** (*rule red-supsetD*)
    with *assms*(*1*) **show** *is-red A q* **by** (*rule red-supsetD*)
  next
    from *assms*(*1*) ‹*is-red B p*› **show** *is-red A p* **by** (*rule red-supsetD*)
  qed *fact*
qed

lemma *red-supset-wf-on*:
  assumes *dickson-grading d* **and** *finite K*
  shows *wfp-on* (⊐*p*) (*Pow* (*dgrad-p-set d m*) ∩ {*F*. *component-of-term* ‘ *Keys F*
⊆ *K*})
proof (*rule wfp-onI-chain*, *rule*, *erule exE*)
  let *?A* = *dgrad-p-set d m*
  fix *f*::*nat* ⇒ (($'t \Rightarrow_0 'b$) *set*)
  assume ∀ *i*. *f i* ∈ *Pow ?A* ∩ {*F*. *component-of-term* ‘ *Keys F* ⊆ *K*} ∧ *f* (*Suc i*)
⊐*p* *f i*

```
hence a1-subset: f i ⊆ ?A and comp-sub: component-of-term ‘ Keys (f i) ⊆ K
  and a1: f (Suc i) ⊐p f i for i by simp-all

have a1-trans: i < j ⟹ f j ⊐p f i for i j
proof −
  assume i < j
  thus f j ⊐p f i
  proof (induct j)
    case 0
    thus ?case by simp
  next
    case (Suc j)
    from Suc(2) have i = j ∨ i < j by auto
    thus ?case
    proof
      assume i = j
      show ?thesis unfolding ‹i = j› by (fact a1)
    next
      assume i < j
      from a1 have f (Suc j) ⊐p f j .
      also from ‹i < j› have ... ⊐p f i by (rule Suc(1))
      finally(red-supset-transitive) show ?thesis .
    qed
  qed
qed

have a2: ∃ p ∈ f (Suc i). ∃ q. is-red {p} q ∧ ¬ is-red (f i) q for i
proof −
  from a1 have f (Suc i) ⊐p f i .
  then obtain q where red: is-red (f (Suc i)) q and irred: ¬ is-red (f i) q
    by (rule red-supsetE)
 from red obtain p where p ∈ f (Suc i) and is-red {p} q by (rule is-red-singletonI)
  show ∃ p∈f (Suc i). ∃ q. is-red {p} q ∧ ¬ is-red (f i) q
  proof
    show ∃ q. is-red {p} q ∧ ¬ is-red (f i) q
    proof (intro exI, intro conjI)
      show is-red {p} q by fact
    qed (fact)
  next
    show p ∈ f (Suc i) by fact
  qed
qed

let ?P = λi p. p ∈ (f (Suc i)) ∧ (∃ q. is-red {p} q ∧ ¬ is-red (f i) q)
define g where g ≡ λi::nat. (SOME p. ?P i p)
have a3: ?P i (g i) for i
proof −
 from a2[of i] obtain gi where gi ∈ f (Suc i) and ∃ q. is-red {gi} q ∧ ¬ is-red
(f i) q ..
```

show *?thesis* **unfolding** *g-def* **by** (*rule someI*[*of - gi*], *intro conjI*, *fact+*)
qed

have *a4*: $i < j \Longrightarrow \neg$ *lt* (*g i*) $adds_t$ (*lt* (*g j*)) **for** *i j*
proof
assume $i < j$ **and** *adds*: *lt* (*g i*) $adds_t$ *lt* (*g j*)
from *a3* **have** ∃ *q*. *is-red* {*g j*} *q* ∧ ¬ *is-red* (*f j*) *q* **..**
then obtain *q* **where** *redj*: *is-red* {*g j*} *q* **and** ¬ *is-red* (*f j*) *q* **by** *auto*
have ∗: ¬ *is-red* (*f* (*Suc i*)) *q*
proof −
from ‹*i* < *j*› **have** *i* + *1* < *j* ∨ *i* + *1* = *j* **by** *auto*
thus *?thesis*
proof
assume *i* + *1* < *j*
from *red-supsetD*[*OF a1-trans*[*rule-format*, *OF this*], *of q*] ‹¬ *is-red* (*f j*) *q*›
show *?thesis* **by** *auto*
next
assume *i* + *1* = *j*
thus *?thesis* **using** ‹¬ *is-red* (*f j*) *q*› **by** *simp*
qed
qed
from *a3* **have** *g i* ∈ *f* (*i* + *1*) **and** *redi*: ∃ *q*. *is-red* {*g i*} *q* ∧ ¬ *is-red* (*f i*) *q* **by** *simp-all*
have ¬ *is-red* {*g i*} *q*
proof
assume *is-red* {*g i*} *q*
from *is-red-singletonD*[*OF this* ‹*g i* ∈ *f* (*i* + *1*)›] ∗ **show** *False* **by** *simp*
qed
have *g i* ≠ *0*
proof −
from *redi* **obtain** *q0* **where** *is-red* {*g i*} *q0* **by** *auto*
from *is-red-singleton-not-0*[*OF this*] **show** *?thesis* **.**
qed
from ‹¬ *is-red* {*g i*} *q*› *is-red-singleton-trans*[*OF redj adds* ‹*g i* ≠ *0*›] **show** *False* **by** *simp*
qed

from - *assms*(*2*) **have** *a5*: *finite* (*component-of-term* ‘ *range* (*lt* ∘ *g*))
proof (*rule finite-subset*)
show *component-of-term* ‘ *range* (*lt* ∘ *g*) ⊆ *K*
proof (*rule*, *elim imageE*, *simp*)
fix *i*
from *a3* **have** *g i* ∈ *f* (*Suc i*) **and** ∃ *q*. *is-red* {*g i*} *q* ∧ ¬ *is-red* (*f i*) *q* **by** *simp-all*
from *this*(*2*) **obtain** *q* **where** *is-red* {*g i*} *q* **by** *auto*
hence *g i* ≠ *0* **by** (*rule is-red-singleton-not-0*)
hence *lt* (*g i*) ∈ *keys* (*g i*) **by** (*rule lt-in-keys*)
hence *component-of-term* (*lt* (*g i*)) ∈ *component-of-term* ‘ *keys* (*g i*) **by** *simp*
also have ... ⊆ *component-of-term* ‘ *Keys* (*f* (*Suc i*))

by (*rule image-mono*, *rule keys-subset-Keys*, *fact*)
also have ... ⊆ *K* **by** (*fact comp-sub*)
finally show *component-of-term* (*lt* (*g i*)) ∈ *K* **.**
qed
qed

have *a6*: *pp-of-term* ' *range* (*lt* ∘ *g*) ⊆ *dgrad-set d m*
proof (*rule*, *elim imageE*, *simp*)
fix *i*
from *a3* **have** *g i* ∈ *f* (*Suc i*) **and** ∃ *q*. *is-red* {*g i*} *q* ∧ ¬ *is-red* (*f i*) *q* **by** *simp-all*
from *this*(*2*) **obtain** *q* **where** *is-red* {*g i*} *q* **by** *auto*
hence *g i* ≠ *0* **by** (*rule is-red-singleton-not-0*)
from *a1-subset* ‹*g i* ∈ *f* (*Suc i*)› **have** *g i* ∈ *?A* **..**
from *this* ‹*g i* ≠ *0*› **have** *d* (*lp* (*g i*)) ≤ *m* **by** (*rule dgrad-p-setD-lp*)
thus *lp* (*g i*) ∈ *dgrad-set d m* **by** (*rule dgrad-setI*)
qed

from *assms*(*1*) *a5 a6* **obtain** *i j* **where** *i* < *j* **and** (*lt* ∘ *g*) *i* $adds_t$ (*lt* ∘ *g*) *j* **by** (*rule Dickson-termE*)
from *this a4*[*OF* ‹*i* < *j*›] **show** *False* **by** *simp*
qed

end

lemma *in-lex-prod-alt*:
(*x*, *y*) ∈ *r* <∗*lex*∗> *s* ⟷ (((*fst x*), (*fst y*)) ∈ *r* ∨ (*fst x* = *fst y* ∧ ((*snd x*), (*snd y*)) ∈ *s*))
by (*metis in-lex-prod prod.collapse prod.inject surj-pair*)

5.9 Context *od-term*

context *od-term*
begin

lemmas *red-wf* = *red-wf-dgrad-p-set*[*OF dickson-grading-zero subset-dgrad-p-set-zero*]
lemmas *Buchberger-criterion* = *Buchberger-criterion-dgrad-p-set*[*OF dickson-grading-zero subset-dgrad-p-set-zero*]

end

end

6 A General Algorithm Schema for Computing Gröbner Bases

theory *Algorithm-Schema*
imports *General Groebner-Bases*

begin

This theory formalizes a general algorithm schema for computing Gröbner bases, generalizing Buchberger's original critical-pair/completion algorithm. The algorithm schema depends on several functional parameters that can be instantiated by a variety of concrete functions. Possible instances yield Buchberger's algorithm, Faugère's F4 algorithm, and (as far as we can tell) even his F5 algorithm.

6.1 *processed*

definition *minus-pairs* (**infixl** ‹$-_p$› *65*) **where** *minus-pairs* A B = $A - (B \cup$ *prod.swap* ‘ $B)$
definition *Int-pairs* (**infixl** ‹$\cap_p$› *65*) **where** *Int-pairs* A $B = A \cap (B \cup$ *prod.swap* ‘ $B)$
definition *in-pair* (**infix** ‹$\in_p$› *50*) **where** *in-pair* p $A \longleftrightarrow (p \in A \cup$ *prod.swap* ‘ $A)$
definition *subset-pairs* (**infix** ‹$\subseteq_p$› *50*) **where** *subset-pairs* A $B \longleftrightarrow (\forall x.\ x \in_p A \longrightarrow x \in_p B)$
abbreviation *not-in-pair* (**infix** ‹$\notin_p$› *50*) **where** *not-in-pair* p $A \equiv \neg\ p \in_p A$

lemma *in-pair-alt*: $p \in_p A \longleftrightarrow (p \in A \vee$ *prod.swap* $p \in A)$
by (*metis* (*mono-tags*, *lifting*) *UnCI UnE image-iff in-pair-def prod.collapse swap-simp*)

lemma *in-pair-iff*: $(a, b) \in_p A \longleftrightarrow ((a, b) \in A \vee (b, a) \in A)$
by (*simp add*: *in-pair-alt*)

lemma *in-pair-minus-pairs* [*simp*]: $p \in_p A -_p B \longleftrightarrow (p \in_p A \wedge p \notin_p B)$
by (*metis Diff-iff in-pair-def in-pair-iff minus-pairs-def prod.collapse*)

lemma *in-minus-pairs* [*simp*]: $p \in A -_p B \longleftrightarrow (p \in A \wedge p \notin_p B)$
by (*metis Diff-iff in-pair-def minus-pairs-def*)

lemma *in-pair-Int-pairs* [*simp*]: $p \in_p A \cap_p B \longleftrightarrow (p \in_p A \wedge p \in_p B)$
by (*metis* (*no-types*, *opaque-lifting*) *Int-iff Int-pairs-def in-pair-alt in-pair-def old.prod.exhaust swap-simp*)

lemma *in-pair-Un* [*simp*]: $p \in_p A \cup B \longleftrightarrow (p \in_p A \vee p \in_p B)$
by (*metis* (*mono-tags*, *lifting*) *UnE UnI1 UnI2 image-Un in-pair-def*)

lemma *in-pair-trans* [*trans*]:
assumes $p \in_p A$ **and** $A \subseteq B$
shows $p \in_p B$
using *assms* **by** (*auto simp*: *in-pair-def*)

lemma *in-pair-same* [*simp*]: $p \in_p A \times A \longleftrightarrow p \in A \times A$
by (*auto simp*: *in-pair-def*)

lemma *subset-pairsI* [*intro*]:
  assumes $\bigwedge x.\ x \in_p A \Longrightarrow x \in_p B$
  shows $A \subseteq_p B$
  unfolding *subset-pairs-def* **using** *assms* **by** *blast*

lemma *subset-pairsD* [*trans*]:
  assumes $x \in_p A$ **and** $A \subseteq_p B$
  shows $x \in_p B$
  using *assms* **unfolding** *subset-pairs-def* **by** *blast*

definition *processed* :: $('a \times 'a) \Rightarrow 'a\ list \Rightarrow ('a \times 'a)\ list \Rightarrow bool$
  where $processed\ p\ xs\ ps \longleftrightarrow p \in set\ xs \times set\ xs \wedge p \notin_p set\ ps$

lemma *processed-alt*:
  $processed\ (a,\ b)\ xs\ ps \longleftrightarrow ((a \in set\ xs) \wedge (b \in set\ xs) \wedge (a,\ b) \notin_p set\ ps)$
  unfolding *processed-def* **by** *auto*

lemma *processedI*:
  assumes $a \in set\ xs$ **and** $b \in set\ xs$ **and** $(a,\ b) \notin_p set\ ps$
  shows $processed\ (a,\ b)\ xs\ ps$
  unfolding *processed-alt* **using** *assms* **by** *simp*

lemma *processedD1*:
  assumes $processed\ (a,\ b)\ xs\ ps$
  shows $a \in set\ xs$
  using *assms* **by** (*simp add*: *processed-alt*)

lemma *processedD2*:
  assumes $processed\ (a,\ b)\ xs\ ps$
  shows $b \in set\ xs$
  using *assms* **by** (*simp add*: *processed-alt*)

lemma *processedD3*:
  assumes $processed\ (a,\ b)\ xs\ ps$
  shows $(a,\ b) \notin_p set\ ps$
  using *assms* **by** (*simp add*: *processed-alt*)

lemma *processed-Nil*: $processed\ (a,\ b)\ xs\ [] \longleftrightarrow (a \in set\ xs \wedge b \in set\ xs)$
  by (*simp add*: *processed-alt in-pair-iff*)

lemma *processed-Cons*:
  assumes $processed\ (a,\ b)\ xs\ ps$
    and *a1*: $a = p \Longrightarrow b = q \Longrightarrow thesis$
    and *a2*: $a = q \Longrightarrow b = p \Longrightarrow thesis$
    and *a3*: $processed\ (a,\ b)\ xs\ ((p,\ q)\ \#\ ps) \Longrightarrow thesis$
  shows *thesis*
proof −
  from *assms*(*1*) **have** $a \in set\ xs$ **and** $b \in set\ xs$ **and** $(a,\ b) \notin_p set\ ps$
    by (*simp-all add*: *processed-alt*)

```
  show ?thesis
  proof (cases (a, b) = (p, q))
    case True
    hence a = p and b = q by simp-all
    thus ?thesis by (rule a1)
  next
    case False
     with ‹(a, b) ∉p set ps› have *: (a, b) ∉ set ((p, q) # ps) by (auto simp:
in-pair-iff)
    show ?thesis
    proof (cases (b, a) = (p, q))
      case True
      hence a = q and b = p by simp-all
      thus ?thesis by (rule a2)
    next
      case False
        with ‹(a, b) ∉p set ps› have (b, a) ∉ set ((p, q) # ps) by (auto simp:
in-pair-iff)
      with * have (a, b) ∉p set ((p, q) # ps) by (simp add: in-pair-iff)
      with ‹a ∈ set xs› ‹b ∈ set xs› have processed (a, b) xs ((p, q) # ps)
        by (rule processedI)
      thus ?thesis by (rule a3)
    qed
  qed
qed

lemma processed-minus:
  assumes processed (a, b) xs (ps −− qs)
    and a1: (a, b) ∈p set qs ⟹ thesis
    and a2: processed (a, b) xs ps ⟹ thesis
  shows thesis
proof −
  from assms(1) have a ∈ set xs and b ∈ set xs and (a, b) ∉p set (ps −− qs)
    by (simp-all add: processed-alt)
  show ?thesis
  proof (cases (a, b) ∈p set qs)
    case True
    thus ?thesis by (rule a1)
  next
    case False
    with ‹(a, b) ∉p set (ps −− qs)› have (a, b) ∉p set ps
      by (auto simp: set-diff-list in-pair-iff)
    with ‹a ∈ set xs› ‹b ∈ set xs› have processed (a, b) xs ps
      by (rule processedI)
    thus ?thesis by (rule a2)
  qed
qed
```

6.2 Algorithm Schema

6.2.1 *const-lt-component*

context *ordered-term*
begin

definition *const-lt-component* :: $('t \Rightarrow_0 'b{::}zero) \Rightarrow 'k$ *option*
  where *const-lt-component p* =
    (*let v* = *lt p in if pp-of-term v* = *0 then Some* (*component-of-term v*) *else None*)

lemma *const-lt-component-SomeI*:
  assumes *lp p* = *0* **and** *component-of-term* (*lt p*) = *cmp*
  shows *const-lt-component p* = *Some cmp*
  using *assms* **by** (*simp add*: *const-lt-component-def*)

lemma *const-lt-component-SomeD1*:
  assumes *const-lt-component p* = *Some cmp*
  shows *lp p* = *0*
  using *assms* **by** (*simp add*: *const-lt-component-def Let-def split*: *if-split-asm*)

lemma *const-lt-component-SomeD2*:
  assumes *const-lt-component p* = *Some cmp*
  shows *component-of-term* (*lt p*) = *cmp*
  using *assms* **by** (*simp add*: *const-lt-component-def Let-def split*: *if-split-asm*)

lemma *const-lt-component-subset*:
  const-lt-component ‘ (*B* − {*0*}) − {*None*} ⊆ *Some* ‘ *component-of-term* ‘ *Keys B*
proof
  fix *k*
  assume *k* ∈ *const-lt-component* ‘ (*B* − {*0*}) − {*None*}
  hence *k* ∈ *const-lt-component* ‘ (*B* − {*0*}) **and** *k* ≠ *None* **by** *simp-all*
  from *this*(*1*) **obtain** *p* **where** *p* ∈ *B* − {*0*} **and** *k* = *const-lt-component p* **..**
  moreover from ‹*k* ≠ *None*› **obtain** *k′* **where** *k* = *Some k′* **by** *blast*
  ultimately have *const-lt-component p* = *Some k′* **and** *p* ∈ *B* **and** *p* ≠ *0* **by** *simp-all*
  from *this*(*1*) **have** *component-of-term* (*lt p*) = *k′* **by** (*rule const-lt-component-SomeD2*)
  moreover have *lt p* ∈ *Keys B* **by** (*rule in-KeysI*, *rule lt-in-keys*, *fact+*)
  ultimately have *k′* ∈ *component-of-term* ‘ *Keys B* **by** *fastforce*
  thus *k* ∈ *Some* ‘ *component-of-term* ‘ *Keys B* **by** (*simp add*: ‹*k* = *Some k′*›)
qed

corollary *card-const-lt-component-le*:
  assumes *finite B*
  shows *card* (*const-lt-component* ‘ (*B* − {*0*}) − {*None*}) ≤ *card* (*component-of-term* ‘ *Keys B*)
proof (*rule surj-card-le*)
  show *finite* (*component-of-term* ‘ *Keys B*)

by (*intro finite-imageI finite-Keys*, *fact*)
next
show *const-lt-component* ‘ ($B - \{0\}$) − {*None*} ⊆ *Some* ‘ *component-of-term* ‘ *Keys B*
by (*fact const-lt-component-subset*)
qed

end

6.2.2 Type synonyms

type-synonym ('a, 'b, 'c) *pdata'* = ('a ⇒₀ 'b) × 'c
type-synonym ('a, 'b, 'c) *pdata* = ('a ⇒₀ 'b) × *nat* × 'c
type-synonym ('a, 'b, 'c) *pdata-pair* = ('a, 'b, 'c) *pdata* × ('a, 'b, 'c) *pdata*
type-synonym ('a, 'b, 'c, 'd) *selT* = ('a, 'b, 'c) *pdata list* ⇒ ('a, 'b, 'c) *pdata list* ⇒
('a, 'b, 'c) *pdata-pair list* ⇒ *nat* × 'd ⇒ ('a, 'b, 'c) *pdata-pair list*
type-synonym ('a, 'b, 'c, 'd) *complT* = ('a, 'b, 'c) *pdata list* ⇒ ('a, 'b, 'c) *pdata list* ⇒
('a, 'b, 'c) *pdata-pair list* ⇒ ('a, 'b, 'c) *pdata-pair list* ⇒
nat × 'd ⇒ (('a, 'b, 'c) *pdata' list* × 'd)
type-synonym ('a, 'b, 'c, 'd) *apT* = ('a, 'b, 'c) *pdata list* ⇒ ('a, 'b, 'c) *pdata list* ⇒
('a, 'b, 'c) *pdata-pair list* ⇒ ('a, 'b, 'c) *pdata list* ⇒ *nat* × 'd ⇒
('a, 'b, 'c) *pdata-pair list*
type-synonym ('a, 'b, 'c, 'd) *abT* = ('a, 'b, 'c) *pdata list* ⇒ ('a, 'b, 'c) *pdata list* ⇒
('a, 'b, 'c) *pdata list* ⇒ *nat* × 'd ⇒ ('a, 'b, 'c) *pdata list*

6.2.3 Specification of the *selector* parameter

definition *sel-spec* :: ('a, 'b, 'c, 'd) *selT* ⇒ *bool*
where *sel-spec sel* ⟷
(∀ *gs bs ps data*. *ps* ≠ [] ⟶ (*sel gs bs ps data* ≠ [] ∧ *set* (*sel gs bs ps data*) ⊆ *set ps*))

lemma *sel-specI*:
assumes ⋀*gs bs ps data*. *ps* ≠ [] ⟹ (*sel gs bs ps data* ≠ [] ∧ *set* (*sel gs bs ps data*) ⊆ *set ps*)
shows *sel-spec sel*
unfolding *sel-spec-def* **using** *assms* **by** *blast*

lemma *sel-specD1*:
assumes *sel-spec sel* **and** *ps* ≠ []
shows *sel gs bs ps data* ≠ []
using *assms* **unfolding** *sel-spec-def* **by** *blast*

lemma *sel-specD2*:
assumes *sel-spec sel* **and** $ps \neq []$
shows *set* (*sel gs bs ps data*) $\subseteq$ *set ps*
using *assms* **unfolding** *sel-spec-def* **by** *blast*

6.2.4 Specification of the *add-basis* parameter

definition *ab-spec* :: ('*a*, '*b*, '*c*, '*d*) *abT* $\Rightarrow$ *bool*
where *ab-spec ab* $\longleftrightarrow$
($\forall$ *gs bs ns data*. $ns \neq [] \longrightarrow$ *set* (*ab gs bs ns data*) = *set bs* $\cup$ *set ns*) $\wedge$
($\forall$ *gs bs data*. *ab gs bs* [] *data* = *bs*)

lemma *ab-specI*:
assumes $\bigwedge$*gs bs ns data*. $ns \neq [] \Longrightarrow$ *set* (*ab gs bs ns data*) = *set bs* $\cup$ *set ns*
and $\bigwedge$*gs bs data*. *ab gs bs* [] *data* = *bs*
shows *ab-spec ab*
unfolding *ab-spec-def* **using** *assms* **by** *blast*

lemma *ab-specD1*:
assumes *ab-spec ab*
shows *set* (*ab gs bs ns data*) = *set bs* $\cup$ *set ns*
using *assms* **unfolding** *ab-spec-def* **by** (*metis empty-set sup-bot.right-neutral*)

lemma *ab-specD2*:
assumes *ab-spec ab*
shows *ab gs bs* [] *data* = *bs*
using *assms* **unfolding** *ab-spec-def* **by** *blast*

6.2.5 Specification of the *add-pairs* parameter

definition *unique-idx* :: ('*t*, '*b*, '*c*) *pdata list* $\Rightarrow$ (*nat* $\times$ '*d*) $\Rightarrow$ *bool*
where *unique-idx bs data* $\longleftrightarrow$
($\forall$ *f*$\in$*set bs*. $\forall$ *g*$\in$*set bs*. *fst* (*snd f*) = *fst* (*snd g*) $\longrightarrow$ *f* = *g*) $\wedge$
($\forall$ *f*$\in$*set bs*. *fst* (*snd f*) < *fst data*)

lemma *unique-idxI*:
assumes $\bigwedge$*f g*. *f* $\in$ *set bs* $\Longrightarrow$ *g* $\in$ *set bs* $\Longrightarrow$ *fst* (*snd f*) = *fst* (*snd g*) $\Longrightarrow$ *f* = *g*
and $\bigwedge$*f*. *f* $\in$ *set bs* $\Longrightarrow$ *fst* (*snd f*) < *fst data*
shows *unique-idx bs data*
unfolding *unique-idx-def* **using** *assms* **by** *blast*

lemma *unique-idxD1*:
assumes *unique-idx bs data* **and** *f* $\in$ *set bs* **and** *g* $\in$ *set bs* **and** *fst* (*snd f*) = *fst* (*snd g*)
shows *f* = *g*
using *assms* **unfolding** *unique-idx-def* **by** *blast*

lemma *unique-idxD2*:
assumes *unique-idx bs data* **and** *f* $\in$ *set bs*
shows *fst* (*snd f*) < *fst data*

using *assms* **unfolding** *unique-idx-def* **by** *blast*

lemma *unique-idx-Nil*: *unique-idx* [] *data*
by (*simp add*: *unique-idx-def*)

lemma *unique-idx-subset*:
assumes *unique-idx bs data* **and** *set bs'* ⊆ *set bs*
shows *unique-idx bs' data*
proof (*rule unique-idxI*)
fix *f g*
assume *f* ∈ *set bs'* **and** *g* ∈ *set bs'*
with *assms* **have** *unique-idx bs data* **and** *f* ∈ *set bs* **and** *g* ∈ *set bs* **by** *auto*
moreover assume *fst* (*snd f*) = *fst* (*snd g*)
ultimately show *f* = *g* **by** (*rule unique-idxD1*)
next
fix *f*
assume *f* ∈ *set bs'*
with *assms*(*2*) **have** *f* ∈ *set bs* **by** *auto*
with *assms*(*1*) **show** *fst* (*snd f*) < *fst data* **by** (*rule unique-idxD2*)
qed

context *gd-term*
begin

definition *ap-spec* :: (*'t*, *'b::field*, *'c*, *'d*) *apT* ⇒ *bool*
where *ap-spec ap* ⟷ (∀ *gs bs ps hs data*.
set (*ap gs bs ps hs data*) ⊆ *set ps* ∪ (*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)) ∧
(∀ *B d m*. ∀ *h*∈*set hs*. ∀ *g*∈*set gs* ∪ *set bs* ∪ *set hs*. *dickson-grading d* ⟶
set gs ∪ *set bs* ∪ *set hs* ⊆ *B* ⟶ *fst* ‘ *B* ⊆ *dgrad-p-set d m* ⟶
set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟶ *unique-idx* (*gs* @ *bs* @ *hs*) *data* ⟶
is-Groebner-basis (*fst* ‘ *set gs*) ⟶ *h* ≠ *g* ⟶ *fst h* ≠ *0* ⟶ *fst g* ≠ *0* ⟶
(∀ *a b*. (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟶ *fst a* ≠ *0* ⟶ *fst b* ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst a*) (*fst b*)) ⟶
(∀ *a b*. *a* ∈ *set gs* ∪ *set bs* ⟶ *b* ∈ *set gs* ∪ *set bs* ⟶ *fst a* ≠ *0* ⟶ *fst b* ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst a*) (*fst b*)) ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst h*) (*fst g*)) ∧
(∀ *B d m*. ∀ *h g*. *dickson-grading d* ⟶
set gs ∪ *set bs* ∪ *set hs* ⊆ *B* ⟶ *fst* ‘ *B* ⊆ *dgrad-p-set d m* ⟶
set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟶ (*set gs* ∪ *set bs*) ∩ *set hs* = {} ⟶
unique-idx (*gs* @ *bs* @ *hs*) *data* ⟶ *is-Groebner-basis* (*fst* ‘ *set gs*) ⟶
h ≠ *g* ⟶ *fst h* ≠ *0* ⟶ *fst g* ≠ *0* ⟶
(*h*, *g*) ∈ *set ps* $-_p$ *set* (*ap gs bs ps hs data*) ⟶
(∀ *a b*. (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟶ (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⟶
fst a ≠ *0* ⟶ *fst b* ≠ *0* ⟶ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst a*) (*fst b*)) ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst h*) (*fst g*)))

Informally, *ap-spec ap* means that, for suitable arguments *gs*, *bs*, *ps* and *hs*, the value of *ap gs bs ps hs* is a list of pairs *ps'* such that for every element (*a*, *b*) missing in *ps'* there exists a set of pairs *C* by reference to which (*a*, *b*) can be discarded, i. e. as soon as all critical pairs of the elements in *C* can be connected below some set *B*, the same is true for the critical pair of (*a*, *b*).

lemma *ap-specI*:
  assumes $\bigwedge$*gs bs ps hs data. set* (*ap gs bs ps hs data*) ⊆ *set ps* ∪ (*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*))
  assumes $\bigwedge$*gs bs ps hs data B d m h g. dickson-grading d* ⟹
    set gs ∪ *set bs* ∪ *set hs* ⊆ *B* ⟹ *fst ‘ B* ⊆ *dgrad-p-set d m* ⟹
    h ∈ *set hs* ⟹ *g* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹
    set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟹ *unique-idx* (*gs* @ *bs* @ *hs*) *data* ⟹
    is-Groebner-basis (*fst ‘ set gs*) ⟹ *h* ≠ *g* ⟹ *fst h* ≠ *0* ⟹ *fst g* ≠ *0* ⟹
    ($\bigwedge$*a b.* (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟹ *fst a* ≠ *0* ⟹ *fst b* ≠ *0* ⟹
      crit-pair-cbelow-on d m (*fst ‘ B*) (*fst a*) (*fst b*)) ⟹
    ($\bigwedge$*a b. a* ∈ *set gs* ∪ *set bs* ⟹ *b* ∈ *set gs* ∪ *set bs* ⟹ *fst a* ≠ *0* ⟹ *fst b* ≠ *0* ⟹
      crit-pair-cbelow-on d m (*fst ‘ B*) (*fst a*) (*fst b*)) ⟹
    crit-pair-cbelow-on d m (*fst ‘ B*) (*fst h*) (*fst g*)
  assumes $\bigwedge$*gs bs ps hs data B d m h g. dickson-grading d* ⟹
    set gs ∪ *set bs* ∪ *set hs* ⊆ *B* ⟹ *fst ‘ B* ⊆ *dgrad-p-set d m* ⟹
    set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟹ (*set gs* ∪ *set bs*) ∩ *set hs* = {} ⟹
    unique-idx (*gs* @ *bs* @ *hs*) *data* ⟹ *is-Groebner-basis* (*fst ‘ set gs*) ⟹ *h* ≠ *g* ⟹
    fst h ≠ *0* ⟹ *fst g* ≠ *0* ⟹ (*h*, *g*) ∈ *set ps* $-_p$ *set* (*ap gs bs ps hs data*) ⟹
    ($\bigwedge$*a b.* (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟹ (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⟹
      fst a ≠ *0* ⟹ *fst b* ≠ *0* ⟹ *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst a*) (*fst b*)) ⟹
    crit-pair-cbelow-on d m (*fst ‘ B*) (*fst h*) (*fst g*)
  shows *ap-spec ap*
  unfolding *ap-spec-def*
  apply (*intro allI conjI impI*)
    subgoal by (*rule assms*(*1*))
    subgoal by (*intro ballI impI*, *rule assms*(*2*), *blast*+)
    subgoal by (*rule assms*(*3*), *blast*+)
  done

lemma *ap-specD1*:
  assumes *ap-spec ap*
  shows *set* (*ap gs bs ps hs data*) ⊆ *set ps* ∪ (*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*))
  using *assms* **unfolding** *ap-spec-def* **by** (*elim allE conjE*) (*assumption*)

lemma *ap-specD2*:
assumes *ap-spec ap* **and** *dickson-grading d* **and** *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B*
and *fst* ' *B* ⊆ *dgrad-p-set d m* **and** (*h*, *g*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *unique-idx* (*gs* @ *bs* @ *hs*) *data*
and *is-Groebner-basis* (*fst* ' *set gs*) **and** *h* ≠ *g* **and** *fst h* ≠ *0* **and** *fst g* ≠ *0*
and ⋀*a b*. (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟹ *fst a* ≠ *0* ⟹ *fst b* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst* ' *B*) (*fst a*) (*fst b*)
and ⋀*a b*. *a* ∈ *set gs* ∪ *set bs* ⟹ *b* ∈ *set gs* ∪ *set bs* ⟹ *fst a* ≠ *0* ⟹ *fst b* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst* ' *B*) (*fst a*) (*fst b*)
shows *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst h*) (*fst g*)
proof −
from *assms*(*5*) **have** (*h*, *g*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∨ (*g*, *h*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*simp only*: *in-pair-iff*)
thus *?thesis*
proof
assume (*h*, *g*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
hence *h* ∈ *set hs* **and** *g* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
from *assms*(*1*)[*unfolded ap-spec-def*, *rule-format*, *of gs bs ps hs data*] *assms*(*2*−*4*) *this assms* (*6*−)
show *?thesis* **by** *metis*
next
assume (*g*, *h*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
hence *g* ∈ *set hs* **and** *h* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
hence *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst g*) (*fst h*)
using *assms*(*1*)[*unfolded ap-spec-def*, *rule-format*, *of gs bs ps hs data*]
assms(*2*,*3*,*4*,*6*,*7*,*8*,*10*,*11*,*12*,*13*) *assms*(*9*)[*symmetric*]
by *metis*
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed

lemma *ap-specD3*:
assumes *ap-spec ap* **and** *dickson-grading d* **and** *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B*
and *fst* ' *B* ⊆ *dgrad-p-set d m* **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and (*set gs* ∪ *set bs*) ∩ *set hs* = {} **and** *unique-idx* (*gs* @ *bs* @ *hs*) *data*
and *is-Groebner-basis* (*fst* ' *set gs*) **and** *h* ≠ *g* **and** *fst h* ≠ *0* **and** *fst g* ≠ *0*
and (*h*, *g*) $\in_p$ *set ps* $-_p$ *set* (*ap gs bs ps hs data*)
and ⋀*a b*. *a* ∈ *set hs* ⟹ *b* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹ (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) ⟹
fst a ≠ *0* ⟹ *fst b* ≠ *0* ⟹ *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst a*) (*fst b*)
shows *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst h*) (*fst g*)
proof −
have ∗: *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst a*) (*fst b*)
if *1*: (*a*, *b*) $\in_p$ *set* (*ap gs bs ps hs data*) **and** *2*: (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
and *3*: *fst a* ≠ *0* **and** *4*: *fst b* ≠ *0* **for** *a b*

proof −
from *2* **have** $(a, b) \in set\ hs \times (set\ gs \cup set\ bs \cup set\ hs) \vee (b, a) \in set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$
by (*simp only*: *in-pair-iff*)
thus *?thesis*
proof
assume $(a, b) \in set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$
hence $a \in set\ hs$ **and** $b \in set\ gs \cup set\ bs \cup set\ hs$ **by** *simp-all*
thus *?thesis* **using** *1 3 4* **by** (*rule assms*(*13*))
next
assume $(b, a) \in set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$
hence $b \in set\ hs$ **and** $a \in set\ gs \cup set\ bs \cup set\ hs$ **by** *simp-all*
moreover from *1* **have** $(b, a) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data)$ **by** (*auto simp*: *in-pair-iff*)
ultimately have *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst b*) (*fst a*) **using** *4 3* **by** (*rule assms*(*13*))
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed
from *assms*(*12*) **have** $(h, g) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data) \vee$
$(g, h) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data)$ **by** (*simp only*: *in-pair-iff*)
thus *?thesis*
proof
assume $(h, g) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data)$
with *assms*(*1*)[*unfolded ap-spec-def*, *rule-format*, *of gs bs ps hs data*] *assms*(*2*−*11*)
show *?thesis* **using** *assms*(*10*) ∗ **by** *metis*
next
assume $(g, h) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data)$
with *assms*(*1*)[*unfolded ap-spec-def*, *rule-format*, *of gs bs ps hs data*] *assms*(*2*−*11*)
have *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst g*) (*fst h*) **using** *assms*(*10*) ∗ **by** *metis*
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed

lemma *ap-spec-Nil-subset*:
assumes *ap-spec ap*
shows $set\ (ap\ gs\ bs\ ps\ []\ data) \subseteq set\ ps$
using *ap-specD1*[*OF assms*] **by** *fastforce*

lemma *ap-spec-fst-subset*:
assumes *ap-spec ap*
shows $fst\ ‘\ set\ (ap\ gs\ bs\ ps\ hs\ data) \subseteq fst\ ‘\ set\ ps \cup set\ hs$
proof −
from *ap-specD1*[*OF assms*]
have $fst\ ‘\ set\ (ap\ gs\ bs\ ps\ hs\ data) \subseteq fst\ ‘\ (set\ ps \cup set\ hs \times (set\ gs \cup set\ bs \cup set\ hs))$
by (*rule image-mono*)

thus *?thesis* **by** *auto*
qed

lemma *ap-spec-snd-subset*:
assumes *ap-spec ap*
shows *snd ‘ set* (*ap gs bs ps hs data*) ⊆ *snd ‘ set ps* ∪ *set gs* ∪ *set bs* ∪ *set hs*
proof −
from *ap-specD1*[*OF assms*]
have *snd ‘ set* (*ap gs bs ps hs data*) ⊆ *snd ‘* (*set ps* ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*))
by (*rule image-mono*)
thus *?thesis* **by** *auto*
qed

lemma *ap-spec-inE*:
assumes *ap-spec ap* **and** (*p*, *q*) ∈ *set* (*ap gs bs ps hs data*)
assumes *1*: (*p*, *q*) ∈ *set ps* ⟹ *thesis*
assumes *2*: *p* ∈ *set hs* ⟹ *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹ *thesis*
shows *thesis*
proof −
from *assms*(*2*) *ap-specD1*[*OF assms*(*1*)] **have** (*p*, *q*) ∈ *set ps* ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **..**
thus *?thesis*
proof
assume (*p*, *q*) ∈ *set ps*
thus *?thesis* **by** (*rule 1*)
next
assume (*p*, *q*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
hence *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *blast+*
thus *?thesis* **by** (*rule 2*)
qed
qed

lemma *subset-Times-ap*:
assumes *ap-spec ap* **and** *ab-spec ab* **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
shows *set* (*ap gs bs* (*ps* −− *sps*) *hs data*) ⊆ *set* (*ab gs bs hs data*) × (*set gs* ∪ *set* (*ab gs bs hs data*))
proof
fix *p q*
assume (*p*, *q*) ∈ *set* (*ap gs bs* (*ps* −− *sps*) *hs data*)
with *assms*(*1*) **show** (*p*, *q*) ∈ *set* (*ab gs bs hs data*) × (*set gs* ∪ *set* (*ab gs bs hs data*))
proof (*rule ap-spec-inE*)
assume (*p*, *q*) ∈ *set* (*ps* −− *sps*)
hence (*p*, *q*) ∈ *set ps* **by** (*simp add*: *set-diff-list*)
from *this assms*(*3*) **have** (*p*, *q*) ∈ *set bs* × (*set gs* ∪ *set bs*) **..**
hence *p* ∈ *set bs* **and** *q* ∈ *set gs* ∪ *set bs* **by** *blast+*
thus *?thesis* **by** (*auto simp add*: *ab-specD1*[*OF assms*(*2*)])
next

assume *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs*
thus *?thesis* **by** (*simp add*: *ab-specD1*[*OF assms*(*2*)])
qed
qed

6.2.6 Function *args-to-set*

definition *args-to-set* :: (*'t*, *'b*::*field*, *'c*) *pdata list* × (*'t*, *'b*, *'c*) *pdata list* × (*'t*, *'b*, *'c*) *pdata-pair list* ⇒ (*'t* $\Rightarrow_0$ *'b*) *set*
where *args-to-set x* = *fst* ' (*set* (*fst x*) ∪ *set* (*fst* (*snd x*)) ∪ *fst* ' *set* (*snd* (*snd x*)) ∪ *snd* ' *set* (*snd* (*snd x*)))

lemma *args-to-set-alt*:
args-to-set (*gs*, *bs*, *ps*) = *fst* ' *set gs* ∪ *fst* ' *set bs* ∪ *fst* ' *fst* ' *set ps* ∪ *fst* ' *snd* ' *set ps*
by (*simp add*: *args-to-set-def image-Un*)

lemma *args-to-set-subset-Times*:
assumes *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
shows *args-to-set* (*gs*, *bs*, *ps*) = *fst* ' *set gs* ∪ *fst* ' *set bs*
unfolding *args-to-set-alt* **using** *assms* **by** *auto*

lemma *args-to-set-subset*:
assumes *ap-spec ap* **and** *ab-spec ab*
shows *args-to-set* (*gs*, *ab gs bs hs data*, *ap gs bs ps hs data*) ⊆
fst ' (*set gs* ∪ *set bs* ∪ *fst* ' *set ps* ∪ *snd* ' *set ps* ∪ *set hs*) (**is** *?l* ⊆ *fst* ' *?r*)
proof (*simp only*: *args-to-set-alt Un-subset-iff*, *intro conjI image-mono*)
show *set* (*ab gs bs hs data*) ⊆ *?r* **by** (*auto simp add*: *ab-specD1*[*OF assms*(*2*)])
next
from *assms*(*1*) **have** *fst* ' *set* (*ap gs bs ps hs data*) ⊆ *fst* ' *set ps* ∪ *set hs*
by (*rule ap-spec-fst-subset*)
thus *fst* ' *set* (*ap gs bs ps hs data*) ⊆ *?r* **by** *blast*
next
from *assms*(*1*) **have** *snd* ' *set* (*ap gs bs ps hs data*) ⊆ *snd* ' *set ps* ∪ *set gs* ∪ *set bs* ∪ *set hs*
by (*rule ap-spec-snd-subset*)
thus *snd* ' *set* (*ap gs bs ps hs data*) ⊆ *?r* **by** *blast*
qed *blast*

lemma *args-to-set-alt2*:
assumes *ap-spec ap* **and** *ab-spec ab* **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
shows *args-to-set* (*gs*, *ab gs bs hs data*, *ap gs bs* (*ps* −− *sps*) *hs data*) =
fst ' (*set gs* ∪ *set bs* ∪ *set hs*) (**is** *?l* = *fst* ' *?r*)
proof
from *assms*(*1*, *2*) **have** *?l* ⊆ *fst* ' (*set gs* ∪ *set bs* ∪ *fst* ' *set* (*ps* −− *sps*) ∪ *snd* ' *set* (*ps* −− *sps*) ∪ *set hs*)
by (*rule args-to-set-subset*)
also have ... ⊆ *fst* ' *?r*

proof (*rule image-mono*)
have *set gs* ∪ *set bs* ∪ *fst* ‘ *set* (*ps* −− *sps*) ∪ *snd* ‘ *set* (*ps* −− *sps*) ∪ *set hs* ⊆
set gs ∪ *set bs* ∪ *fst* ‘ *set ps* ∪ *snd* ‘ *set ps* ∪ *set hs* **by** (*auto simp*: *set-diff-list*)
also from *assms*(*3*) **have** ... ⊆ *?r* **by** *fastforce*
finally show *set gs* ∪ *set bs* ∪ *fst* ‘ *set* (*ps* −− *sps*) ∪ *snd* ‘ *set* (*ps* −− *sps*) ∪ *set hs* ⊆ *?r* **.**
qed
finally show *?l* ⊆ *fst* ‘ *?r* **.**
next
from *assms*(*2*) **have** *eq*: *set* (*ab gs bs hs data*) = *set bs* ∪ *set hs* **by** (*rule ab-specD1*)
have *fst* ‘ *?r* ⊆ *fst* ‘ *set gs* ∪ *fst* ‘ *set* (*ab gs bs hs data*) **unfolding** *eq* **using** *assms*(*3*)
by *fastforce*
also have ... ⊆ *?l* **unfolding** *args-to-set-alt* **by** *fastforce*
finally show *fst* ‘ *?r* ⊆ *?l* **.**
qed

lemma *args-to-set-subset1*:
assumes *set gs1* ⊆ *set gs2*
shows *args-to-set* (*gs1*, *bs*, *ps*) ⊆ *args-to-set* (*gs2*, *bs*, *ps*)
using *assms* **by** (*auto simp add*: *args-to-set-alt*)

lemma *args-to-set-subset2*:
assumes *set bs1* ⊆ *set bs2*
shows *args-to-set* (*gs*, *bs1*, *ps*) ⊆ *args-to-set* (*gs*, *bs2*, *ps*)
using *assms* **by** (*auto simp add*: *args-to-set-alt*)

lemma *args-to-set-subset3*:
assumes *set ps1* ⊆ *set ps2*
shows *args-to-set* (*gs*, *bs*, *ps1*) ⊆ *args-to-set* (*gs*, *bs*, *ps2*)
using *assms* **unfolding** *args-to-set-alt* **by** *blast*

6.2.7 Functions *count-const-lt-components*, *count-rem-comps* and *full-gb*

definition *rem-comps-spec* :: (*'t*, *'b*::*zero*, *'c*) *pdata list* ⇒ *nat* × *'d* ⇒ *bool*
where *rem-comps-spec bs data* ⟷ (*card* (*component-of-term* ‘ *Keys* (*fst* ‘ *set bs*)) =
fst data + *card* (*const-lt-component* ‘ (*fst* ‘ *set bs* − {*0*}) − {*None*}))

definition *count-const-lt-components* :: (*'t*, *'b*::*zero*, *'c*) *pdata' list* ⇒ *nat*
where *count-const-lt-components hs* = *length* (*remdups* (*filter* (λ*x*. *x* ≠ *None*) (*map* (*const-lt-component* ∘ *fst*) *hs*)))

definition *count-rem-components* :: (*'t*, *'b*::*zero*, *'c*) *pdata' list* ⇒ *nat*
where *count-rem-components bs* = *length* (*remdups* (*map component-of-term* (*Keys-to-list* (*map fst bs*)))) −

count-const-lt-components [*b*←*bs* . *fst b* ≠ *0*]

lemma *count-const-lt-components-alt*:
count-const-lt-components hs = *card* (*const-lt-component* ‘ *fst* ‘ *set hs* − {*None*})
by (*simp add*: *count-const-lt-components-def card-set*[*symmetric*] *set-diff-eq image-comp del*: *not-None-eq*)

lemma *count-rem-components-alt*:
count-rem-components bs + *card* (*const-lt-component* ‘ (*fst* ‘ *set bs* − {*0*}) − {*None*}) =
card (*component-of-term* ‘ *Keys* (*fst* ‘ *set bs*))
proof −
have *eq*: *fst* ‘ {*x* ∈ *set bs*. *fst x* ≠ *0*} = *fst* ‘ *set bs* − {*0*} **by** *fastforce*
have *card* (*const-lt-component* ‘ (*fst* ‘ *set bs* − {*0*}) − {*None*}) ≤ *card* (*component-of-term* ‘ *Keys* (*fst* ‘ *set bs*))
by (*rule card-const-lt-component-le*, *rule finite-imageI*, *fact finite-set*)
thus *?thesis*
by (*simp add*: *count-rem-components-def card-set*[*symmetric*] *set-Keys-to-list count-const-lt-components-alt eq*)
qed

lemma *rem-comps-spec-count-rem-components*: *rem-comps-spec bs* (*count-rem-components bs*, *data*)
by (*simp only*: *rem-comps-spec-def fst-conv count-rem-components-alt*)

definition *full-gb* :: (′*t*, ′*b*, ′*c*) *pdata list* ⇒ (′*t*, ′*b*::*zero-neq-one*, ′*c*::*default*) *pdata list*
where *full-gb bs* = *map* (λ*k*. (*monomial 1* (*term-of-pair* (*0*, *k*)), *0*, *default*))
(*remdups* (*map component-of-term* (*Keys-to-list* (*map fst bs*))))

lemma *fst-set-full-gb*:
fst ‘ *set* (*full-gb bs*) = (λ*v*. *monomial 1* (*term-of-pair* (*0*, *component-of-term v*))) ‘ *Keys* (*fst* ‘ *set bs*)
by (*simp add*: *full-gb-def set-Keys-to-list image-comp*)

lemma *Keys-full-gb*:
Keys (*fst* ‘ *set* (*full-gb bs*)) = (λ*v*. *term-of-pair* (*0*, *component-of-term v*)) ‘ *Keys* (*fst* ‘ *set bs*)
by (*auto simp add*: *fst-set-full-gb Keys-def image-image*)

lemma *pps-full-gb*: *pp-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb bs*)) ⊆ {*0*}
by (*simp add*: *Keys-full-gb image-comp image-subset-iff term-simps*)

lemma *components-full-gb*:
component-of-term ‘ *Keys* (*fst* ‘ *set* (*full-gb bs*)) = *component-of-term* ‘ *Keys* (*fst* ‘ *set bs*)
by (*simp add*: *Keys-full-gb image-comp*, *rule image-cong*, *fact refl*, *simp add*: *term-simps*)

lemma *full-gb-is-full-pmdl*: *is-full-pmdl* (*fst* ‘ *set* (*full-gb bs*))
  for *bs*::(*'t*, *'b*::*field*, *'c*::*default*) *pdata list*
proof (*rule is-full-pmdlI-lt-finite*)
 from *finite-set* **show** *finite* (*fst* ‘ *set* (*full-gb bs*)) **by** (*rule finite-imageI*)
next
 fix *k*
 assume *k* ∈ *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb bs*))
 then obtain *v* **where** *v* ∈ *Keys* (*fst* ‘ *set* (*full-gb bs*)) **and** *k*: *k* = *component-of-term v* **..**
 from *this*(*1*) **obtain** *b* **where** *b* ∈ *fst* ‘ *set* (*full-gb bs*) **and** *v* ∈ *keys b* **by** (*rule in-KeysE*)
 from *this*(*1*) **obtain** *u* **where** *u* ∈ *Keys* (*fst* ‘ *set bs*) **and** *b*: *b* = *monomial 1* (*term-of-pair* (*0*, *component-of-term u*))
  unfolding *fst-set-full-gb* **..**
 have *lt*: *lt b* = *term-of-pair* (*0*, *component-of-term u*) **by** (*simp add*: *b lt-monomial*)
 from ‹*v* ∈ *keys b*› **have** *v*: *v* = *term-of-pair* (*0*, *component-of-term u*) **by** (*simp add*: *b*)
 show ∃ *b*∈*fst* ‘ *set* (*full-gb bs*). *b* ≠ *0* ∧ *component-of-term* (*lt b*) = *k* ∧ *lp b* = *0*
 proof (*intro bexI conjI*)
  show *b* ≠ *0* **by** (*simp add*: *b monomial-0-iff*)
 next
  show *component-of-term* (*lt b*) = *k* **by** (*simp add*: *lt term-simps k v*)
 next
  show *lp b* = *0* **by** (*simp add*: *lt term-simps*)
 qed *fact*
qed

In fact, *is-full-pmdl* (*fst* ‘ *set* (*full-gb ?bs*)) also holds if *'b* is no field.

lemma *full-gb-isGB*: *is-Groebner-basis* (*fst* ‘ *set* (*full-gb bs*))
proof (*rule Buchberger-criterion-finite*)
 from *finite-set* **show** *finite* (*fst* ‘ *set* (*full-gb bs*)) **by** (*rule finite-imageI*)
next
 fix *p q* :: $'t \Rightarrow_0 'b$
 assume *p* ∈ *fst* ‘ *set* (*full-gb bs*)
 then obtain *v* **where** *p*: *p* = *monomial 1* (*term-of-pair* (*0*, *component-of-term v*))
  unfolding *fst-set-full-gb* **..**
 hence *lt*: *component-of-term* (*lt p*) = *component-of-term v* **by** (*simp add*: *lt-monomial term-simps*)
 assume *q* ∈ *fst* ‘ *set* (*full-gb bs*)
 then obtain *u* **where** *q*: *q* = *monomial 1* (*term-of-pair* (*0*, *component-of-term u*))
  unfolding *fst-set-full-gb* **..**
 hence *lq*: *component-of-term* (*lt q*) = *component-of-term u* **by** (*simp add*: *lt-monomial term-simps*)
 assume *component-of-term* (*lt p*) = *component-of-term* (*lt q*)
 hence *component-of-term v* = *component-of-term u* **by** (*simp only*: *lt lq*)
 hence *p* = *q* **by** (*simp only*: *p q*)
 moreover assume *p* ≠ *q*

ultimately show (*red* (*fst* ‘ *set* (*full-gb bs*)))** (*spoly p q*) *0* **by** (*simp only*:)
qed

6.2.8 Specification of the *completion* parameter

definition *compl-struct* :: ('*t*, '*b*::*field*, '*c*, '*d*) *complT* ⇒ *bool*
where *compl-struct compl* ⟷
(∀ *gs bs ps sps data*. *sps* ≠ [] ⟶ *set sps* ⊆ *set ps* ⟶
(∀ *d*. *dickson-grading d* ⟶
dgrad-p-set-le d (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) (*args-to-set* (*gs*, *bs*, *ps*))) ∧
component-of-term ‘ *Keys* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) ∧
0 ∉ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) ∧
(∀ *h*∈*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)). ∀ *b*∈*set gs* ∪ *set bs*. *fst b* ≠ *0* ⟶ ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)))

lemma *compl-structI*:
assumes ⋀*d gs bs ps sps data*. *dickson-grading d* ⟹ *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹
dgrad-p-set-le d (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) (*args-to-set* (*gs*, *bs*, *ps*))
assumes ⋀*gs bs ps sps data*. *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹
component-of-term ‘ *Keys* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
assumes ⋀*gs bs ps sps data*. *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹ *0* ∉ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
assumes ⋀*gs bs ps sps h b data*. *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹ *h* ∈ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) ⟹
b ∈ *set gs* ∪ *set bs* ⟹ *fst b* ≠ *0* ⟹ ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
shows *compl-struct compl*
unfolding *compl-struct-def* **using** *assms* **by** *auto*

lemma *compl-structD1*:
assumes *compl-struct compl* **and** *dickson-grading d* **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
shows *dgrad-p-set-le d* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) (*args-to-set* (*gs*, *bs*, *ps*))
using *assms* **unfolding** *compl-struct-def* **by** *blast*

lemma *compl-structD2*:
assumes *compl-struct compl* **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
shows *component-of-term* ‘ *Keys* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
using *assms* **unfolding** *compl-struct-def* **by** *blast*

lemma *compl-structD3*:
assumes *compl-struct compl* **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
shows *0* ∉ *fst* ' *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
using *assms* **unfolding** *compl-struct-def* **by** *blast*

lemma *compl-structD4*:
assumes *compl-struct compl* **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
and *h* ∈ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) **and** *b* ∈ *set gs* ∪ *set bs* **and** *fst b* ≠ *0*
shows ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
using *assms* **unfolding** *compl-struct-def* **by** *blast*

definition *struct-spec* :: ('t, 'b::field, 'c, 'd) *selT* ⇒ ('t, 'b, 'c, 'd) *apT* ⇒ ('t, 'b, 'c, 'd) *abT* ⇒
('t, 'b, 'c, 'd) *complT* ⇒ *bool*
where *struct-spec sel ap ab compl* ⟷ (*sel-spec sel* ∧ *ap-spec ap* ∧ *ab-spec ab* ∧ *compl-struct compl*)

lemma *struct-specI*:
assumes *sel-spec sel* **and** *ap-spec ap* **and** *ab-spec ab* **and** *compl-struct compl*
shows *struct-spec sel ap ab compl*
unfolding *struct-spec-def* **using** *assms* **by** (*intro conjI*)

lemma *struct-specD1*:
assumes *struct-spec sel ap ab compl*
shows *sel-spec sel*
using *assms* **unfolding** *struct-spec-def* **by** (*elim conjE*)

lemma *struct-specD2*:
assumes *struct-spec sel ap ab compl*
shows *ap-spec ap*
using *assms* **unfolding** *struct-spec-def* **by** (*elim conjE*)

lemma *struct-specD3*:
assumes *struct-spec sel ap ab compl*
shows *ab-spec ab*
using *assms* **unfolding** *struct-spec-def* **by** (*elim conjE*)

lemma *struct-specD4*:
assumes *struct-spec sel ap ab compl*
shows *compl-struct compl*
using *assms* **unfolding** *struct-spec-def* **by** (*elim conjE*)

lemmas *struct-specD* = *struct-specD1 struct-specD2 struct-specD3 struct-specD4*

definition *compl-pmdl* :: ('t, 'b::field, 'c, 'd) *complT* ⇒ *bool*
where *compl-pmdl compl* ⟷
(∀ *gs bs ps sps data*. *is-Groebner-basis* (*fst* ' *set gs*) ⟶ *sps* ≠ [] ⟶ *set sps* ⊆ *set ps* ⟶

unique-idx (*gs* @ *bs*) *data* ⟶
fst ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*)))

lemma *compl-pmdlI*:
assumes ⋀*gs bs ps sps data*. *is-Groebner-basis* (*fst* ‘ *set gs*) ⟹ *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹
unique-idx (*gs* @ *bs*) *data* ⟹
fst ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*))
shows *compl-pmdl compl*
unfolding *compl-pmdl-def* **using** *assms* **by** *blast*

lemma *compl-pmdlD*:
assumes *compl-pmdl compl* **and** *is-Groebner-basis* (*fst* ‘ *set gs*)
and *sps* ≠ [] **and** *set sps* ⊆ *set ps* **and** *unique-idx* (*gs* @ *bs*) *data*
shows *fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*))
using *assms* **unfolding** *compl-pmdl-def* **by** *blast*

definition *compl-conn* :: (′*t*, ′*b*::*field*, ′*c*, ′*d*) *complT* ⇒ *bool*
where *compl-conn compl* ⟷
(∀ *d m gs bs ps sps p q data*. *dickson-grading d* ⟶ *fst* ‘ *set gs* ⊆ *dgrad-p-set d m* ⟶
is-Groebner-basis (*fst* ‘ *set gs*) ⟶ *fst* ‘ *set bs* ⊆ *dgrad-p-set d m* ⟶
set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟶ *sps* ≠ [] ⟶ *set sps* ⊆ *set ps* ⟶
unique-idx (*gs* @ *bs*) *data* ⟶ (*p*, *q*) ∈ *set sps* ⟶ *fst p* ≠ *0* ⟶ *fst q* ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) (*fst p*) (*fst q*))

Informally, *compl-conn compl* means that, for suitable arguments *gs*, *bs*, *ps* and *sps*, the value of *compl gs bs ps sps* is a list *hs* such that the critical pairs of all elements in *sps* can be connected modulo *set gs* ∪ *set bs* ∪ *set hs*.

lemma *compl-connI*:
assumes ⋀*d m gs bs ps sps p q data*. *dickson-grading d* ⟹ *fst* ‘ *set gs* ⊆ *dgrad-p-set d m* ⟹
is-Groebner-basis (*fst* ‘ *set gs*) ⟹ *fst* ‘ *set bs* ⊆ *dgrad-p-set d m* ⟹
set ps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟹ *sps* ≠ [] ⟹ *set sps* ⊆ *set ps* ⟹
unique-idx (*gs* @ *bs*) *data* ⟹ (*p*, *q*) ∈ *set sps* ⟹ *fst p* ≠ *0* ⟹ *fst q* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) (*fst p*) (*fst q*)
shows *compl-conn compl*
unfolding *compl-conn-def* **using** *assms* **by** *presburger*

lemma *compl-connD*:
assumes *compl-conn compl* **and** *dickson-grading d* **and** *fst* ‘ *set gs* ⊆ *dgrad-p-set*

d m
and *is-Groebner-basis* (*fst ‘ set gs*) **and** *fst ‘ set bs* ⊆ *dgrad-p-set d m*
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
and *unique-idx* (*gs* @ *bs*) *data* **and** (*p*, *q*) ∈ *set sps* **and** *fst p* ≠ *0* **and** *fst q* ≠ *0*
shows *crit-pair-cbelow-on d m* (*fst ‘* (*set gs* ∪ *set bs*) ∪ *fst ‘ set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))) (*fst p*) (*fst q*)
using *assms* **unfolding** *compl-conn-def Un-assoc* **by** *blast*

6.2.9 Function *gb-schema-dummy*

definition (**in** −) *add-indices* :: ((*′a*, *′b*, *′c*) *pdata′ list* × *′d*) ⇒ (*nat* × *′d*) ⇒ ((*′a*, *′b*, *′c*) *pdata list* × *nat* × *′d*)
where [*code del*]: *add-indices ns data* =
(*map-idx* (λ*h i*. (*fst h*, *i*, *snd h*)) (*fst ns*) (*fst data*), *fst data* + *length* (*fst ns*), *snd ns*)

lemma (**in** −) *add-indices-code* [*code*]:
add-indices (*ns*, *data*) (*n*, *data′*) = (*map-idx* (λ(*h*, *d*) *i*. (*h*, *i*, *d*)) *ns n*, *n* + *length ns*, *data*)
by (*simp add*: *add-indices-def case-prod-beta′*)

lemma *fst-add-indices*: *map fst* (*fst* (*add-indices ns data′*)) = *map fst* (*fst ns*)
by (*simp add*: *add-indices-def map-map-idx map-idx-no-idx*)

corollary *fst-set-add-indices*: *fst ‘ set* (*fst* (*add-indices ns data′*)) = *fst ‘ set* (*fst ns*)
using *fst-add-indices* **by** (*metis set-map*)

lemma *in-set-add-indicesE*:
assumes *f* ∈ *set* (*fst* (*add-indices aux data*))
obtains *i* **where** *i* < *length* (*fst aux*) **and** *f* = (*fst* ((*fst aux*) ! *i*), *fst data* + *i*, *snd* ((*fst aux*) ! *i*))
proof −
let *?hs* = *fst* (*add-indices aux data*)
from *assms* **obtain** *i* **where** *i* < *length ?hs* **and** *f* = *?hs* ! *i* **by** (*metis in-set-conv-nth*)
from *this*(*1*) **have** *i* < *length* (*fst aux*) **by** (*simp add*: *add-indices-def*)
hence *?hs* ! *i* = (*fst* ((*fst aux*) ! *i*), *fst data* + *i*, *snd* ((*fst aux*) ! *i*))
unfolding *add-indices-def fst-conv* **by** (*rule map-idx-nth*)
hence *f* = (*fst* ((*fst aux*) ! *i*), *fst data* + *i*, *snd* ((*fst aux*) ! *i*)) **by** (*simp add*: ‹*f* = *?hs* ! *i*›)
with ‹*i* < *length* (*fst aux*)› **show** *?thesis* **..**
qed

definition *gb-schema-aux-term1* :: (((*′t*, *′b*::*field*, *′c*) *pdata list* × (*′t*, *′b*, *′c*) *pdata-pair list*) ×
((*′t*, *′b*, *′c*) *pdata list* × (*′t*, *′b*, *′c*) *pdata-pair list*)) *set*
where *gb-schema-aux-term1* = {(*a*, *b*::(*′t*, *′b*, *′c*) *pdata list*). (*fst ‘ set a*) ⊐*p* (*fst ‘ set b*)} <*∗lex∗*>

(*measure* (*card* ∘ *set*))

definition *gb-schema-aux-term2* ::
('*a* ⇒ *nat*) ⇒ ('*t*, '*b*::*field*, '*c*) *pdata list* ⇒ ((('*t*, '*b*, '*c*) *pdata list* × ('*t*, '*b*, '*c*) *pdata-pair list*) ×
(('*t*, '*b*, '*c*) *pdata list* × ('*t*, '*b*, '*c*) *pdata-pair list*)) *set*
where *gb-schema-aux-term2 d gs* = {(*a*, *b*). *dgrad-p-set-le d* (*args-to-set* (*gs*, *a*)) (*args-to-set* (*gs*, *b*)) ∧
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *a*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *b*))}

definition *gb-schema-aux-term* **where** *gb-schema-aux-term d gs* = *gb-schema-aux-term1* ∩ *gb-schema-aux-term2 d gs*

gb-schema-aux-term is needed for proving termination of function *gb-schema-aux*.

lemma *gb-schema-aux-term1-wf-on*:
assumes *dickson-grading d* **and** *finite K*
shows *wfp-on* (λ*x y*. (*x*, *y*) ∈ *gb-schema-aux-term1*)
{*x*::(('*t*, '*b*, '*c*) *pdata list*) × ((('*t*, '*b*::*field*, '*c*) *pdata-pair list*)).
args-to-set (*gs*, *x*) ⊆ *dgrad-p-set d m* ∧ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *x*)) ⊆ *K*}
proof (*rule wfp-onI-min*)
let *?B* = *dgrad-p-set d m*
let *?A* = {*x*::(('*t*, '*b*, '*c*) *pdata list*) × ((('*t*, '*b*, '*c*) *pdata-pair list*)).
args-to-set (*gs*, *x*) ⊆ *?B* ∧ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *x*)) ⊆ *K*}
let *?C* = *Pow ?B* ∩ {*F*. *component-of-term* ‘ *Keys F* ⊆ *K*}
have *A-sub-Pow*: (*image fst*) ‘ *set* ‘ *fst* ‘ *?A* ⊆ *?C*
proof
fix *x*
assume *x* ∈ (*image fst*) ‘ *set* ‘ *fst* ‘ *?A*
then obtain *x1* **where** *x1* ∈ *set* ‘ *fst* ‘ *?A* **and** *x*: *x* = *fst* ‘ *x1* **by** *auto*
from *this*(*1*) **obtain** *x2* **where** *x2* ∈ *fst* ‘ *?A* **and** *x1*: *x1* = *set x2* **by** *auto*
from *this*(*1*) **obtain** *x3* **where** *x3* ∈ *?A* **and** *x2*: *x2* = *fst x3* **by** *auto*
from *this*(*1*) **have** *args-to-set* (*gs*, *x3*) ⊆ *?B* **and** *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *x3*)) ⊆ *K*
by *simp-all*
thus *x* ∈ *?C* **by** (*simp add*: *args-to-set-def x x1 x2 image-Un Keys-Un*)
qed

fix *x Q*
assume *x* ∈ *Q* **and** *Q* ⊆ *?A*
have *Q-sub-A*: (*image fst*) ‘ *set* ‘ *fst* ‘ *Q* ⊆ (*image fst*) ‘ *set* ‘ *fst* ‘ *?A*
by ((*rule image-mono*)+, *fact*)
from *assms* **have** *wfp-on* (⊐*p*) *?C* **by** (*rule red-supset-wf-on*)
moreover have *fst* ‘ *set* (*fst x*) ∈ (*image fst*) ‘ *set* ‘ *fst* ‘ *Q*
by (*rule*, *fact refl*, *rule*, *fact refl*, *rule*, *fact refl*, *simp add*: ‹*x* ∈ *Q*›)
moreover from *Q-sub-A A-sub-Pow* **have** (*image fst*) ‘ *set* ‘ *fst* ‘ *Q* ⊆ *?C* **by** (*rule subset-trans*)

```
  ultimately obtain z1 where z1 ∈ (image fst) ` set ` fst ` Q
    and 2: ⋀y. y ⊐p z1 ⟹ y ∉ (image fst) ` set ` fst ` Q by (rule wfp-onE-min,
auto)
  from this(1) obtain x1 where x1 ∈ Q and z1: z1 = fst ` set (fst x1) by auto

  let ?Q2 = {q ∈ Q. fst ` set (fst q) = z1}
  have snd x1 ∈ snd ` ?Q2 by (rule, fact refl, simp add: ‹x1 ∈ Q› z1)
  with wf-measure obtain z2 where z2 ∈ snd ` ?Q2
    and 3: ⋀y. (y, z2) ∈ measure (card ∘ set) ⟹ y ∉ snd ` ?Q2
    by (rule wfE-min, blast)
  from this(1) obtain z where z ∈ ?Q2 and z2: z2 = snd z ..
  from this(1) have z ∈ Q and eq1: fst ` set (fst z) = z1 by blast+
  from this(1) show ∃ z∈Q. ∀ y∈?A. (y, z) ∈ gb-schema-aux-term1 ⟶ y ∉ Q
  proof
    show ∀ y∈?A. (y, z) ∈ gb-schema-aux-term1 ⟶ y ∉ Q
    proof (intro ballI impI)
      fix y
      assume y ∈ ?A
      assume (y, z) ∈ gb-schema-aux-term1
      hence (fst ` set (fst y) ⊐p z1 ∨ (fst y = fst z ∧ (snd y, z2) ∈ measure (card
∘ set)))
        by (simp add: gb-schema-aux-term1-def eq1[symmetric] z2 in-lex-prod-alt)
      thus y ∉ Q
      proof (elim disjE conjE)
        assume fst ` set (fst y) ⊐p z1
        hence fst ` set (fst y) ∉ (image fst) ` set ` fst ` Q by (rule 2)
        thus ?thesis by auto
      next
        assume (snd y, z2) ∈ measure (card ∘ set)
        hence snd y ∉ snd ` ?Q2 by (rule 3)
        hence y ∉ ?Q2 by blast
        moreover assume fst y = fst z
        ultimately show ?thesis by (simp add: eq1)
      qed
    qed
  qed
qed

lemma gb-schema-aux-term-wf:
  assumes dickson-grading d
  shows wf (gb-schema-aux-term d gs)
proof (rule wfI-min)
  fix x::(('t, 'b, 'c) pdata list) × (('t, 'b, 'c) pdata-pair list) and Q
  assume x ∈ Q
  let ?A = args-to-set (gs, x)
  have finite ?A by (simp add: args-to-set-def)
  then obtain m where A: ?A ⊆ dgrad-p-set d m by (rule dgrad-p-set-exhaust)
  define K where K = component-of-term ` Keys ?A
  from ‹finite ?A› have finite K unfolding K-def by (rule finite-imp-finite-component-Keys)
```

let *?B* = *dgrad-p-set d m*
let *?Q* = {*q* ∈ *Q*. *args-to-set* (*gs*, *q*) ⊆ *?B* ∧ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *q*)) ⊆ *K*}
from *assms* ‹*finite K*› **have** *wfp-on* (λ*x y*. (*x*, *y*) ∈ *gb-schema-aux-term1*) {*x*. *args-to-set* (*gs*, *x*) ⊆ *?B* ∧ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *x*)) ⊆ *K*}
by (*rule gb-schema-aux-term1-wf-on*)
moreover from ‹*x* ∈ *Q*› *A* **have** *x* ∈ *?Q* **by** (*simp add*: *K-def*)
moreover have *?Q* ⊆ {*x*. *args-to-set* (*gs*, *x*) ⊆ *?B* ∧ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *x*)) ⊆ *K*} **by** *auto*
ultimately obtain *z* **where** *z* ∈ *?Q*
and ∗: ⋀*y*. (*y*, *z*) ∈ *gb-schema-aux-term1* ⟹ *y* ∉ *?Q* **by** (*rule wfp-onE-min*, *blast*)
from *this*(*1*) **have** *z* ∈ *Q* **and** *a*: *args-to-set* (*gs*, *z*) ⊆ *?B* **and** *b*: *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *z*)) ⊆ *K*
by *simp-all*
from *this*(*1*) **show** ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *gb-schema-aux-term d gs* ⟶ *y* ∉ *Q*
proof
show ∀ *y*. (*y*, *z*) ∈ *gb-schema-aux-term d gs* ⟶ *y* ∉ *Q*
proof (*intro allI impI*)
fix *y*
assume (*y*, *z*) ∈ *gb-schema-aux-term d gs*
hence (*y*, *z*) ∈ *gb-schema-aux-term1* **and** (*y*, *z*) ∈ *gb-schema-aux-term2 d gs*
by (*simp-all add*: *gb-schema-aux-term-def*)
from *this*(*2*) **have** *dgrad-p-set-le d* (*args-to-set* (*gs*, *y*)) (*args-to-set* (*gs*, *z*))
and *comp-sub*: *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *y*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *z*))
by (*simp-all add*: *gb-schema-aux-term2-def*)
from *this*(*1*) ‹*args-to-set* (*gs*, *z*) ⊆ *?B*› **have** *args-to-set* (*gs*, *y*) ⊆ *?B*
by (*rule dgrad-p-set-le-dgrad-p-set*)
moreover from *comp-sub b* **have** *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *y*)) ⊆ *K*
by (*rule subset-trans*)
moreover from ‹(*y*, *z*) ∈ *gb-schema-aux-term1*› **have** *y* ∉ *?Q* **by** (*rule* ∗)
ultimately show *y* ∉ *Q* **by** *simp*
qed
qed
qed

lemma *dgrad-p-set-le-args-to-set-ab*:
assumes *dickson-grading d* **and** *ap-spec ap* **and** *ab-spec ab* **and** *compl-struct compl*
assumes *sps* ≠ [] **and** *set sps* ⊆ *set ps* **and** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*)
shows *dgrad-p-set-le d* (*args-to-set* (*gs*, *ab gs bs hs data'*, *ap gs bs* (*ps* −− *sps*) *hs data'*)) (*args-to-set* (*gs*, *bs*, *ps*))
(**is** *dgrad-p-set-le* - *?l ?r*)
proof −
have *dgrad-p-set-le d ?l*

(*fst* ‘ (*set gs* ∪ *set bs* ∪ *fst* ‘ *set* (*ps* −− *sps*) ∪ *snd* ‘ *set* (*ps* −− *sps*) ∪ *set hs*))
by (*rule dgrad-p-set-le-subset*, *rule args-to-set-subset*[*OF assms*(*2*, *3*)])
also have *dgrad-p-set-le d* ... *?r* **unfolding** *image-Un*
proof (*intro dgrad-p-set-leI-Un*)
show *dgrad-p-set-le d* (*fst* ‘ *set gs*) (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-subset*, *auto simp add*: *args-to-set-def*)
next
show *dgrad-p-set-le d* (*fst* ‘ *set bs*) (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-subset*, *auto simp add*: *args-to-set-def*)
next
show *dgrad-p-set-le d* (*fst* ‘ *fst* ‘ *set* (*ps* −− *sps*)) (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-subset*, *auto simp add*: *args-to-set-def set-diff-list*)
next
show *dgrad-p-set-le d* (*fst* ‘ *snd* ‘ *set* (*ps* −− *sps*)) (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-subset*, *auto simp add*: *args-to-set-def set-diff-list*)
next
from *assms*(*4*, *1*, *5*, *6*) **show** *dgrad-p-set-le d* (*fst* ‘ *set hs*) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *assms*(*7*) *fst-set-add-indices* **by** (*rule compl-structD1*)
qed
finally show *?thesis* **.**
qed

corollary *dgrad-p-set-le-args-to-set-struct*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl* **and** *ps* ≠ []
assumes *sps* = *sel gs bs ps data* **and** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*)
shows *dgrad-p-set-le d* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) (*args-to-set* (*gs*, *bs*, *ps*))
proof −
from *assms*(*2*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab*
and *compl*: *compl-struct compl* **by** (*rule struct-specD*)+
from *sel assms*(*3*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *assms*(*4*) **by** (*rule sel-specD1*, *rule sel-specD2*)
from *assms*(*1*) *ap ab compl this assms*(*5*) **show** *?thesis* **by** (*rule dgrad-p-set-le-args-to-set-ab*)
qed

lemma *components-subset-ab*:
assumes *ap-spec ap* **and** *ab-spec ab* **and** *compl-struct compl*
assumes *sps* ≠ [] **and** *set sps* ⊆ *set ps* **and** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*)
shows *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) (**is** *?l* ⊆ *?r*)
proof −
have *?l* ⊆ *component-of-term* ‘ *Keys* (*fst* ‘ (*set gs* ∪ *set bs* ∪ *fst* ‘ *set* (*ps* −− *sps*) ∪ *snd* ‘ *set* (*ps* −− *sps*) ∪ *set hs*))
by (*rule image-mono*, *rule Keys-mono*, *rule args-to-set-subset*[*OF assms*(*1*, *2*)])

also have ... ⊆ *?r* **unfolding** *image-Un Keys-Un Un-subset-iff*
proof (*intro conjI*)
show *component-of-term* ‘ *Keys* (*fst* ‘ *set gs*) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *args-to-set-def*)
next
show *component-of-term* ‘ *Keys* (*fst* ‘ *set bs*) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *args-to-set-def*)
next
show *component-of-term* ‘ *Keys* (*fst* ‘ *fst* ‘ *set* (*ps* −− *sps*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *set-diff-list args-to-set-def*)
next
show *component-of-term* ‘ *Keys* (*fst* ‘ *snd* ‘ *set* (*ps* −− *sps*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *args-to-set-def set-diff-list*)
next
from *assms*(*3*, *4*, *5*) **show** *component-of-term* ‘ *Keys* (*fst* ‘ *set hs*) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *assms*(*6*) *fst-set-add-indices* **by** (*rule compl-structD2*)
qed
finally show *?thesis* **.**
qed

corollary *components-subset-struct*:
assumes *struct-spec sel ap ab compl* **and** *ps* ≠ []
assumes *sps* = *sel gs bs ps data* **and** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*)
shows *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
proof −
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab*
and *compl*: *compl-struct compl* **by** (*rule struct-specD*)+
from *sel assms*(*2*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *assms*(*3*) **by** (*rule sel-specD1*, *rule sel-specD2*)
from *ap ab compl this assms*(*4*) **show** *?thesis* **by** (*rule components-subset-ab*)
qed

corollary *components-struct*:
assumes *struct-spec sel ap ab compl* **and** *ps* ≠ [] **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
assumes *sps* = *sel gs bs ps data* **and** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*)
shows *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) =
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) (**is** *?l* = *?r*)
proof

from *assms*(*1*, *2*, *4*, *5*) **show** *?l* ⊆ *?r* **by** (*rule components-subset-struct*)
next
from *assms*(*1*) **have** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **and** *compl*: *compl-struct compl*
by (*rule struct-specD*)+
from *ap ab assms*(*3*)
have *sub*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆ *set* (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)
show *?r* ⊆ *?l*
by (*simp add*: *args-to-set-subset-Times*[*OF sub*] *args-to-set-subset-Times*[*OF assms*(*3*)] *ab-specD1*[*OF ab*],
rule image-mono, *rule Keys-mono*, *blast*)
qed

lemma *struct-spec-red-supset*:
assumes *struct-spec sel ap ab compl* **and** *ps* ≠ [] **and** *sps* = *sel gs bs ps data*
and *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps data*) *data*) **and** *hs* ≠ []
shows (*fst* ‘ *set* (*ab gs bs hs data′*)) ⊐*p* (*fst* ‘ *set bs*)
proof −
from *assms*(*5*) **have** *set hs* ≠ {} **by** *simp*
then obtain *h′* **where** *h′* ∈ *set hs* **by** *fastforce*
let *?h* = *fst h′*
let *?m* = *monomial* (*lc ?h*) (*lt ?h*)
from ‹*h′* ∈ *set hs*› **have** *h-in*: *?h* ∈ *fst* ‘ *set hs* **by** *simp*
hence *?h* ∈ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
by (*simp only*: *assms*(*4*) *fst-set-add-indices*)
then obtain *h′′* **where** *h′′-in*: *h′′* ∈ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
and *?h* = *fst h′′* **..**
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab*
and *compl*: *compl-struct compl* **by** (*rule struct-specD*)+
from *sel assms*(*2*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps* **unfolding** *assms*(*3*)
by (*rule sel-specD1*, *rule sel-specD2*)
from *h-in compl-structD3*[*OF compl this*] **have** *?h* ≠ *0* **unfolding** *assms*(*4*) *fst-set-add-indices*
by *metis*
show *?thesis*
proof (*simp add*: *ab-specD1*[*OF ab*] *image-Un*, *rule*)
fix *q*
assume *is-red* (*fst* ‘ *set bs*) *q*
moreover have *fst* ‘ *set bs* ⊆ *fst* ‘ *set bs* ∪ *fst* ‘ *set hs* **by** *simp*
ultimately show *is-red* (*fst* ‘ *set bs* ∪ *fst* ‘ *set hs*) *q* **by** (*rule is-red-subset*)
next
from ‹*?h* ≠ *0*› **have** *lc ?h* ≠ *0* **by** (*rule lc-not-0*)
moreover have *?h* ∈ {*?h*} **..**
ultimately have *is-red* {*?h*} *?m* **using** ‹*?h* ≠ *0*› *adds-term-refl* **by** (*rule is-red-monomialI*)
moreover have {*?h*} ⊆ *fst* ‘ *set bs* ∪ *fst* ‘ *set hs* **using** *h-in* **by** *simp*

ultimately show *is-red* (*fst* ' *set bs* ∪ *fst* ' *set hs*) *?m* **by** (*rule is-red-subset*)
next
show ¬ *is-red* (*fst* ' *set bs*) *?m*
proof
assume *is-red* (*fst* ' *set bs*) *?m*
then obtain *b′* **where** *b′* ∈ *fst* ' *set bs* **and** *b′* ≠ *0* **and** *lt b′* $adds_t$ *lt ?h*
by (*rule is-red-monomialE*)
from *this*(*1*) **obtain** *b* **where** *b* ∈ *set bs* **and** *b′*: *b′* = *fst b* **..**
from *this*(*1*) **have** *b* ∈ *set gs* ∪ *set bs* **by** *simp*
from ‹*b′* ≠ *0*› **have** *fst b* ≠ *0* **by** (*simp add*: *b′*)
with *compl* ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*› *h″-in* ‹*b* ∈ *set gs* ∪ *set bs*› **have** ¬
lt (*fst b*) $adds_t$ *lt ?h*
unfolding ‹*?h* = *fst h″*› **by** (*rule compl-structD4*)
from *this* ‹*lt b′* $adds_t$ *lt ?h*› **show** *False* **by** (*simp add*: *b′*)
qed
qed
qed

lemma *unique-idx-append*:
assumes *unique-idx gs data* **and** (*hs*, *data′*) = *add-indices aux data*
shows *unique-idx* (*gs* @ *hs*) *data′*
proof −
from *assms*(*2*) **have** *hs*: *hs* = *fst* (*add-indices aux data*) **and** *data′*: *data′* = *snd*
(*add-indices aux data*)
by (*metis fst-conv*, *metis snd-conv*)
have *len*: *length hs* = *length* (*fst aux*) **by** (*simp add*: *hs add-indices-def*)
have *eq*: *fst data′* = *fst data* + *length hs* **by** (*simp add*: *data′ add-indices-def hs*)
show *?thesis*
proof (*rule unique-idxI*)
fix *f g*
assume *f* ∈ *set* (*gs* @ *hs*) **and** *g* ∈ *set* (*gs* @ *hs*)
hence *d1*: *f* ∈ *set gs* ∪ *set hs* **and** *d2*: *g* ∈ *set gs* ∪ *set hs* **by** *simp-all*
assume *id-eq*: *fst* (*snd f*) = *fst* (*snd g*)
from *d1* **show** *f* = *g*
proof
assume *f* ∈ *set gs*
from *d2* **show** *?thesis*
proof
assume *g* ∈ *set gs*
from *assms*(*1*) ‹*f* ∈ *set gs*› *this id-eq* **show** *?thesis* **by** (*rule unique-idxD1*)
next
assume *g* ∈ *set hs*
then obtain *j* **where** *g* = (*fst* (*fst aux* ! *j*), *fst data* + *j*, *snd* (*fst aux* ! *j*))
unfolding *hs*
by (*rule in-set-add-indicesE*)
hence *fst* (*snd g*) = *fst data* + *j* **by** *simp*
moreover from *assms*(*1*) ‹*f* ∈ *set gs*› **have** *fst* (*snd f*) < *fst data*
by (*rule unique-idxD2*)
ultimately show *?thesis* **by** (*simp add*: *id-eq*)

qed
next
assume *f* ∈ *set hs*
then obtain *i* **where** *f*: *f* = (*fst* (*fst aux* ! *i*), *fst data* + *i*, *snd* (*fst aux* ! *i*)) **unfolding** *hs*
by (*rule in-set-add-indicesE*)
hence ∗: *fst* (*snd f*) = *fst data* + *i* **by** *simp*
from *d2* **show** *?thesis*
proof
assume *g* ∈ *set gs*
with *assms*(*1*) **have** *fst* (*snd g*) < *fst data* **by** (*rule unique-idxD2*)
with ∗ **show** *?thesis* **by** (*simp add*: *id-eq*)
next
assume *g* ∈ *set hs*
then obtain *j* **where** *g*: *g* = (*fst* (*fst aux* ! *j*), *fst data* + *j*, *snd* (*fst aux* ! *j*)) **unfolding** *hs*
by (*rule in-set-add-indicesE*)
hence *fst* (*snd g*) = *fst data* + *j* **by** *simp*
with ∗ **have** *i* = *j* **by** (*simp add*: *id-eq*)
thus *?thesis* **by** (*simp add*: *f g*)
qed
qed
next
fix *f*
assume *f* ∈ *set* (*gs* @ *hs*)
hence *f* ∈ *set gs* ∪ *set hs* **by** *simp*
thus *fst* (*snd f*) < *fst data′*
proof
assume *f* ∈ *set gs*
with *assms*(*1*) **have** *fst* (*snd f*) < *fst data* **by** (*rule unique-idxD2*)
also have ... ≤ *fst data′* **by** (*simp add*: *eq*)
finally show *?thesis* **.**
next
assume *f* ∈ *set hs*
then obtain *i* **where** *i* < *length* (*fst aux*)
and *f* = (*fst* (*fst aux* ! *i*), *fst data* + *i*, *snd* (*fst aux* ! *i*)) **unfolding** *hs*
by (*rule in-set-add-indicesE*)
from *this*(*2*) **have** *fst* (*snd f*) = *fst data* + *i* **by** *simp*
also from ‹*i* < *length* (*fst aux*)› **have** ... < *fst data* + *length* (*fst aux*) **by** *simp*
finally show *?thesis* **by** (*simp only*: *eq len*)
qed
qed
qed

corollary *unique-idx-ab*:
assumes *ab-spec ab* **and** *unique-idx* (*gs* @ *bs*) *data* **and** (*hs*, *data′*) = *add-indices aux data*
shows *unique-idx* (*gs* @ *ab gs bs hs data′*) *data′*

proof −
from *assms*(*2*, *3*) **have** *unique-idx* ((*gs* @ *bs*) @ *hs*) *data′* **by** (*rule unique-idx-append*)
thus *?thesis* **by** (*simp add*: *unique-idx-def ab-specD1*[*OF assms*(*1*)])
qed

lemma *rem-comps-spec-struct*:
assumes *struct-spec sel ap ab compl* **and** *rem-comps-spec* (*gs* @ *bs*) *data* **and** *ps* ≠ []
and *set ps* ⊆ (*set bs*) × (*set gs* ∪ *set bs*) **and** *sps* = *sel gs bs ps* (*snd data*)
and *aux* = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*) **and** (*hs*, *data′*) = *add-indices aux* (*snd data*)
shows *rem-comps-spec* (*gs* @ *ab gs bs hs data′*) (*fst data* − *count-const-lt-components* (*fst aux*), *data′*)
proof −
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **and** *compl*: *compl-struct compl*
by (*rule struct-specD*)+
from *ap ab assms*(*4*)
have *sub*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆ *set* (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)
have *hs*: *hs* = *fst* (*add-indices aux* (*snd data*)) **by** (*simp add*: *assms*(*7*)[*symmetric*])
from *sel assms*(*3*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps* **unfolding** *assms*(*5*)
by (*rule sel-specD1*, *rule sel-specD2*)
have *eq0*: *fst* ‘ *set* (*fst aux*) − {*0*} = *fst* ‘ *set* (*fst aux*)
by (*rule Diff-triv*, *simp add*: *Int-insert-right assms*(*6*), *rule compl-structD3*, *fact*+)
have *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*)) =
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*))
by (*simp add*: *args-to-set-subset-Times*[*OF sub*] *image-Un*)
also from *assms*(*1*, *3*, *4*, *5*) *hs*
have ... = *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) **unfolding** *assms*(*6*)
by (*rule components-struct*)
also have ... = *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*))
by (*simp add*: *args-to-set-subset-Times*[*OF assms*(*4*)] *image-Un*)
finally have *eq*: *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*)) =
component-of-term ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*)) **.**
from *assms*(*2*)
have *eq2*: *card* (*component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*))) =
fst data + *card* (*const-lt-component* ‘ (*fst* ‘ *set* (*gs* @ *bs*) − {*0*}) − {*None*}) (**is** *?a* = - + *?b*)
by (*simp only*: *rem-comps-spec-def*)
have *eq3*: *card* (*const-lt-component* ‘ (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*) − {*0*}) − {*None*}) =
?b + *count-const-lt-components* (*fst aux*) (**is** *?c* = -)
proof (*simp add*: *ab-specD1*[*OF ab*] *image-Un Un-assoc*[*symmetric*] *Un-Diff count-const-lt-components-alt*
hs fst-set-add-indices eq0, *rule card-Un-disjoint*)

show *finite* (*const-lt-component* ‘ (*fst* ‘ *set gs* − {*0*}) − {*None*} ∪ (*const-lt-component* ‘ (*fst* ‘ *set bs* − {*0*}) − {*None*}))
by (*intro finite-UnI finite-Diff finite-imageI finite-set*)
next
show *finite* (*const-lt-component* ‘ *fst* ‘ *set* (*fst aux*) − {*None*})
by (*rule finite-Diff*, *intro finite-imageI*, *fact finite-set*)
next
have (*const-lt-component* ‘ (*fst* ‘ (*set gs* ∪ *set bs*) − {*0*}) − {*None*}) ∩
(*const-lt-component* ‘ *fst* ‘ *set* (*fst aux*) − {*None*}) =
(*const-lt-component* ‘ (*fst* ‘ (*set gs* ∪ *set bs*) − {*0*}) ∩
const-lt-component ‘ *fst* ‘ *set* (*fst aux*)) − {*None*} **by** *blast*
also have ... = {}
proof (*simp*, *rule*, *simp*, *elim conjE*)
fix *k*
assume *k* ∈ *const-lt-component* ‘ (*fst* ‘ (*set gs* ∪ *set bs*) − {*0*})
then obtain *b* **where** *b* ∈ *set gs* ∪ *set bs* **and** *fst b* ≠ *0* **and** *k1*: *k* = *const-lt-component* (*fst b*)
by *blast*
assume *k* ∈ *const-lt-component* ‘ *fst* ‘ *set* (*fst aux*)
then obtain *h* **where** *h* ∈ *set* (*fst aux*) **and** *k2*: *k* = *const-lt-component* (*fst h*) **by** *blast*
show *k* = *None*
proof (*rule ccontr*, *simp*, *elim exE*)
fix *k'*
assume *k* = *Some k'*
hence *lp* (*fst b*) = *0* **and** *component-of-term* (*lt* (*fst b*)) = *k'* **unfolding** *k1*
by (*rule const-lt-component-SomeD1*, *rule const-lt-component-SomeD2*)
moreover from ‹*k* = *Some k'*› **have** *lp* (*fst h*) = *0* **and** *component-of-term* (*lt* (*fst h*)) = *k'*
unfolding *k2* **by** (*rule const-lt-component-SomeD1*, *rule const-lt-component-SomeD2*)
ultimately have *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **by** (*simp add*: *adds-term-def*)
moreover from *compl* ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*› ‹*h* ∈ *set* (*fst aux*)› ‹*b* ∈ *set gs* ∪ *set bs*› ‹*fst b* ≠ *0*›
have ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **unfolding** *assms*(*6*) **by** (*rule compl-structD4*)
ultimately show *False* **by** *simp*
qed
qed
finally show (*const-lt-component* ‘ (*fst* ‘ *set gs* − {*0*}) − {*None*} ∪ (*const-lt-component* ‘ (*fst* ‘ *set bs* − {*0*}) − {*None*})) ∩
(*const-lt-component* ‘ *fst* ‘ *set* (*fst aux*) − {*None*}) = {} **by** (*simp only*: *Un-Diff image-Un*)
qed
have *?c* ≤ *?a* **unfolding** *eq*[*symmetric*]
by (*rule card-const-lt-component-le*, *rule finite-imageI*, *fact finite-set*)
hence *le*: *count-const-lt-components* (*fst aux*) ≤ *fst data* **by** (*simp only*: *eq2 eq3*)
show *?thesis* **by** (*simp only*: *rem-comps-spec-def eq eq2 eq3*, *simp add*: *le*)
qed

lemma *pmdl-struct*:

assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis* (*fst ‘ set gs*)
and *ps* ≠ [] **and** *set ps* ⊆ (*set bs*) × (*set gs* ∪ *set bs*) **and** *unique-idx* (*gs* @ *bs*) (*snd data*)
and *sps* = *sel gs bs ps* (*snd data*) **and** *aux* = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)
and (*hs*, *data′*) = *add-indices aux* (*snd data*)
shows *pmdl* (*fst ‘ set* (*gs* @ *ab gs bs hs data′*)) = *pmdl* (*fst ‘ set* (*gs* @ *bs*))
proof −
have *hs*: *hs* = *fst* (*add-indices aux* (*snd data*)) **by** (*simp add*: *assms*(*9*)[*symmetric*])
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
have *eq*: *fst ‘* (*set gs* ∪ *set* (*ab gs bs hs data′*)) = *fst ‘* (*set gs* ∪ *set bs*) ∪ *fst ‘ set hs*
by (*auto simp add*: *ab-specD1*[*OF ab*])
show *?thesis*
proof (*simp add*: *eq*, *rule*)
show *pmdl* (*fst ‘* (*set gs* ∪ *set bs*) ∪ *fst ‘ set hs*) ⊆ *pmdl* (*fst ‘* (*set gs* ∪ *set bs*))
proof (*rule pmdl.span-subset-spanI*, *simp only*: *Un-subset-iff*, *rule*)
show *fst ‘* (*set gs* ∪ *set bs*) ⊆ *pmdl* (*fst ‘* (*set gs* ∪ *set bs*))
by (*fact pmdl.span-superset*)
next
from *sel assms*(*4*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *assms*(*7*) **by** (*rule sel-specD1*, *rule sel-specD2*)
with *assms*(*2*, *3*) **have** *fst ‘ set hs* ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *hs assms*(*8*) *fst-set-add-indices* **using** *assms*(*6*) **by** (*rule compl-pmdlD*)
thus *fst ‘ set hs* ⊆ *pmdl* (*fst ‘* (*set gs* ∪ *set bs*))
by (*simp only*: *args-to-set-subset-Times*[*OF assms*(*5*)] *image-Un*)
qed
next
show *pmdl* (*fst ‘* (*set gs* ∪ *set bs*)) ⊆ *pmdl* (*fst ‘* (*set gs* ∪ *set bs*) ∪ *fst ‘ set hs*)
by (*rule pmdl.span-mono*, *blast*)
qed
qed

lemma *discarded-subset*:
assumes *ab-spec ab*
and *D′* = *D* ∪ (*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*) $-_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*))
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *D* ⊆ (*set gs* ∪ *set bs*) × (*set gs* ∪ *set bs*)
shows *D′* ⊆ (*set gs* ∪ *set* (*ab gs bs hs data′*)) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
proof −
from *assms*(*1*) **have** *eq*: *set* (*ab gs bs hs data′*) = *set bs* ∪ *set hs* **by** (*rule ab-specD1*)
from *assms*(*4*) **have** *D* ⊆ (*set gs* ∪ (*set bs* ∪ *set hs*)) × (*set gs* ∪ (*set bs* ∪ *set hs*)) **by** *fastforce*
moreover have *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*) $-_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆

(*set gs* ∪ (*set bs* ∪ *set hs*)) × (*set gs* ∪ (*set bs* ∪ *set hs*)) (**is** *?l* ⊆ *?r*)
proof (*rule subset-trans*)
show *?l* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)
by (*simp add*: *minus-pairs-def*)
next
have *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *?r* **by** *fastforce*
moreover have *set* (*ps* −− *sps*) ⊆ *?r*
proof (*rule subset-trans*)
show *set* (*ps* −− *sps*) ⊆ *set ps* **by** (*auto simp*: *set-diff-list*)
next
from *assms*(*3*) **show** *set ps* ⊆ *?r* **by** *fastforce*
qed
ultimately show *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*) ⊆ *?r*
by (*rule Un-least*)
qed
ultimately show *?thesis* **unfolding** *eq assms*(*2*) **by** (*rule Un-least*)
qed

lemma *compl-struct-disjoint*:
assumes *compl-struct compl* **and** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
shows *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) ∩ *fst* ‘ (*set gs* ∪ *set bs*) = {}
proof (*rule*, *rule*)
fix *x*
assume *x* ∈ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) ∩ *fst* ‘ (*set gs* ∪ *set bs*)
hence *x-in*: *x* ∈ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) **and** *x* ∈ *fst* ‘ (*set gs* ∪ *set bs*)
by *simp-all*
from *x-in* **obtain** *h* **where** *h-in*: *h* ∈ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) **and** *x1*: *x* = *fst h* **..**
from *compl-structD3*[*OF assms*, *of gs bs data*] *x-in* **have** *x* ≠ *0* **by** *auto*
from ‹*x* ∈ *fst* ‘ (*set gs* ∪ *set bs*)› **obtain** *b* **where** *b-in*: *b* ∈ *set gs* ∪ *set bs* **and** *x2*: *x* = *fst b* **..**
from ‹*x* ≠ *0*› **have** *fst b* ≠ *0* **by** (*simp add*: *x2*)
with *assms h-in b-in* **have** ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **by** (*rule compl-structD4*)
hence ¬ *lt x* $adds_t$ *lt x* **by** (*simp add*: *x1*[*symmetric*] *x2*)
from *this adds-term-refl* **show** *x* ∈ {} **..**
qed *simp*

context
fixes *sel*::(′*t*, ′*b*::*field*, ′*c*::*default*, ′*d*) *selT* **and** *ap*::(′*t*, ′*b*, ′*c*, ′*d*) *apT*
and *ab*::(′*t*, ′*b*, ′*c*, ′*d*) *abT* **and** *compl*::(′*t*, ′*b*, ′*c*, ′*d*) *complT*
and *gs*::(′*t*, ′*b*, ′*c*) *pdata list*
begin

function (*domintros*) *gb-schema-dummy* :: *nat* × *nat* × ′*d* ⇒ (′*t*, ′*b*, ′*c*) *pdata-pair set* ⇒
(′*t*, ′*b*, ′*c*) *pdata list* ⇒ (′*t*, ′*b*, ′*c*) *pdata-pair list* ⇒

((′t, ′b, ′c) pdata list × (′t, ′b, ′c) pdata-pair set)
where
gb-schema-dummy data D bs ps =
(*if ps* = [] *then*
(*gs* @ *bs*, *D*)
else
(*let sps* = *sel gs bs ps* (*snd data*); *ps0* = *ps* −− *sps*; *aux* = *compl gs bs ps0 sps* (*snd data*);
remcomps = *fst* (*data*) − *count-const-lt-components* (*fst aux*) *in*
(*if remcomps* = *0 then*
(*full-gb* (*gs* @ *bs*), *D*)
else
let (*hs*, *data′*) = *add-indices aux* (*snd data*) *in*
gb-schema-dummy (*remcomps*, *data′*)
(*D* ∪ ((*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)) $-_p$ *set* (*ap gs bs ps0 hs data′*)))
(*ab gs bs hs data′*) (*ap gs bs ps0 hs data′*)
)
)
)
by *pat-completeness auto*

lemma *gb-schema-dummy-domI1*: *gb-schema-dummy-dom* (*data*, *D*, *bs*, [])
by (*rule gb-schema-dummy.domintros*, *simp*)

lemma *gb-schema-dummy-domI2*:
assumes *struct-spec sel ap ab compl*
shows *gb-schema-dummy-dom* (*data*, *D*, *args*)
proof −
from *assms* **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
from *ex-dgrad* **obtain** *d*::′*a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
let *?R* = (*gb-schema-aux-term d gs*)
from *dg* **have** *wf ?R* **by** (*rule gb-schema-aux-term-wf*)
thus *?thesis*
proof (*induct args arbitrary*: *data D rule*: *wf-induct-rule*)
fix *x data D*
assume *IH*: ⋀*y data′ D′*. (*y*, *x*) ∈ *?R* ⟹ *gb-schema-dummy-dom* (*data′*, *D′*, *y*)
obtain *bs ps* **where** *x*: *x* = (*bs*, *ps*) **by** (*meson case-prodE case-prodI2*)
show *gb-schema-dummy-dom* (*data*, *D*, *x*) **unfolding** *x*
proof (*rule gb-schema-dummy.domintros*)
fix *rc0 n0 data0 hs n1 data1*
assume *ps* ≠ []
and *hs-data′*: (*hs*, *n1*, *data1*) = *add-indices* (*compl gs bs* (*ps* −− *sel gs bs ps* (*n0*, *data0*))
(*sel gs bs ps* (*n0*, *data0*)) (*n0*, *data0*)) (*n0*, *data0*)
and *data*: *data* = (*rc0*, *n0*, *data0*)

define *sps* **where** *sps* = *sel gs bs ps* (*n0*, *data0*)
define *data′* **where** *data′* = (*n1*, *data1*)
define *D′* **where** *D′* = *D* ∪
(*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*) $-_p$
set (*ap gs bs* (*ps* −− *sps*) *hs data′*))
define *rc* **where** *rc* = *rc0* − *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sel gs bs ps* (*n0*, *data0*))
(*sel gs bs ps* (*n0*, *data0*)) (*n0*, *data0*)))
from *hs-data′* **have** *hs*: *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)) (*snd data*))
unfolding *sps-def data snd-conv* **by** (*metis fstI*)
show *gb-schema-dummy-dom* ((*rc*, *data′*), *D′*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)
proof (*rule IH*, *simp add*: *x gb-schema-aux-term-def gb-schema-aux-term1-def gb-schema-aux-term2-def*, *intro conjI*)
show *fst* ‘ *set* (*ab gs bs hs data′*) ⊐*p* *fst* ‘ *set bs* ∨
ab gs bs hs data′ = *bs* ∧ *card* (*set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)) < *card* (*set ps*)
proof (*cases hs* = [])
case *True*
have *ab gs bs hs data′* = *bs* ∧ *card* (*set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)) < *card* (*set ps*)
proof (*simp only*: *True*, *rule*)
from *ab* **show** *ab gs bs* [] *data′* = *bs* **by** (*rule ab-specD2*)
next
from *sel* ‹*ps* ≠ []› **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *sps-def* **by** (*rule sel-specD1*, *rule sel-specD2*)
moreover from *sel-specD1*[*OF sel* ‹*ps* ≠ []›] **have** *set sps* ≠ {} **by** (*simp add*: *sps-def*)
ultimately have *set ps* ∩ *set sps* ≠ {} **by** (*simp add*: *inf.absorb-iff2*)
hence *set* (*ps* −− *sps*) ⊂ *set ps* **unfolding** *set-diff-list* **by** *fastforce*
hence *card* (*set* (*ps* −− *sps*)) < *card* (*set ps*) **by** (*simp add*: *psubset-card-mono*)
moreover have *card* (*set* (*ap gs bs* (*ps* −− *sps*) [] *data′*)) ≤ *card* (*set* (*ps* −− *sps*))
by (*rule card-mono*, *fact finite-set*, *rule ap-spec-Nil-subset*, *fact ap*)
ultimately show *card* (*set* (*ap gs bs* (*ps* −− *sps*) [] *data′*)) < *card* (*set ps*) **by** *simp*
qed
thus *?thesis* **..**
next
case *False*
with *assms* ‹*ps* ≠ []› *sps-def hs* **have** *fst* ‘ *set* (*ab gs bs hs data′*) ⊐*p* *fst* ‘ *set bs*
unfolding *data snd-conv* **by** (*rule struct-spec-red-supset*)
thus *?thesis* **..**
qed
next

from *dg assms* ‹*ps* ≠ []› *sps-def hs*
show *dgrad-p-set-le d* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *data snd-conv* **by** (*rule dgrad-p-set-le-args-to-set-struct*)
next
from *assms* ‹*ps* ≠ []› *sps-def hs*
show *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *data snd-conv* **by** (*rule components-subset-struct*)
qed
qed
qed
qed

lemmas *gb-schema-dummy-simp* = *gb-schema-dummy.psimps*[*OF gb-schema-dummy-domI2*]

lemma *gb-schema-dummy-Nil* [*simp*]: *gb-schema-dummy data D bs* [] = (*gs* @ *bs*, *D*)
by (*simp add*: *gb-schema-dummy.psimps*[*OF gb-schema-dummy-domI1*])

lemma *gb-schema-dummy-not-Nil*:
assumes *struct-spec sel ap ab compl* **and** *ps* ≠ []
shows *gb-schema-dummy data D bs ps* =
(*let sps* = *sel gs bs ps* (*snd data*); *ps0* = *ps* −− *sps*; *aux* = *compl gs bs ps0 sps* (*snd data*);
remcomps = *fst* (*data*) − *count-const-lt-components* (*fst aux*) *in*
(*if remcomps* = *0 then*
(*full-gb* (*gs* @ *bs*), *D*)
else
let (*hs*, *data′*) = *add-indices aux* (*snd data*) *in*
gb-schema-dummy (*remcomps*, *data′*)
(*D* ∪ ((*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)) $-_p$ *set* (*ap gs bs ps0 hs data′*)))
(*ab gs bs hs data′*) (*ap gs bs ps0 hs data′*)
)
)
by (*simp add*: *gb-schema-dummy-simp*[*OF assms*(*1*)] *assms*(*2*))

lemma *gb-schema-dummy-induct* [*consumes 1*, *case-names base rec1 rec2*]:
assumes *struct-spec sel ap ab compl*
assumes *base*: ⋀*bs data D*. *P data D bs* [] (*gs* @ *bs*, *D*)
and *rec1*: ⋀*bs ps sps data D*. *ps* ≠ [] ⟹ *sps* = *sel gs bs ps* (*snd data*) ⟹
fst (*data*) ≤ *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*))) ⟹
P data D bs ps (*full-gb* (*gs* @ *bs*), *D*)
and *rec2*: ⋀*bs ps sps aux hs rc data data′ D D′*. *ps* ≠ [] ⟹ *sps* = *sel gs bs ps* (*snd data*) ⟹
aux = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*) ⟹ (*hs*, *data′*) =

add-indices aux (*snd data*) $\Longrightarrow$
rc = *fst data* − *count-const-lt-components* (*fst aux*) $\Longrightarrow$ *0* < *rc* $\Longrightarrow$
D' = (*D* ∪ ((*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)) $-_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data'*))) $\Longrightarrow$
P (*rc*, *data'*) *D'* (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*)
(*gb-schema-dummy* (*rc*, *data'*) *D'* (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*)) $\Longrightarrow$
P data D bs ps (*gb-schema-dummy* (*rc*, *data'*) *D'* (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*))
shows *P data D bs ps* (*gb-schema-dummy data D bs ps*)
proof −
from *assms*(*1*) **have** *gb-schema-dummy-dom* (*data*, *D*, *bs*, *ps*) **by** (*rule gb-schema-dummy-domI2*)
thus *?thesis*
proof (*induct data D bs ps rule*: *gb-schema-dummy.pinduct*)
case (*1 data D bs ps*)
show *?case*
proof (*cases ps* = [])
case *True*
show *?thesis* **by** (*simp add*: *True*, *rule base*)
next
case *False*
show *?thesis*
proof (*simp only*: *gb-schema-dummy-not-Nil*[*OF assms*(*1*) *False*] *Let-def split*: *if-split*, *intro conjI impI*)
define *sps* **where** *sps* = *sel gs bs ps* (*snd data*)
assume *fst data* − *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*))) = *0*
hence *fst data* ≤ *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)))
by *simp*
with *False sps-def* **show** *P data D bs ps* (*full-gb* (*gs* @ *bs*), *D*) **by** (*rule rec1*)
next
define *sps* **where** *sps* = *sel gs bs ps* (*snd data*)
define *aux* **where** *aux* = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)
define *hs* **where** *hs* = *fst* (*add-indices aux* (*snd data*))
define *data'* **where** *data'* = *snd* (*add-indices aux* (*snd data*))
define *rc* **where** *rc* = *fst data* − *count-const-lt-components* (*fst aux*)
define *D'* **where** *D'* = (*D* ∪ ((*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)) $-_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data'*)))
have *eq*: *add-indices aux* (*snd data*) = (*hs*, *data'*) **by** (*simp add*: *hs-def data'-def*)
assume *rc* ≠ *0*
hence *0* < *rc* **by** *simp*
show *P data D bs ps*
(*case add-indices aux* (*snd data*) *of*
(*hs*, *data'*) ⇒
gb-schema-dummy (*rc*, *data'*)
(*D* ∪ (*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*) $-_p$ *set* (*ap*

gs bs (*ps* −− *sps*) *hs data*′)))
(*ab gs bs hs data*′) (*ap gs bs* (*ps* −− *sps*) *hs data*′))
unfolding *eq prod.case D*′*-def*[*symmetric*] **using** *False sps-def aux-def eq*[*symmetric*] *rc-def* ‹*0* < *rc*› *D*′*-def*
proof (*rule rec2*)
show *P* (*rc*, *data*′) *D*′ (*ab gs bs hs data*′) (*ap gs bs* (*ps* −− *sps*) *hs data*′)
(*gb-schema-dummy* (*rc*, *data*′) *D*′ (*ab gs bs hs data*′) (*ap gs bs* (*ps* −− *sps*) *hs data*′))
unfolding *D*′*-def* **using** *False sps-def refl aux-def rc-def* ‹*rc* ≠ *0*› *eq*[*symmetric*] *refl*
by (*rule 1*)
qed
qed
qed
qed
qed

lemma *fst-gb-schema-dummy-dgrad-p-set-le*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
shows *dgrad-p-set-le d* (*fst* ‘ *set* (*fst* (*gb-schema-dummy data D bs ps*))) (*args-to-set* (*gs*, *bs*, *ps*))
using *assms*(*2*)
proof (*induct rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
show *?case* **by** (*simp add*: *args-to-set-def*, *rule dgrad-p-set-le-subset*, *fact subset-refl*)
next
case (*rec1 bs ps sps data D*)
show *?case*
proof (*cases fst* ‘ *set gs* ∪ *fst* ‘ *set bs* ⊆ {*0*})
case *True*
hence *Keys* (*fst* ‘ *set* (*gs* @ *bs*)) = {} **by** (*auto simp add*: *image-Un Keys-def*)
hence *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*))) = {}
by (*simp add*: *components-full-gb*)
hence *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*))) = {} **by** *simp*
thus *?thesis* **by** (*simp add*: *dgrad-p-set-le-def dgrad-set-le-def*)
next
case *False*
from *pps-full-gb* **have** *dgrad-set-le d* (*pp-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*)))) {*0*}
by (*rule dgrad-set-le-subset*)
also have *dgrad-set-le d* ... (*pp-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)))
proof (*rule dgrad-set-leI*, *simp*)
from *False* **have** *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) ≠ {}
by (*simp add*: *args-to-set-alt Keys-Un*, *metis Keys-not-empty singletonI subsetI*)
then obtain *v* **where** *v* ∈ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) **by** *blast*
moreover have *d 0* ≤ *d* (*pp-of-term v*) **by** (*simp add*: *assms*(*1*) *dickson-grading-adds-imp-le*)

ultimately show ∃ *t*∈*Keys* (*args-to-set* (*gs*, *bs*, *ps*)). *d 0* ≤ *d* (*pp-of-term t*)
..
qed
finally show *?thesis* **by** (*simp add*: *dgrad-p-set-le-def*)
qed
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
from *rec2*(*4*) **have** *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)) (*snd data*))
unfolding *rec2*(*3*) **by** (*metis fstI*)
with *assms rec2*(*1*, *2*)
have *dgrad-p-set-le d* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-args-to-set-struct*)
with *rec2*(*8*) **show** *?case* **by** (*rule dgrad-p-set-le-trans*)
qed

lemma *fst-gb-schema-dummy-components*:
assumes *struct-spec sel ap ab compl* **and** *set ps* ⊆ (*set bs*) × (*set gs* ∪ *set bs*)
shows *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*fst* (*gb-schema-dummy data D bs ps*))) =
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
using *assms*
proof (*induct rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
show *?case* **by** (*simp add*: *args-to-set-def*)
next
case (*rec1 bs ps sps data D*)
have *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*))) =
component-of-term ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*)) **by** (*fact components-full-gb*)
also have ... = *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*simp add*: *args-to-set-subset-Times*[*OF rec1.prems*] *image-Un*)
finally show *?case* **by** *simp*
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
from *assms*(*1*) **have** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
from *this rec2.prems*
have *sub*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆ *set* (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)
from *rec2*(*4*) **have** *hs*: *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)) (*snd data*))
unfolding *rec2*(*3*) **by** (*metis fstI*)
have *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)) =
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) (**is** *?l* = *?r*)
proof
from *assms*(*1*) *rec2*(*1*, *2*) *hs* **show** *?l* ⊆ *?r* **by** (*rule components-subset-struct*)
next

show *?r* ⊆ *?l*
by (*simp add*: *args-to-set-subset-Times*[*OF rec2.prems*] *args-to-set-alt2*[*OF ap ab rec2.prems*] *image-Un*,
rule image-mono, *rule Keys-mono*, *blast*)
qed
with *rec2.hyps*(*8*)[*OF sub*] **show** *?case* **by** (*rule trans*)
qed

lemma *fst-gb-schema-dummy-pmdl*:
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis* (*fst* ' *set gs*)
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *unique-idx* (*gs* @ *bs*) (*snd data*)
and *rem-comps-spec* (*gs* @ *bs*) *data*
shows *pmdl* (*fst* ' *set* (*fst* (*gb-schema-dummy data D bs ps*))) = *pmdl* (*fst* ' *set* (*gs* @ *bs*))
proof −
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **and** *compl*: *compl-struct compl*
by (*rule struct-specD*)+
from *assms*(*1*, *4*, *5*, *6*) **show** *?thesis*
proof (*induct bs ps rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
show *?case* **by** *simp*
next
case (*rec1 bs ps sps data D*)
define *aux* **where** *aux* = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)
define *data′* **where** *data′* = *snd* (*add-indices aux* (*snd data*))
define *hs* **where** *hs* = *fst* (*add-indices aux* (*snd data*))
have *hs-data′*: (*hs*, *data′*) = *add-indices aux* (*snd data*) **by** (*simp add*: *hs-def data′-def*)
have *eq*: *set* (*gs* @ *ab gs bs hs data′*) = *set* (*gs* @ *bs* @ *hs*) **by** (*simp add*: *ab-specD1*[*OF ab*])
from *sel rec1*(*1*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps* **unfolding** *rec1*(*2*)
by (*rule sel-specD1*, *rule sel-specD2*)
from *full-gb-is-full-pmdl* **have** *pmdl* (*fst* ' *set* (*full-gb* (*gs* @ *bs*))) = *pmdl* (*fst* ' *set* (*gs* @ *ab gs bs hs data′*))
proof (*rule is-full-pmdl-eq*)
show *is-full-pmdl* (*fst* ' *set* (*gs* @ *ab gs bs hs data′*))
proof (*rule is-full-pmdlI-lt-finite*)
from *finite-set* **show** *finite* (*fst* ' *set* (*gs* @ *ab gs bs hs data′*)) **by** (*rule finite-imageI*)
next
fix *k*
assume *k* ∈ *component-of-term* ' *Keys* (*fst* ' *set* (*gs* @ *ab gs bs hs data′*))
hence *Some k* ∈ *Some* ' *component-of-term* ' *Keys* (*fst* ' *set* (*gs* @ *ab gs bs hs data′*)) **by** *simp*
also have ... = *const-lt-component* ' (*fst* ' *set* (*gs* @ *ab gs bs hs data′*) − {*0*}) − {*None*} (**is** *?A* = *?B*)
proof (*rule card-seteq*[*symmetric*])

show *finite ?A* **by** (*intro finite-imageI finite-Keys*, *fact finite-set*)
next
have *rem-comps-spec* (*gs* @ *ab gs bs hs data′*) (*fst data* − *count-const-lt-components* (*fst aux*), *data′*)
using *assms*(*1*) *rec1.prems*(*3*) *rec1.hyps*(*1*) *rec1.prems*(*1*) *rec1.hyps*(*2*) *aux-def hs-data′*
by (*rule rem-comps-spec-struct*)
also have ... = (*0*, *data′*) **by** (*simp add*: *aux-def rec1.hyps*(*3*))
finally have *card* (*const-lt-component* ‘ (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*) − {*0*}) − {*None*}) =
card (*component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*)))
by (*simp add*: *rem-comps-spec-def*)
also have ... = *card* (*Some* ‘ *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*)))
by (*rule card-image*[*symmetric*], *simp*)
finally show *card ?A* ≤ *card ?B* **by** *simp*
qed (*fact const-lt-component-subset*)
finally have *Some k* ∈ *const-lt-component* ‘ (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*) − {*0*})
by *simp*
then obtain *b* **where** *b* ∈ *fst* ‘ *set* (*gs* @ *ab gs bs hs data′*) **and** *b* ≠ *0*
and ∗: *const-lt-component b* = *Some k* **by** *fastforce*
show ∃ *b*∈*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*). *b* ≠ *0* ∧ *component-of-term* (*lt b*) = *k* ∧ *lp b* = *0*
proof (*intro bexI conjI*)
from ∗ **show** *component-of-term* (*lt b*) = *k* **by** (*rule const-lt-component-SomeD2*)
next
from ∗ **show** *lp b* = *0* **by** (*rule const-lt-component-SomeD1*)
qed *fact*+
qed
next
from *compl* ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*›
have *component-of-term* ‘ *Keys* (*fst* ‘ *set hs*) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *hs-def aux-def fst-set-add-indices* **by** (*rule compl-structD2*)
hence *sub*: *component-of-term* ‘ *Keys* (*fst* ‘ *set hs*) ⊆ *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*))
by (*simp add*: *args-to-set-subset-Times*[*OF rec1.prems*(*1*)] *image-Un*)
have *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*))) =
component-of-term ‘ *Keys* (*fst* ‘ *set* (*gs* @ *bs*)) **by** (*fact components-full-gb*)
also have ... = *component-of-term* ‘ *Keys* (*fst* ‘ *set* ((*gs* @ *bs*) @ *hs*))
by (*simp only*: *set-append*[*of - hs*] *image-Un Keys-Un Un-absorb2 sub*)
finally show *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*full-gb* (*gs* @ *bs*))) =
component-of-term ‘ *Keys* (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*))
by (*simp only*: *eq append-assoc*)
qed
also have ... = *pmdl* (*fst* ‘ *set* (*gs* @ *bs*))
using *assms*(*1*, *2*, *3*) *rec1.hyps*(*1*) *rec1.prems*(*1*, *2*) *rec1.hyps*(*2*) *aux-def*

hs-data′
by (*rule pmdl-struct*)
finally show *?case* **by** *simp*
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
from *rec2*(*4*) **have** *hs*: *hs* = *fst* (*add-indices aux* (*snd data*)) **by** (*metis fstI*)
have *pmdl* (*fst* ‘ *set* (*fst* (*gb-schema-dummy* (*rc*, *data′*) *D′* (*ab gs bs hs data′*) (*ap gs bs* (*ps* −− *sps*) *hs data′*)))) =
pmdl (*fst* ‘ *set* (*gs* @ *ab gs bs hs data′*))
proof (*rule rec2.hyps*(*8*))
from *ap ab rec2.prems*(*1*)
show *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆ *set* (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)
next
from *ab rec2.prems*(*2*) *rec2*(*4*) **show** *unique-idx* (*gs* @ *ab gs bs hs data′*) (*snd* (*rc*, *data′*))
unfolding *snd-conv* **by** (*rule unique-idx-ab*)
next
show *rem-comps-spec* (*gs* @ *ab gs bs hs data′*) (*rc*, *data′*) **unfolding** *rec2.hyps*(*5*)
using *assms*(*1*) *rec2.prems*(*3*) *rec2.hyps*(*1*) *rec2.prems*(*1*) *rec2.hyps*(*2*, *3*, *4*)
by (*rule rem-comps-spec-struct*)
qed
also have ... = *pmdl* (*fst* ‘ *set* (*gs* @ *bs*))
using *assms*(*1*, *2*, *3*) *rec2.hyps*(*1*) *rec2.prems*(*1*, *2*) *rec2.hyps*(*2*, *3*, *4*) **by** (*rule pmdl-struct*)
finally show *?case* **.**
qed
qed

lemma *snd-gb-schema-dummy-subset*:
assumes *struct-spec sel ap ab compl* **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and *D* ⊆ (*set gs* ∪ *set bs*) × (*set gs* ∪ *set bs*) **and** *res* = *gb-schema-dummy data D bs ps*
shows *snd res* ⊆ *set* (*fst res*) × *set* (*fst res*) ∨ (∃ *xs*. *fst* (*res*) = *full-gb xs*)
using *assms*
proof (*induct data D bs ps rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
from *base*(*2*) **show** *?case* **by** (*simp add*: *base*(*3*))
next
case (*rec1 bs ps sps data D*)
have ∃ *xs*. *fst res* = *full-gb xs* **by** (*auto simp*: *rec1*(*6*))
thus *?case* **..**
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
from *assms*(*1*) **have** *ab*: *ab-spec ab* **and** *ap*: *ap-spec ap* **by** (*rule struct-specD*)+
from - - *rec2.prems*(*3*) **show** *?case*
proof (*rule rec2.hyps*(*8*))

from *ap ab rec2.prems*(*1*)
show *set* (*ap gs bs* (*ps* −− *sps*) *hs data*′) ⊆ *set* (*ab gs bs hs data*′) × (*set gs* ∪ *set* (*ab gs bs hs data*′))
by (*rule subset-Times-ap*)
next
from *ab rec2.hyps*(*7*) *rec2.prems*(*1*) *rec2.prems*(*2*)
show *D*′ ⊆ (*set gs* ∪ *set* (*ab gs bs hs data*′)) × (*set gs* ∪ *set* (*ab gs bs hs data*′))
by (*rule discarded-subset*)
qed
qed

lemma *gb-schema-dummy-connectible1*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *dickson-grading d*
and *fst* ' *set gs* ⊆ *dgrad-p-set d m* **and** *is-Groebner-basis* (*fst* ' *set gs*)
and *fst* ' *set bs* ⊆ *dgrad-p-set d m*
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and *unique-idx* (*gs* @ *bs*) (*snd data*)
and ⋀*p q*. *processed* (*p*, *q*) (*gs* @ *bs*) *ps* ⟹ (*p*, *q*) $\notin_p$ *D* ⟹ *fst p* ≠ *0* ⟹ *fst q* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst* ' (*set gs* ∪ *set bs*)) (*fst p*) (*fst q*)
and ¬(∃ *xs*. *fst* (*gb-schema-dummy data D bs ps*) = *full-gb xs*)
assumes *f* ∈ *set* (*fst* (*gb-schema-dummy data D bs ps*))
and *g* ∈ *set* (*fst* (*gb-schema-dummy data D bs ps*))
and (*f*, *g*) $\notin_p$ *snd* (*gb-schema-dummy data D bs ps*)
and *fst f* ≠ *0* **and** *fst g* ≠ *0*
shows *crit-pair-cbelow-on d m* (*fst* ' *set* (*fst* (*gb-schema-dummy data D bs ps*))) (*fst f*) (*fst g*)
using *assms*(*1*, *6*, *7*, *8*, *9*, *10*, *11*, *12*, *13*)
proof (*induct data D bs ps rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
show *?case*
proof (*cases f* ∈ *set gs*)
case *True*
show *?thesis*
proof (*cases g* ∈ *set gs*)
case *True*
note *assms*(*3*, *4*, *5*)
moreover from ‹*f* ∈ *set gs*› **have** *fst f* ∈ *fst* ' *set gs* **by** *simp*
moreover from ‹*g* ∈ *set gs*› **have** *fst g* ∈ *fst* ' *set gs* **by** *simp*
ultimately have *crit-pair-cbelow-on d m* (*fst* ' *set gs*) (*fst f*) (*fst g*)
using *assms*(*14*, *15*) **by** (*rule GB-imp-crit-pair-cbelow-dgrad-p-set*)
moreover have *fst* ' *set gs* ⊆ *fst* ' *set* (*fst* (*gs* @ *bs*, *D*)) **by** *auto*
ultimately show *?thesis* **by** (*rule crit-pair-cbelow-mono*)
next
case *False*
from *this base*(*6*, *7*) **have** *processed* (*g*, *f*) (*gs* @ *bs*) [] **by** (*simp add*: *processed-Nil*)
moreover from *base.prems*(*8*) **have** (*g*, *f*) $\notin_p$ *D* **by** (*simp add*: *in-pair-iff*)

ultimately have *crit-pair-cbelow-on d m* (*fst ' set* (*gs @ bs*)) (*fst g*) (*fst f*)
using ‹*fst g* ≠ *0*› ‹*fst f* ≠ *0*› **unfolding** *set-append* **by** (*rule base*(*4*))
thus *?thesis* **unfolding** *fst-conv* **by** (*rule crit-pair-cbelow-sym*)
qed
next
case *False*
from *this base*(*6*, *7*) **have** *processed* (*f*, *g*) (*gs @ bs*) [] **by** (*simp add*: *processed-Nil*)
moreover from *base.prems*(*8*) **have** (*f*, *g*) $\notin_p$ *D* **by** *simp*
ultimately show *?thesis* **unfolding** *fst-conv set-append* **using** ‹*fst f* ≠ *0*› ‹*fst g* ≠ *0*› **by** (*rule base*(*4*))
qed
next
case (*rec1 bs ps sps data D*)
from *rec1.prems*(*5*) **show** *?case* **by** *auto*
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
from *rec2.hyps*(*4*) **have** *hs*: *hs* = *fst* (*add-indices aux* (*snd data*)) **by** (*metis fstI*)
from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab*
and *compl*: *compl-struct compl*
by (*rule struct-specD1*, *rule struct-specD2*, *rule struct-specD3*, *rule struct-specD4*)
from *sel rec2.hyps*(*1*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *rec2.hyps*(*2*) **by** (*rule sel-specD1*, *rule sel-specD2*)
from *ap ab rec2.prems*(*2*) **have** *ap-sub*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆
set (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)
have *ns-sub*: *fst ' set hs* ⊆ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
from *compl assms*(*3*) ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*›
show *dgrad-p-set-le d* (*fst ' set hs*) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *hs rec2.hyps*(*3*) *fst-set-add-indices* **by** (*rule compl-structD1*)
next
from *assms*(*4*) *rec2.prems*(*1*) **show** *args-to-set* (*gs*, *bs*, *ps*) ⊆ *dgrad-p-set d m*
by (*simp add*: *args-to-set-subset-Times*[*OF rec2.prems*(*2*)])
qed
with *rec2.prems*(*1*) **have** *ab-sub*: *fst ' set* (*ab gs bs hs data′*) ⊆ *dgrad-p-set d m*
by (*auto simp add*: *ab-specD1*[*OF ab*])

have *cpq*: (*p*, *q*) $\in_p$ *set sps* ⟹ *fst p* ≠ *0* ⟹ *fst q* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst '* (*set gs* ∪ *set* (*ab gs bs hs data′*))) (*fst p*) (*fst q*) **for** *p q*
proof −
assume (*p*, *q*) $\in_p$ *set sps* **and** *fst p* ≠ *0* **and** *fst q* ≠ *0*
from *this*(*1*) **have** (*p*, *q*) ∈ *set sps* ∨ (*q*, *p*) ∈ *set sps* **by** (*simp only*: *in-pair-iff*)
hence *crit-pair-cbelow-on d m* (*fst '* (*set gs* ∪ *set bs*) ∪ *fst ' set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*))))
(*fst p*) (*fst q*)
proof

assume (*p*, *q*) ∈ *set sps*
from *assms*(*2*, *3*, *4*, *5*) *rec2.prems*(*1*, *2*) ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*› *rec2.prems*(*3*) *this*
‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*› **show** *?thesis* **by** (*rule compl-connD*)
next
assume (*q*, *p*) ∈ *set sps*
from *assms*(*2*, *3*, *4*, *5*) *rec2.prems*(*1*, *2*) ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*› *rec2.prems*(*3*) *this*
‹*fst q* ≠ *0*› ‹*fst p* ≠ *0*›
have *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*))))
(*fst q*) (*fst p*) **by** (*rule compl-connD*)
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
thus *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set* (*ab gs bs hs data*′))) (*fst p*) (*fst q*)
by (*simp add*: *ab-specD1*[*OF ab*] *hs rec2.hyps*(*3*) *fst-set-add-indices image-Un Un-assoc*)
qed

from *ab-sub ap-sub* - - *rec2.prems*(*5*, *6*, *7*, *8*) **show** *?case*
proof (*rule rec2.hyps*(*8*))
from *ab rec2.prems*(*3*) *rec2*(*4*) **show** *unique-idx* (*gs* @ *ab gs bs hs data*′) (*snd* (*rc*, *data*′))
unfolding *snd-conv* **by** (*rule unique-idx-ab*)
next
fix *p q* :: (′*t*, ′*b*, ′*c*) *pdata*
define *ps*′ **where** *ps*′ = *ap gs bs* (*ps* −− *sps*) *hs data*′
assume *fst p* ≠ *0* **and** *fst q* ≠ *0* **and** (*p*, *q*) $\notin_p$ *D*′
assume *processed* (*p*, *q*) (*gs* @ *ab gs bs hs data*′) *ps*′
hence *p-in*: *p* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *q-in*: *q* ∈ *set gs* ∪ *set bs* ∪ *set hs*
and (*p*, *q*) $\notin_p$ *set ps*′ **by** (*simp-all add*: *processed-alt ab-specD1*[*OF ab*])
from *this*(*3*) ‹(*p*, *q*) $\notin_p$ *D*′› **have** (*p*, *q*) $\notin_p$ *D* **and** (*p*, *q*) $\notin_p$ *set* (*ps* −− *sps*)
and (*p*, *q*) $\notin_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*auto simp*: *in-pair-iff rec2.hyps*(*7*) *ps*′*-def*)
from *this*(*3*) *p-in q-in* **have** *p* ∈ *set gs* ∪ *set bs* **and** *q* ∈ *set gs* ∪ *set bs*
by (*meson SigmaI UnE in-pair-iff*)+
show *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set* (*ab gs bs hs data*′))) (*fst p*) (*fst q*)
proof (*cases component-of-term* (*lt* (*fst p*)) = *component-of-term* (*lt* (*fst q*)))
case *True*
show *?thesis*
proof (*cases* (*p*, *q*) $\in_p$ *set sps*)
case *True*
from *this* ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*› **show** *?thesis* **by** (*rule cpq*)
next
case *False*
with ‹(*p*, *q*) $\notin_p$ *set* (*ps* −− *sps*)› **have** (*p*, *q*) $\notin_p$ *set ps*
by (*auto simp*: *in-pair-iff set-diff-list*)

with ‹$p \in$ *set gs* $\cup$ *set bs*› ‹$q \in$ *set gs* $\cup$ *set bs*› **have** *processed* (p, q) (*gs* @ *bs*) *ps*
by (*simp add*: *processed-alt*)
from *this* ‹$(p, q) \notin_p D$› ‹*fst p* $\neq$ *0*› ‹*fst q* $\neq$ *0*›
have *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* $\cup$ *set bs*)) (*fst p*) (*fst q*)
by (*rule rec2.prems*(*4*))
moreover have *fst* ‘ (*set gs* $\cup$ *set bs*) $\subseteq$ *fst* ‘ (*set gs* $\cup$ *set* (*ab gs bs hs data′*))
by (*auto simp*: *ab-specD1*[*OF ab*])
ultimately show *?thesis* **by** (*rule crit-pair-cbelow-mono*)
qed
next
case *False*
thus *?thesis* **by** (*rule crit-pair-cbelow-distinct-component*)
qed
qed
qed

lemma *gb-schema-dummy-connectible2*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *dickson-grading d*
and *fst* ‘ *set gs* $\subseteq$ *dgrad-p-set d m* **and** *is-Groebner-basis* (*fst* ‘ *set gs*)
and *fst* ‘ *set bs* $\subseteq$ *dgrad-p-set d m*
and *set ps* $\subseteq$ *set bs* $\times$ (*set gs* $\cup$ *set bs*) **and** $D \subseteq$ (*set gs* $\cup$ *set bs*) $\times$ (*set gs* $\cup$ *set bs*)
and *set ps* $\cap_p D = \{\}$ **and** *unique-idx* (*gs* @ *bs*) (*snd data*)
and $\bigwedge B\ a\ b.$ *set gs* $\cup$ *set bs* $\subseteq B \Longrightarrow$ *fst* ‘ $B \subseteq$ *dgrad-p-set d m* $\Longrightarrow (a, b) \in_p D \Longrightarrow$
fst a $\neq$ *0* $\Longrightarrow$ *fst b* $\neq$ *0* $\Longrightarrow$
($\bigwedge x\ y.\ x \in$ *set gs* $\cup$ *set bs* $\Longrightarrow y \in$ *set gs* $\cup$ *set bs* $\Longrightarrow \neg\ (x, y) \in_p D \Longrightarrow$
fst x $\neq$ *0* $\Longrightarrow$ *fst y* $\neq$ *0* $\Longrightarrow$ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst x*) (*fst y*)) $\Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst a*) (*fst b*)
and $\bigwedge x\ y.\ x \in$ *set* (*fst* (*gb-schema-dummy data D bs ps*)) $\Longrightarrow y \in$ *set* (*fst* (*gb-schema-dummy data D bs ps*)) $\Longrightarrow$
$(x, y) \notin_p$ *snd* (*gb-schema-dummy data D bs ps*) $\Longrightarrow$ *fst x* $\neq$ *0* $\Longrightarrow$ *fst y* $\neq$ *0* $\Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *set* (*fst* (*gb-schema-dummy data D bs ps*))) (*fst x*) (*fst y*)
and $\neg(\exists$ *xs*. *fst* (*gb-schema-dummy data D bs ps*) = *full-gb xs*)
assumes $(f, g) \in_p$ *snd* (*gb-schema-dummy data D bs ps*)
and *fst f* $\neq$ *0* **and** *fst g* $\neq$ *0*
shows *crit-pair-cbelow-on d m* (*fst* ‘ *set* (*fst* (*gb-schema-dummy data D bs ps*))) (*fst f*) (*fst g*)
using *assms*(*1*, *6*, *7*, *8*, *9*, *10*, *11*, *12*, *13*, *14*, *15*, *16*)
proof (*induct data D bs ps rule*: *gb-schema-dummy-induct*)
case (*base bs data D*)
have *set gs* $\cup$ *set bs* $\subseteq$ *set* (*fst* (*gs* @ *bs*, *D*)) **by** *simp*
moreover from *assms*(*4*) *base.prems*(*1*) **have** *fst* ‘ *set* (*fst* (*gs* @ *bs*, *D*)) $\subseteq$

dgrad-p-set d m **by** *auto*
  moreover from *base.prems*(*9*) **have** (*f*, *g*) $\in_p$ *D* **by** *simp*
  moreover note *assms*(*15*, *16*)
  ultimately show *?case*
  proof (*rule base.prems*(*6*))
    fix *x y*
    assume *x* ∈ *set gs* ∪ *set bs* **and** *y* ∈ *set gs* ∪ *set bs* **and** (*x*, *y*) $\notin_p$ *D*
    hence *x* ∈ *set* (*fst* (*gs* @ *bs*, *D*)) **and** *y* ∈ *set* (*fst* (*gs* @ *bs*, *D*)) **and** (*x*, *y*) $\notin_p$ *snd* (*gs* @ *bs*, *D*)
      by *simp-all*
    moreover assume *fst x* ≠ *0* **and** *fst y* ≠ *0*
    ultimately show *crit-pair-cbelow-on d m* (*fst* ‘ *set* (*fst* (*gs* @ *bs*, *D*))) (*fst x*) (*fst y*)
      by (*rule base.prems*(*7*))
  qed
next
  case (*rec1 bs ps sps data D*)
  from *rec1.prems*(*8*) **show** *?case* **by** *auto*
next
  case (*rec2 bs ps sps aux hs rc data data′ D D′*)
  from *rec2.hyps*(*4*) **have** *hs*: *hs* = *fst* (*add-indices aux* (*snd data*)) **by** (*metis fstI*)
  from *assms*(*1*) **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab*
    and *compl*: *compl-struct compl* **by** (*rule struct-specD*)+

  let *?X* = *set* (*ps* −− *sps*) ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)

  from *sel rec2.hyps*(*1*) **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
    unfolding *rec2.hyps*(*2*) **by** (*rule sel-specD1*, *rule sel-specD2*)

  have *fst* ‘ *set hs* ∩ *fst* ‘ (*set gs* ∪ *set bs*) = {}
    unfolding *hs fst-set-add-indices rec2.hyps*(*3*) **using** *compl* ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*›
    by (*rule compl-struct-disjoint*)
  hence *disj1*: (*set gs* ∪ *set bs*) ∩ *set hs* = {} **by** *fastforce*

  have *disj2*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) $\cap_p$ *D′* = {}
  proof (*rule*, *rule*)
    fix *x y*
    assume (*x*, *y*) ∈ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) $\cap_p$ *D′*
    hence (*x*, *y*) $\in_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) $\cap_p$ *D′* **by** (*simp add*: *in-pair-alt*)
    hence *1*: (*x*, *y*) $\in_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) **and** (*x*, *y*) $\in_p$ *D′* **by** *simp-all*
    hence (*x*, *y*) $\in_p$ *D* **by** (*simp add*: *rec2.hyps*(*7*))
    from *this rec2.prems*(*3*) **have** *x* ∈ *set gs* ∪ *set bs* **and** *y* ∈ *set gs* ∪ *set bs*
      by (*auto simp*: *in-pair-iff*)
    from *1 ap-specD1*[*OF ap*] **have** (*x*, *y*) $\in_p$ *?X* **by** (*rule in-pair-trans*)
    thus (*x*, *y*) ∈ {} **unfolding** *in-pair-Un*
    proof

assume $(x, y) \in_p$ *set* (*ps* −− *sps*)
also have ... ⊆ *set ps* **by** (*auto simp*: *set-diff-list*)
finally have $(x, y) \in_p$ *set ps* $\cap_p$ *D* **using** ‹$(x, y) \in_p D$› **by** *simp*
also have ... = {} **by** (*fact rec2.prems*(*4*))
finally show *?thesis* **by** (*simp add*: *in-pair-iff*)
next
assume $(x, y) \in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
hence *x* ∈ *set hs* ∨ *y* ∈ *set hs* **by** (*auto simp*: *in-pair-iff*)
thus *?thesis*
proof
assume *x* ∈ *set hs*
with ‹*x* ∈ *set gs* ∪ *set bs*› **have** *x* ∈ (*set gs* ∪ *set bs*) ∩ *set hs* **..**
thus *?thesis* **by** (*simp add*: *disj1*)
next
assume *y* ∈ *set hs*
with ‹*y* ∈ *set gs* ∪ *set bs*› **have** *y* ∈ (*set gs* ∪ *set bs*) ∩ *set hs* **..**
thus *?thesis* **by** (*simp add*: *disj1*)
qed
qed
qed *simp*

have *hs-sub*: *fst* ‘ *set hs* ⊆ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
from *compl assms*(*3*) ‹*sps* ≠ []› ‹*set sps* ⊆ *set ps*›
show *dgrad-p-set-le d* (*fst* ‘ *set hs*) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *hs rec2.hyps*(*3*) *fst-set-add-indices* **by** (*rule compl-structD1*)
next
from *assms*(*4*) *rec2.prems*(*1*) **show** *args-to-set* (*gs*, *bs*, *ps*) ⊆ *dgrad-p-set d m*
by (*simp add*: *args-to-set-subset-Times*[*OF rec2.prems*(*2*)])
qed
with *rec2.prems*(*1*) **have** *ab-sub*: *fst* ‘ *set* (*ab gs bs hs data′*) ⊆ *dgrad-p-set d m*
by (*auto simp add*: *ab-specD1*[*OF ab*])

moreover from *ap ab rec2.prems*(*2*)
have *ap-sub*: *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*) ⊆ *set* (*ab gs bs hs data′*) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule subset-Times-ap*)

moreover from *ab rec2.hyps*(*7*) *rec2.prems*(*2*) *rec2.prems*(*3*)
have *D′* ⊆ (*set gs* ∪ *set* (*ab gs bs hs data′*)) × (*set gs* ∪ *set* (*ab gs bs hs data′*))
by (*rule discarded-subset*)

moreover note *disj2*

moreover from *ab rec2.prems*(*5*) *rec2.hyps*(*4*) **have** *uid*: *unique-idx* (*gs* @ *ab gs bs hs data′*) (*snd* (*rc*, *data′*))
unfolding *snd-conv* **by** (*rule unique-idx-ab*)

ultimately show *?case* **using** - - *rec2.prems*(*8*, *9*, *10*, *11*)

proof (*rule rec2.hyps*(*8*), *simp only*: *ab-specD1*[*OF ab*] *Un-assoc*[*symmetric*])
define *ps'* **where** *ps'* = *ap gs bs* (*ps* −− *sps*) *hs data'*
fix *B a b*
assume *B-sup*: *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B*
hence *set gs* ∪ *set bs* ⊆ *B* **and** *set hs* ⊆ *B* **by** *simp-all*
assume (*a*, *b*) $\in_p$ *D'*
hence *ab-cases*: (*a*, *b*) $\in_p$ *D* ∨ (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) $-_p$ *set ps'* ∨
(*a*, *b*) $\in_p$ *set* (*ps* −− *sps*) $-_p$ *set ps'* **by** (*auto simp*: *rec2.hyps*(*7*) *ps'-def*)
assume *B-sub*: *fst* ' *B* ⊆ *dgrad-p-set d m* **and** *fst a* ≠ *0* **and** *fst b* ≠ *0*
assume ∗: ⋀*x y*. *x* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹ *y* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹
(*x*, *y*) $\notin_p$ *D'* ⟹ *fst x* ≠ *0* ⟹ *fst y* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst* ' *B*) (*fst x*) (*fst y*)

from *rec2.prems*(*2*) **have** *ps-sps-sub*: *set* (*ps* −− *sps*) ⊆ *set bs* × (*set gs* ∪ *set bs*)
by (*auto simp*: *set-diff-list*)
from *uid* **have** *uid'*: *unique-idx* (*gs* @ *bs* @ *hs*) *data'* **by** (*simp add*: *unique-idx-def ab-specD1*[*OF ab*])

have *a*: *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst x*) (*fst y*)
if *fst x* ≠ *0* **and** *fst y* ≠ *0* **and** *xy-in*: (*x*, *y*) $\in_p$ *set* (*ps* −− *sps*) $-_p$ *set ps'* **for** *x y*
proof (*cases x* = *y*)
case *True*
from *xy-in rec2.prems*(*2*) **have** *y* ∈ *set gs* ∪ *set bs*
unfolding *in-pair-minus-pairs* **unfolding** *True in-pair-iff set-diff-list* **by** *auto*
hence *fst y* ∈ *fst* ' *set gs* ∪ *fst* ' *set bs* **by** *fastforce*
from *this assms*(*4*) *rec2.prems*(*1*) **have** *fst y* ∈ *dgrad-p-set d m* **by** *blast*
with *assms*(*3*) **show** *?thesis* **unfolding** *True* **by** (*rule crit-pair-cbelow-same*)
next
case *False*
from *ap assms*(*3*) *B-sup B-sub ps-sps-sub disj1 uid' assms*(*5*) *False* ‹*fst x* ≠ *0*› ‹*fst y* ≠ *0*› *xy-in*
show *?thesis* **unfolding** *ps'-def*
proof (*rule ap-specD3*)
fix *a1 b1* :: (*'t*, *'b*, *'c*) *pdata*
assume *fst a1* ≠ *0* **and** *fst b1* ≠ *0*
assume *a1* ∈ *set hs* **and** *b1-in*: *b1* ∈ *set gs* ∪ *set bs* ∪ *set hs*
hence *a1-in*: *a1* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *fastforce*
assume (*a1*, *b1*) $\in_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data'*)
hence (*a1*, *b1*) $\in_p$ *set ps'* **by** (*simp only*: *ps'-def*)
with *disj2* **have** (*a1*, *b1*) $\notin_p$ *D'* **unfolding** *ps'-def*
by (*metis empty-iff in-pair-Int-pairs in-pair-alt*)
with *a1-in b1-in* **show** *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst a1*) (*fst b1*)
using ‹*fst a1* ≠ *0*› ‹*fst b1* ≠ *0*› **by** (*rule* ∗)

qed
qed

have *b*: *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst x*) (*fst y*)
if (*x*, *y*) $\in_p$ *D* **and** *fst x* ≠ *0* **and** *fst y* ≠ *0* **for** *x y*
using ‹*set gs* ∪ *set bs* ⊆ *B*› *B-sub that*
proof (*rule rec2.prems*(*6*))
fix *a1 b1* :: (′*t*, ′*b*, ′*c*) *pdata*
assume *a1* ∈ *set gs* ∪ *set bs* **and** *b1* ∈ *set gs* ∪ *set bs*
hence *a1-in*: *a1* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *b1-in*: *b1* ∈ *set gs* ∪ *set bs* ∪ *set hs*
by *fastforce*+
assume (*a1*, *b1*) $\notin_p$ *D* **and** *fst a1* ≠ *0* **and** *fst b1* ≠ *0*
show *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst a1*) (*fst b1*)
proof (*cases* (*a1*, *b1*) $\in_p$ *?X* $-_p$ *set ps*′)
case *True*
moreover from ‹*a1* ∈ *set gs* ∪ *set bs*› ‹*b1* ∈ *set gs* ∪ *set bs*› *disj1*
have (*a1*, *b1*) $\notin_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*auto simp*: *in-pair-def*)
ultimately have (*a1*, *b1*) $\in_p$ *set* (*ps* −− *sps*) $-_p$ *set ps*′ **by** *auto*
with ‹*fst a1* ≠ *0*› ‹*fst b1* ≠ *0*› **show** *?thesis* **by** (*rule a*)
next
case *False*
with ‹(*a1*, *b1*) $\notin_p$ *D*› **have** (*a1*, *b1*) $\notin_p$ *D*′ **by** (*auto simp*: *rec2.hyps*(*7*) *ps*′*-def*)
with *a1-in b1-in* **show** *?thesis* **using** ‹*fst a1* ≠ *0*› ‹*fst b1* ≠ *0*› **by** (*rule* ∗)
qed
qed

have *c*: *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst x*) (*fst y*)
if *x-in*: *x* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *y-in*: *y* ∈ *set gs* ∪ *set bs* ∪ *set hs*
and *xy*: (*x*, *y*) $\notin_p$ (*?X* $-_p$ *set ps*′) **and** *fst x* ≠ *0* **and** *fst y* ≠ *0* **for** *x y*
proof (*cases* (*x*, *y*) $\in_p$ *D*)
case *True*
thus *?thesis* **using** ‹*fst x* ≠ *0*› ‹*fst y* ≠ *0*› **by** (*rule b*)
next
case *False*
with *xy* **have** (*x*, *y*) $\notin_p$ *D*′ **unfolding** *rec2.hyps*(*7*) *ps*′*-def* **by** *auto*
with *x-in y-in* **show** *?thesis* **using** ‹*fst x* ≠ *0*› ‹*fst y* ≠ *0*› **by** (*rule* ∗)
qed

from *ab-cases* **show** *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst a*) (*fst b*)
proof (*elim disjE*)
assume (*a*, *b*) $\in_p$ *D*
thus *?thesis* **using** ‹*fst a* ≠ *0*› ‹*fst b* ≠ *0*› **by** (*rule b*)
next
assume *ab-in*: (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) $-_p$ *set ps*′
hence *ab-in*′: (*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **and** (*a*, *b*) $\notin_p$ *set ps*′ **by** *simp-all*

show *?thesis*
proof (*cases a* = *b*)
case *True*
from *ab-in′ rec2.prems*(*2*) **have** *b* ∈ *set hs* **unfolding** *True in-pair-iff set-diff-list* **by** *auto*
hence *fst b* ∈ *fst ‘ set hs* **by** *fastforce*
from *this hs-sub* **have** *fst b* ∈ *dgrad-p-set d m* **..**
with *assms*(*3*) **show** *?thesis* **unfolding** *True* **by** (*rule crit-pair-cbelow-same*)
next
case *False*
from *ap assms*(*3*) *B-sup B-sub ab-in′ ps-sps-sub uid′ assms*(*5*) *False* ‹*fst a* ≠ *0*› ‹*fst b* ≠ *0*›
show *?thesis*
proof (*rule ap-specD2*)
fix *x y* :: (*′t*, *′b*, *′c*) *pdata*
assume (*x*, *y*) $\in_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)
also from *ap-sub* **have** ... ⊆ (*set bs* ∪ *set hs*) × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*simp only*: *ab-specD1*[*OF ab*] *Un-assoc*)
also have ... ⊆ (*set gs* ∪ *set bs* ∪ *set hs*) × (*set gs* ∪ *set bs* ∪ *set hs*) **by** *fastforce*
finally have (*x*, *y*) ∈ (*set gs* ∪ *set bs* ∪ *set hs*) × (*set gs* ∪ *set bs* ∪ *set hs*)
unfolding *in-pair-same* **.**
hence *x* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *y* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
moreover from ‹(*x*, *y*) $\in_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)› **have** (*x*, *y*) $\notin_p$ *?X* $-_p$ *set ps′*
by (*simp add*: *ps′-def*)
moreover assume *fst x* ≠ *0* **and** *fst y* ≠ *0*
ultimately show *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst x*) (*fst y*) **by** (*rule c*)
next
fix *x y* :: (*′t*, *′b*, *′c*) *pdata*
assume *fst x* ≠ *0* **and** *fst y* ≠ *0*
assume *1*: *x* ∈ *set gs* ∪ *set bs* **and** *2*: *y* ∈ *set gs* ∪ *set bs*
hence *x-in*: *x* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *y-in*: *y* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
show *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst x*) (*fst y*)
proof (*cases* (*x*, *y*) $\in_p$ *set* (*ps* −− *sps*) $-_p$ *set ps′*)
case *True*
with ‹*fst x* ≠ *0*› ‹*fst y* ≠ *0*› **show** *?thesis* **by** (*rule a*)
next
case *False*
have (*x*, *y*) $\notin_p$ *set* (*ps* −− *sps*) ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) $-_p$ *set ps′*
proof
assume (*x*, *y*) $\in_p$ *set* (*ps* −− *sps*) ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) $-_p$ *set ps′*
hence (*x*, *y*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **using** *False*
by *simp*

```
            hence x ∈ set hs ∨ y ∈ set hs by (auto simp: in-pair-iff)
            with 1 2 disj1 show False by blast
          qed
          with x-in y-in show ?thesis using ‹fst x ≠ 0› ‹fst y ≠ 0› by (rule c)
        qed
      qed
    qed
  next
    assume (a, b) ∈p set (ps −− sps) −p set ps'
    with ‹fst a ≠ 0› ‹fst b ≠ 0› show ?thesis by (rule a)
  qed
next
  fix x y :: ('t, 'b, 'c) pdata
  let ?res = gb-schema-dummy (rc, data') D' (ab gs bs hs data') (ap gs bs (ps
−− sps) hs data')
  assume x ∈ set (fst ?res) and y ∈ set (fst ?res) and (x, y) ∉p snd ?res and
fst x ≠ 0 and fst y ≠ 0
  thus crit-pair-cbelow-on d m (fst ` set (fst ?res)) (fst x) (fst y) by (rule
rec2.prems(7))
qed
qed

corollary gb-schema-dummy-connectible:
  assumes struct-spec sel ap ab compl and compl-conn compl and dickson-grading
d
    and fst ` set gs ⊆ dgrad-p-set d m and is-Groebner-basis (fst ` set gs)
    and fst ` set bs ⊆ dgrad-p-set d m
    and set ps ⊆ set bs × (set gs ∪ set bs) and D ⊆ (set gs ∪ set bs) × (set gs ∪
set bs)
    and set ps ∩p D = {} and unique-idx (gs @ bs) (snd data)
    and ⋀p q. processed (p, q) (gs @ bs) ps ⟹ (p, q) ∉p D ⟹ fst p ≠ 0 ⟹ fst
q ≠ 0 ⟹
            crit-pair-cbelow-on d m (fst ` (set gs ∪ set bs)) (fst p) (fst q)
    and ⋀B a b. set gs ∪ set bs ⊆ B ⟹ fst ` B ⊆ dgrad-p-set d m ⟹ (a, b) ∈p
D ⟹
            fst a ≠ 0 ⟹ fst b ≠ 0 ⟹
            (⋀x y. x ∈ set gs ∪ set bs ⟹ y ∈ set gs ∪ set bs ⟹ ¬ (x, y) ∈p D ⟹
              fst x ≠ 0 ⟹ fst y ≠ 0 ⟹ crit-pair-cbelow-on d m (fst ` B) (fst x)
(fst y)) ⟹
            crit-pair-cbelow-on d m (fst ` B) (fst a) (fst b)
  assumes f ∈ set (fst (gb-schema-dummy data D bs ps))
    and g ∈ set (fst (gb-schema-dummy data D bs ps))
    and fst f ≠ 0 and fst g ≠ 0
  shows crit-pair-cbelow-on d m (fst ` set (fst (gb-schema-dummy data D bs ps)))
(fst f) (fst g)
proof (cases ∃xs. fst (gb-schema-dummy data D bs ps) = full-gb xs)
  case True
  then obtain xs where xs: fst (gb-schema-dummy data D bs ps) = full-gb xs ..
  note assms(3)
```

moreover have *fst ‘ set* (*full-gb xs*) ⊆ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
have *dgrad-p-set-le d* (*fst ‘ set* (*full-gb xs*)) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *xs*[*symmetric*] **using** *assms*(*3*, *1*) **by** (*rule fst-gb-schema-dummy-dgrad-p-set-le*)
also from *assms*(*7*) **have** ... = *fst ‘ set gs* ∪ *fst ‘ set bs* **by** (*rule args-to-set-subset-Times*)
finally show *dgrad-p-set-le d* (*fst ‘ set* (*full-gb xs*)) (*fst ‘ set gs* ∪ *fst ‘ set bs*) **.**
next
from *assms*(*4*, *6*) **show** *fst ‘ set gs* ∪ *fst ‘ set bs* ⊆ *dgrad-p-set d m* **by** *blast*
qed
moreover note *full-gb-isGB*
moreover from *assms*(*13*) **have** *fst f* ∈ *fst ‘ set* (*full-gb xs*) **by** (*simp add*: *xs*)
moreover from *assms*(*14*) **have** *fst g* ∈ *fst ‘ set* (*full-gb xs*) **by** (*simp add*: *xs*)
ultimately show *?thesis* **using** *assms*(*15*, *16*) **unfolding** *xs*
by (*rule GB-imp-crit-pair-cbelow-dgrad-p-set*)
next
case *not-full*: *False*
show *?thesis*
proof (*cases* (*f*, *g*) $\in_p$ *snd* (*gb-schema-dummy data D bs ps*))
case *True*
from *assms*(*1*−*10*,*12*) - *not-full True assms*(*15*,*16*) **show** *?thesis*
proof (*rule gb-schema-dummy-connectible2*)
fix *x y*
assume *x* ∈ *set* (*fst* (*gb-schema-dummy data D bs ps*))
and *y* ∈ *set* (*fst* (*gb-schema-dummy data D bs ps*))
and (*x*, *y*) $\notin_p$ *snd* (*gb-schema-dummy data D bs ps*)
and *fst x* ≠ *0* **and** *fst y* ≠ *0*
with *assms*(*1*−*7*,*10*,*11*) *not-full*
show *crit-pair-cbelow-on d m* (*fst ‘ set* (*fst* (*gb-schema-dummy data D bs ps*))) (*fst x*) (*fst y*)
by (*rule gb-schema-dummy-connectible1*)
qed
next
case *False*
from *assms*(*1*−*7*,*10*,*11*) *not-full assms*(*13*,*14*) *False assms*(*15*,*16*) **show** *?thesis*
by (*rule gb-schema-dummy-connectible1*)
qed
qed

lemma *fst-gb-schema-dummy-dgrad-p-set-le-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
shows *dgrad-p-set-le d* (*fst ‘ set* (*fst* (*gb-schema-dummy data D* (*ab gs* [] *bs* (*snd data*)) (*ap gs* [] [] *bs* (*snd data*)))))
(*fst ‘* (*set gs* ∪ *set bs*))
proof −
let *?bs* = *ab gs* [] *bs* (*snd data*)
from *assms*(*2*) **have** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
from *ap-specD1*[*OF ap*, *of gs* [] [] *bs*]
have ∗: *set* (*ap gs* [] [] *bs* (*snd data*)) ⊆ *set ?bs* × (*set gs* ∪ *set ?bs*)
by (*simp add*: *ab-specD1*[*OF ab*])

from *assms* **have** *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*gb-schema-dummy data D ?bs* (*ap gs* [] [] *bs* (*snd data*)))))
(*args-to-set* (*gs*, *?bs*, (*ap gs* [] [] *bs* (*snd data*))))
by (*rule fst-gb-schema-dummy-dgrad-p-set-le*)
also have ... = *fst* ' (*set gs* ∪ *set bs*)
by (*simp add*: *args-to-set-subset-Times*[*OF* ∗] *image-Un ab-specD1*[*OF ab*])
finally show *?thesis* **.**
qed

corollary *fst-gb-schema-dummy-dgrad-p-set-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
and *fst* ' (*set gs* ∪ *set bs*) ⊆ *dgrad-p-set d m*
shows *fst* ' *set* (*fst* (*gb-schema-dummy* (*rc*, *data*) *D* (*ab gs* [] *bs data*) (*ap gs* [] [] *bs data*))) ⊆ *dgrad-p-set d m*
proof (*rule dgrad-p-set-le-dgrad-p-set*)
let *?data* = (*rc*, *data*)
from *assms*(*1*, *2*)
have *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*gb-schema-dummy ?data D* (*ab gs* [] *bs* (*snd ?data*)) (*ap gs* [] [] *bs* (*snd ?data*)))))
(*fst* ' (*set gs* ∪ *set bs*))
by (*rule fst-gb-schema-dummy-dgrad-p-set-le-init*)
thus *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*gb-schema-dummy ?data D* (*ab gs* [] *bs data*) (*ap gs* [] [] *bs data*))))
(*fst* ' (*set gs* ∪ *set bs*))
by (*simp only*: *snd-conv*)
qed *fact*

lemma *fst-gb-schema-dummy-components-init*:
fixes *bs data*
defines *bs0* ≡ *ab gs* [] *bs data*
defines *ps0* ≡ *ap gs* [] [] *bs data*
assumes *struct-spec sel ap ab compl*
shows *component-of-term* ' *Keys* (*fst* ' *set* (*fst* (*gb-schema-dummy* (*rc*, *data*) *D bs0 ps0*))) =
component-of-term ' *Keys* (*fst* ' *set* (*gs* @ *bs*)) (**is** *?l* = *?r*)
proof −
from *assms*(*3*) **have** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
from *ap-specD1*[*OF ap*, *of gs* [] [] *bs*]
have ∗: *set ps0* ⊆ *set bs0* × (*set gs* ∪ *set bs0*) **by** (*simp add*: *ps0-def bs0-def ab-specD1*[*OF ab*])
with *assms*(*3*) **have** *?l* = *component-of-term* ' *Keys* (*args-to-set* (*gs*, *bs0*, *ps0*))
by (*rule fst-gb-schema-dummy-components*)
also have ... = *?r*
by (*simp only*: *args-to-set-subset-Times*[*OF* ∗], *simp add*: *ab-specD1*[*OF ab*] *bs0-def image-Un*)
finally show *?thesis* **.**
qed

lemma *fst-gb-schema-dummy-pmdl-init*:

fixes *bs data*
defines *bs0* ≡ *ab gs* [] *bs data*
defines *ps0* ≡ *ap gs* [] [] *bs data*
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis* (*fst* ' *set gs*)
and *unique-idx* (*gs* @ *bs0*) *data* **and** *rem-comps-spec* (*gs* @ *bs0*) (*rc*, *data*)
shows *pmdl* (*fst* ' *set* (*fst* (*gb-schema-dummy* (*rc*, *data*) *D bs0 ps0*))) =
pmdl (*fst* ' (*set* (*gs* @ *bs*))) (**is** *?l* = *?r*)
proof −
from *assms*(*3*) **have** *ab*: *ab-spec ab* **by** (*rule struct-specD3*)
let *?data* = (*rc*, *data*)
from *assms*(*6*) **have** *unique-idx* (*gs* @ *bs0*) (*snd ?data*) **by** (*simp only*: *snd-conv*)
from *assms*(*3*, *4*, *5*) - *this assms*(*7*) **have** *?l* = *pmdl* (*fst* ' (*set* (*gs* @ *bs0*)))
proof (*rule fst-gb-schema-dummy-pmdl*)
from *assms*(*3*) **have** *ap-spec ap* **by** (*rule struct-specD2*)
from *ap-specD1*[*OF this*, *of gs* [] [] *bs*]
show *set ps0* ⊆ *set bs0* × (*set gs* ∪ *set bs0*) **by** (*simp add*: *ps0-def bs0-def ab-specD1*[*OF ab*])
qed
also have ... = *?r* **by** (*simp add*: *bs0-def ab-specD1*[*OF ab*])
finally show *?thesis* **.**
qed

lemma *fst-gb-schema-dummy-isGB-init*:
fixes *bs data*
defines *bs0* ≡ *ab gs* [] *bs data*
defines *ps0* ≡ *ap gs* [] [] *bs data*
defines *D0* ≡ *set bs* × (*set gs* ∪ *set bs*) $-_p$ *set ps0*
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *is-Groebner-basis* (*fst* ' *set gs*)
and *unique-idx* (*gs* @ *bs0*) *data* **and** *rem-comps-spec* (*gs* @ *bs0*) (*rc*, *data*)
shows *is-Groebner-basis* (*fst* ' *set* (*fst* (*gb-schema-dummy* (*rc*, *data*) *D0 bs0 ps0*)))
proof −
let *?data* = (*rc*, *data*)
let *?res* = *gb-schema-dummy ?data D0 bs0 ps0*
from *assms*(*4*) **have** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD2*, *rule struct-specD3*)
have *set-bs0*: *set bs0* = *set bs* **by** (*simp add*: *bs0-def ab-specD1*[*OF ab*])
from *ap-specD1*[*OF ap*, *of gs* [] [] *bs*] **have** *ps0-sub*: *set ps0* ⊆ *set bs0* × (*set gs* ∪ *set bs0*)
by (*simp add*: *ps0-def set-bs0*)
from *ex-dgrad* **obtain** *d*::*'a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
have *finite* (*fst* ' (*set gs* ∪ *set bs*)) **by** (*rule*, *rule finite-UnI*, *fact finite-set*, *fact finite-set*)
then obtain *m* **where** *gs-bs-sub*: *fst* ' (*set gs* ∪ *set bs*) ⊆ *dgrad-p-set d m* **by** (*rule dgrad-p-set-exhaust*)
with *dg assms*(*4*) **have** *fst* ' *set* (*fst ?res*) ⊆ *dgrad-p-set d m* **unfolding** *bs0-def ps0-def*
by (*rule fst-gb-schema-dummy-dgrad-p-set-init*)

```
  with dg show ?thesis
  proof (rule crit-pair-cbelow-imp-GB-dgrad-p-set)
    fix p0 q0
    assume p0-in: p0 ∈ fst ` set (fst ?res) and q0-in: q0 ∈ fst ` set (fst ?res)
    assume p0 ≠ 0 and q0 ≠ 0
    from ‹fst ` (set gs ∪ set bs) ⊆ dgrad-p-set d m›
    have fst ` set gs ⊆ dgrad-p-set d m and fst ` set bs ⊆ dgrad-p-set d m
      by (simp-all add: image-Un)
    from p0-in obtain p where p-in: p ∈ set (fst ?res) and p0: p0 = fst p ..
    from q0-in obtain q where q-in: q ∈ set (fst ?res) and q0: q0 = fst q ..
   from assms(7) have unique-idx (gs @ bs0) (snd ?data) by (simp only: snd-conv)
    from assms(4, 5) dg ‹fst ` set gs ⊆ dgrad-p-set d m› assms(6) - ps0-sub - -
this - - p-in q-in ‹p0 ≠ 0› ‹q0 ≠ 0›
    show crit-pair-cbelow-on d m (fst ` set (fst ?res)) p0 q0 unfolding p0 q0
    proof (rule gb-schema-dummy-connectible)
      from ‹fst ` set bs ⊆ dgrad-p-set d m› show fst ` set bs0 ⊆ dgrad-p-set d m
        by (simp only: set-bs0)
    next
     have D0 ⊆ set bs × (set gs ∪ set bs) by (auto simp: assms(3) minus-pairs-def)
      also have ... ⊆ (set gs ∪ set bs) × (set gs ∪ set bs) by fastforce
       finally show D0 ⊆ (set gs ∪ set bs0) × (set gs ∪ set bs0) by (simp only:
set-bs0)
    next
      show set ps0 ∩p D0 = {}
      proof
        show set ps0 ∩p D0 ⊆ {}
        proof
          fix x
          assume x ∈ set ps0 ∩p D0
          hence x ∈p set ps0 ∩p D0 by (simp add: in-pair-alt)
          thus x ∈ {} by (auto simp: assms(3))
        qed
      qed simp
    next
      fix p' q'
      assume processed (p', q') (gs @ bs0) ps0
      hence proc: processed (p', q') (gs @ bs) ps0
        by (simp add: set-bs0 processed-alt)
      hence p' ∈ set gs ∪ set bs and q' ∈ set gs ∪ set bs and (p', q') ∉p set ps0
        by (auto dest: processedD1 processedD2 processedD3)
      assume (p', q') ∉p D0 and fst p' ≠ 0 and fst q' ≠ 0
      have crit-pair-cbelow-on d m (fst ` (set gs ∪ set bs)) (fst p') (fst q')
      proof (cases p' = q')
        case True
        from dg show ?thesis unfolding True
        proof (rule crit-pair-cbelow-same)
          from ‹q' ∈ set gs ∪ set bs› have fst q' ∈ fst ` (set gs ∪ set bs) by simp
             from this ‹fst ` (set gs ∪ set bs) ⊆ dgrad-p-set d m› show fst q' ∈
dgrad-p-set d m ..
```

```
      qed
    next
      case False
      show ?thesis
       proof (cases component-of-term (lt (fst p')) = component-of-term (lt (fst
q')))
        case True
        show ?thesis
        proof (cases p' ∈ set gs ∧ q' ∈ set gs)
          case True
          note dg ‹fst ` set gs ⊆ dgrad-p-set d m› assms(6)
          moreover from True have fst p' ∈ fst ` set gs and fst q' ∈ fst ` set gs
by simp-all
          ultimately have crit-pair-cbelow-on d m (fst ` set gs) (fst p') (fst q')
        using ‹fst p' ≠ 0› ‹fst q' ≠ 0› by (rule GB-imp-crit-pair-cbelow-dgrad-p-set)
          moreover have fst ` set gs ⊆ fst ` (set gs ∪ set bs) by blast
          ultimately show ?thesis by (rule crit-pair-cbelow-mono)
        next
          case False
          with ‹p' ∈ set gs ∪ set bs› ‹q' ∈ set gs ∪ set bs›
          have (p', q') ∈p set bs × (set gs ∪ set bs) by (auto simp: in-pair-iff)
         with ‹(p', q') ∉p D0› have (p', q') ∈p set ps0 by (simp add: assms(3))
          with ‹(p', q') ∉p set ps0› show ?thesis ..
        qed
      next
        case False
        thus ?thesis by (rule crit-pair-cbelow-distinct-component)
      qed
    qed
    thus crit-pair-cbelow-on d m (fst ` (set gs ∪ set bs0)) (fst p') (fst q')
      by (simp only: set-bs0)
  next
    fix B a b
    assume set gs ∪ set bs0 ⊆ B
    hence B-sup: set gs ∪ set bs ⊆ B by (simp only: set-bs0)
    assume B-sub: fst ` B ⊆ dgrad-p-set d m
    assume (a, b) ∈p D0
    hence ab-in: (a, b) ∈p set bs × (set gs ∪ set bs) and (a, b) ∉p set ps0
      by (simp-all add: assms(3))
    assume fst a ≠ 0 and fst b ≠ 0
    assume *: ⋀x y. x ∈ set gs ∪ set bs0 ⟹ y ∈ set gs ∪ set bs0 ⟹ (x, y) ∉p
D0 ⟹
               fst x ≠ 0 ⟹ fst y ≠ 0 ⟹ crit-pair-cbelow-on d m (fst ` B) (fst
x) (fst y)
    show crit-pair-cbelow-on d m (fst ` B) (fst a) (fst b)
    proof (cases a = b)
      case True
      from ab-in have b ∈ set gs ∪ set bs unfolding True in-pair-iff set-diff-list
by auto
```

hence *fst b* ∈ *fst* ‘ (*set gs* ∪ *set bs*) **by** *fastforce*
from *this gs-bs-sub* **have** *fst b* ∈ *dgrad-p-set d m* **..**
with *dg* **show** *?thesis* **unfolding** *True* **by** (*rule crit-pair-cbelow-same*)
next
case *False*
note *ap dg*
moreover from *B-sup* **have** *B-sup'*: *set gs* ∪ *set* [] ∪ *set bs* ⊆ *B* **by** *simp*
moreover note *B-sub*
moreover from *ab-in* **have** (*a*, *b*) $\in_p$ *set bs* × (*set gs* ∪ *set* [] ∪ *set bs*) **by** *simp*
moreover have *set* [] ⊆ *set* [] × (*set gs* ∪ *set* []) **by** *simp*
moreover from *assms*(*7*) **have** *unique-idx* (*gs* @ [] @ *bs*) *data* **by** (*simp add*: *unique-idx-def set-bs0*)
ultimately show *?thesis* **using** *assms*(*6*) *False* ‹*fst a* ≠ *0*› ‹*fst b* ≠ *0*›
proof (*rule ap-specD2*)
fix *x y* :: (*'t*, *'b*, *'c*) *pdata*
assume (*x*, *y*) $\in_p$ *set* (*ap gs* [] [] *bs data*)
hence (*x*, *y*) $\in_p$ *set ps0* **by** (*simp only*: *ps0-def*)
also have ... ⊆ *set bs0* × (*set gs* ∪ *set bs0*) **by** (*fact ps0-sub*)
also have ... ⊆ (*set gs* ∪ *set bs0*) × (*set gs* ∪ *set bs0*) **by** *fastforce*
finally have (*x*, *y*) ∈ (*set gs* ∪ *set bs0*) × (*set gs* ∪ *set bs0*) **by** (*simp only*: *in-pair-same*)
hence *x* ∈ *set gs* ∪ *set bs0* **and** *y* ∈ *set gs* ∪ *set bs0* **by** *simp-all*
moreover from ‹(*x*, *y*) $\in_p$ *set ps0*› **have** (*x*, *y*) $\notin_p$ *D0* **by** (*simp add*: *D0-def*)
moreover assume *fst x* ≠ *0* **and** *fst y* ≠ *0*
ultimately show *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst x*) (*fst y*) **by** (*rule* ∗)
next
fix *x y* :: (*'t*, *'b*, *'c*) *pdata*
assume *x* ∈ *set gs* ∪ *set* [] **and** *y* ∈ *set gs* ∪ *set* []
hence *fst x* ∈ *fst* ‘ *set gs* **and** *fst y* ∈ *fst* ‘ *set gs* **by** *simp-all*
assume *fst x* ≠ *0* **and** *fst y* ≠ *0*
with *dg* ‹*fst* ‘ *set gs* ⊆ *dgrad-p-set d m*› *assms*(*6*) ‹*fst x* ∈ *fst* ‘ *set gs*› ‹*fst y* ∈ *fst* ‘ *set gs*›
have *crit-pair-cbelow-on d m* (*fst* ‘ *set gs*) (*fst x*) (*fst y*)
by (*rule GB-imp-crit-pair-cbelow-dgrad-p-set*)
moreover from *B-sup* **have** *fst* ‘ *set gs* ⊆ *fst* ‘ *B* **by** *fastforce*
ultimately show *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst x*) (*fst y*)
by (*rule crit-pair-cbelow-mono*)
qed
qed
qed
qed
qed

6.2.10 Function *gb-schema-aux*

function (*domintros*) *gb-schema-aux* :: *nat* × *nat* × *'d* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒

('t, 'b, 'c) pdata-pair list ⇒ ('t, 'b, 'c) pdata list
where
gb-schema-aux data bs ps =
(if ps = [] then
gs @ bs
else
(let sps = sel gs bs ps (snd data); ps0 = ps −− sps; aux = compl gs bs ps0 sps (snd data);
remcomps = fst (data) − count-const-lt-components (fst aux) in
(if remcomps = 0 then
full-gb (gs @ bs)
else
let (hs, data') = add-indices aux (snd data) in
gb-schema-aux (remcomps, data') (ab gs bs hs data') (ap gs bs ps0 hs data')
)
)
)
by *pat-completeness auto*

The *data* parameter of *gb-schema-aux* is a triple (*c*, *i*, *d*), where *c* is the number of components *cmp* of the input list for which the current basis *gs* @ *bs* does *not* yet contain an element whose leading power-product is *0* and has component *cmp*. As soon as *c* gets *0*, the function can return a trivial Gröbner basis, since then the submodule generated by the input list is just the full module. This idea generalizes the well-known fact that if a set of scalar polynomials contains a non-zero constant, the ideal generated by that set is the whole ring. *i* is the total number of polynomials generated during the execution of the function so far; it is used to attach unique indices to the polynomials for fast equality tests. *d*, finally, is some arbitrary data-field that may be used by concrete instances of *gb-schema-aux* for storing information.

lemma *gb-schema-aux-domI1*: *gb-schema-aux-dom* (*data*, *bs*, [])
by (*rule gb-schema-aux.domintros*, *simp*)

lemma *gb-schema-aux-domI2*:
assumes *struct-spec sel ap ab compl*
shows *gb-schema-aux-dom* (*data*, *args*)
proof −
from *assms* **have** *sel*: *sel-spec sel* **and** *ap*: *ap-spec ap* **and** *ab*: *ab-spec ab* **by** (*rule struct-specD*)+
from *ex-dgrad* **obtain** *d*::'*a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
let *?R* = *gb-schema-aux-term d gs*
from *dg* **have** *wf ?R* **by** (*rule gb-schema-aux-term-wf*)
thus *?thesis*
proof (*induct args arbitrary*: *data rule*: *wf-induct-rule*)
fix *x data*
assume *IH*: ⋀*y data'*. (*y*, *x*) ∈ *?R* ⟹ *gb-schema-aux-dom* (*data'*, *y*)

obtain *bs ps* **where** *x*: *x* = (*bs*, *ps*) **by** (*meson case-prodE case-prodI2*)
show *gb-schema-aux-dom* (*data*, *x*) **unfolding** *x*
proof (*rule gb-schema-aux.domintros*)
fix *rc0 n0 data0 hs n1 data1*
assume *ps* ≠ []
and *hs-data′*: (*hs*, *n1*, *data1*) = *add-indices* (*compl gs bs* (*ps* −− *sel gs bs ps* (*n0*, *data0*))
(*sel gs bs ps* (*n0*, *data0*)) (*n0*, *data0*)) (*n0*, *data0*)
and *data*: *data* = (*rc0*, *n0*, *data0*)
define *sps* **where** *sps* = *sel gs bs ps* (*n0*, *data0*)
define *data′* **where** *data′* = (*n1*, *data1*)
define *rc* **where** *rc* = *rc0* − *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sel gs bs ps* (*n0*, *data0*))
(*sel gs bs ps* (*n0*, *data0*)) (*n0*, *data0*)))
from *hs-data′* **have** *hs*: *hs* = *fst* (*add-indices* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)) (*snd data*))
unfolding *sps-def data snd-conv* **by** (*metis fstI*)
show *gb-schema-aux-dom* ((*rc*, *data′*), *ab gs bs hs data′*, *ap gs bs* (*ps* −− *sps*) *hs data′*)
proof (*rule IH*, *simp add*: *x gb-schema-aux-term-def gb-schema-aux-term1-def gb-schema-aux-term2-def*, *intro conjI*)
show *fst* ‘ *set* (*ab gs bs hs data′*) ⊐*p* *fst* ‘ *set bs* ∨
ab gs bs hs data′ = *bs* ∧ *card* (*set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)) < *card* (*set ps*)
proof (*cases hs* = [])
case *True*
have *ab gs bs hs data′* = *bs* ∧ *card* (*set* (*ap gs bs* (*ps* −− *sps*) *hs data′*)) < *card* (*set ps*)
proof (*simp only*: *True*, *rule*)
from *ab* **show** *ab gs bs* [] *data′* = *bs* **by** (*rule ab-specD2*)
next
from *sel* ‹*ps* ≠ []› **have** *sps* ≠ [] **and** *set sps* ⊆ *set ps*
unfolding *sps-def* **by** (*rule sel-specD1*, *rule sel-specD2*)
moreover from *sel-specD1*[*OF sel* ‹*ps* ≠ []›] **have** *set sps* ≠ {} **by** (*simp add*: *sps-def*)
ultimately have *set ps* ∩ *set sps* ≠ {} **by** (*simp add*: *inf.absorb-iff2*)
hence *set* (*ps* −− *sps*) ⊂ *set ps* **unfolding** *set-diff-list* **by** *fastforce*
hence *card* (*set* (*ps* −− *sps*)) < *card* (*set ps*) **by** (*simp add*: *psubset-card-mono*)
moreover have *card* (*set* (*ap gs bs* (*ps* −− *sps*) [] *data′*)) ≤ *card* (*set* (*ps* −− *sps*))
by (*rule card-mono*, *fact finite-set*, *rule ap-spec-Nil-subset*, *fact ap*)
ultimately show *card* (*set* (*ap gs bs* (*ps* −− *sps*) [] *data′*)) < *card* (*set ps*) **by** *simp*
qed
thus *?thesis* **..**
next

case *False*
with *assms* ‹*ps* ≠ []› *sps-def hs* **have** *fst* ' *set* (*ab gs bs hs data'*) ⊐p *fst* ' *set bs*
unfolding *data snd-conv* **by** (*rule struct-spec-red-supset*)
thus *?thesis* **..**
qed
next
from *dg assms* ‹*ps* ≠ []› *sps-def hs*
show *dgrad-p-set-le d* (*args-to-set* (*gs*, *ab gs bs hs data'*, *ap gs bs* (*ps* −− *sps*) *hs data'*)) (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *data snd-conv* **by** (*rule dgrad-p-set-le-args-to-set-struct*)
next
from *assms* ‹*ps* ≠ []› *sps-def hs*
show *component-of-term* ' *Keys* (*args-to-set* (*gs*, *ab gs bs hs data'*, *ap gs bs* (*ps* −− *sps*) *hs data'*)) ⊆
component-of-term ' *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
unfolding *data snd-conv* **by** (*rule components-subset-struct*)
qed
qed
qed
qed

lemma *gb-schema-aux-Nil* [*simp*, *code*]: *gb-schema-aux data bs* [] = *gs* @ *bs*
by (*simp add*: *gb-schema-aux.psimps*[*OF gb-schema-aux-domI1*])

lemmas *gb-schema-aux-simps* = *gb-schema-aux.psimps*[*OF gb-schema-aux-domI2*]

lemma *gb-schema-aux-induct* [*consumes 1*, *case-names base rec1 rec2*]:
assumes *struct-spec sel ap ab compl*
assumes *base*: ⋀*bs data*. *P data bs* [] (*gs* @ *bs*)
and *rec1*: ⋀*bs ps sps data*. *ps* ≠ [] ⟹ *sps* = *sel gs bs ps* (*snd data*) ⟹
fst (*data*) ≤ *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*))) ⟹
P data bs ps (*full-gb* (*gs* @ *bs*))
and *rec2*: ⋀*bs ps sps aux hs rc data data'*. *ps* ≠ [] ⟹ *sps* = *sel gs bs ps* (*snd data*) ⟹
aux = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*) ⟹ (*hs*, *data'*) = *add-indices aux* (*snd data*) ⟹
rc = *fst data* − *count-const-lt-components* (*fst aux*) ⟹ *0* < *rc* ⟹
P (*rc*, *data'*) (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*)
(*gb-schema-aux* (*rc*, *data'*) (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*)) ⟹
P data bs ps (*gb-schema-aux* (*rc*, *data'*) (*ab gs bs hs data'*) (*ap gs bs* (*ps* −− *sps*) *hs data'*))
shows *P data bs ps* (*gb-schema-aux data bs ps*)
proof −
from *assms*(*1*) **have** *gb-schema-aux-dom* (*data*, *bs*, *ps*) **by** (*rule gb-schema-aux-domI2*)
thus *?thesis*
proof (*induct data bs ps rule*: *gb-schema-aux.pinduct*)

case (*1 data bs ps*)
show *?case*
proof (*cases ps* = [])
case *True*
show *?thesis* **by** (*simp add*: *True*, *rule base*)
next
case *False*
show *?thesis*
proof (*simp add*: *gb-schema-aux-simps*[*OF assms*(*1*), *of data bs ps*] *False Let-def split*: *if-split*,
intro conjI impI)
define *sps* **where** *sps* = *sel gs bs ps* (*snd data*)
assume *fst data* ≤ *count-const-lt-components* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)))
with *False sps-def* **show** *P data bs ps* (*full-gb* (*gs* @ *bs*)) **by** (*rule rec1*)
next
define *sps* **where** *sps* = *sel gs bs ps* (*snd data*)
define *aux* **where** *aux* = *compl gs bs* (*ps* −− *sps*) *sps* (*snd data*)
define *hs* **where** *hs* = *fst* (*add-indices aux* (*snd data*))
define *data′* **where** *data′* = *snd* (*add-indices aux* (*snd data*))
define *rc* **where** *rc* = *fst data* − *count-const-lt-components* (*fst aux*)
have *eq*: *add-indices aux* (*snd data*) = (*hs*, *data′*) **by** (*simp add*: *hs-def data′-def*)
assume ¬ *fst data* ≤ *count-const-lt-components* (*fst aux*)
hence *0* < *rc* **by** (*simp add*: *rc-def*)
hence *rc* ≠ *0* **by** *simp*
show *P data bs ps*
(*case add-indices aux* (*snd data*) *of*
(*hs*, *data′*) ⇒ *gb-schema-aux* (*rc*, *data′*) (*ab gs bs hs data′*) (*ap gs bs* (*ps* −− *sps*) *hs data′*))
unfolding *eq prod.case* **using** *False sps-def aux-def eq*[*symmetric*] *rc-def* ‹*0* < *rc*›
proof (*rule rec2*)
show *P* (*rc*, *data′*) (*ab gs bs hs data′*) (*ap gs bs* (*ps* −− *sps*) *hs data′*)
(*gb-schema-aux* (*rc*, *data′*) (*ab gs bs hs data′*) (*ap gs bs* (*ps* −− *sps*) *hs data′*))
using *False sps-def refl aux-def rc-def* ‹*rc* ≠ *0*› *eq*[*symmetric*] *refl* **by** (*rule 1*)
qed
qed
qed
qed
qed

lemma *gb-schema-dummy-eq-gb-schema-aux*:
assumes *struct-spec sel ap ab compl*
shows *fst* (*gb-schema-dummy data D bs ps*) = *gb-schema-aux data bs ps*
using *assms*
proof (*induct data D bs ps rule*: *gb-schema-dummy-induct*)

case (*base bs data D*)
show *?case* **by** *simp*
next
case (*rec1 bs ps sps data D*)
thus *?case* **by** (*simp add*: *gb-schema-aux.psimps*[*OF gb-schema-aux-domI2*, *OF assms*])
next
case (*rec2 bs ps sps aux hs rc data data′ D D′*)
note *rec2.hyps*(*8*)
also from *rec2.hyps*(*1*, *2*, *3*) *rec2.hyps*(*4*)[*symmetric*] *rec2.hyps*(*5*, *6*, *7*)
have *gb-schema-aux* (*rc*, *data′*) (*ab gs bs hs data′*) (*ap gs bs* (*ps* −− *sps*) *hs data′*) =
gb-schema-aux data bs ps
by (*simp add*: *gb-schema-aux.psimps*[*OF gb-schema-aux-domI2*, *OF assms*, *of data*] *Let-def*)
finally show *?case* **.**
qed

corollary *gb-schema-aux-dgrad-p-set-le*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
shows *dgrad-p-set-le d* (*fst ‘ set* (*gb-schema-aux data bs ps*)) (*args-to-set* (*gs*, *bs*, *ps*))
using *fst-gb-schema-dummy-dgrad-p-set-le*[*OF assms*] **unfolding** *gb-schema-dummy-eq-gb-schema-aux*[*OF assms*(*2*)] **.**

corollary *gb-schema-aux-components*:
assumes *struct-spec sel ap ab compl* **and** *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
shows *component-of-term ‘ Keys* (*fst ‘ set* (*gb-schema-aux data bs ps*)) =
component-of-term ‘ Keys (*args-to-set* (*gs*, *bs*, *ps*))
using *fst-gb-schema-dummy-components*[*OF assms*] **unfolding** *gb-schema-dummy-eq-gb-schema-aux*[*OF assms*(*1*)] **.**

lemma *gb-schema-aux-pmdl*:
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis* (*fst ‘ set gs*)
and *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *unique-idx* (*gs* @ *bs*) (*snd data*)
and *rem-comps-spec* (*gs* @ *bs*) *data*
shows *pmdl* (*fst ‘ set* (*gb-schema-aux data bs ps*)) = *pmdl* (*fst ‘ set* (*gs* @ *bs*))
using *fst-gb-schema-dummy-pmdl*[*OF assms*] **unfolding** *gb-schema-dummy-eq-gb-schema-aux*[*OF assms*(*1*)] **.**

corollary *gb-schema-aux-dgrad-p-set-le-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
shows *dgrad-p-set-le d* (*fst ‘ set* (*gb-schema-aux data* (*ab gs* [] *bs* (*snd data*)) (*ap gs* [] [] *bs* (*snd data*))))
(*fst ‘* (*set gs* ∪ *set bs*))
using *fst-gb-schema-dummy-dgrad-p-set-le-init*[*OF assms*] **unfolding** *gb-schema-dummy-eq-gb-schema-aux*[*O assms*(*2*)] **.**

corollary *gb-schema-aux-dgrad-p-set-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
and *fst ‘ (set gs ∪ set bs) ⊆ dgrad-p-set d m*
shows *fst ‘ set (gb-schema-aux (rc, data) (ab gs [] bs data) (ap gs [] [] bs data))*
⊆ dgrad-p-set d m
using *fst-gb-schema-dummy-dgrad-p-set-init[OF assms]* **unfolding** *gb-schema-dummy-eq-gb-schema-aux[OF assms(2)]* **.**

corollary *gb-schema-aux-components-init*:
assumes *struct-spec sel ap ab compl*
shows *component-of-term ‘ Keys (fst ‘ set (gb-schema-aux (rc, data) (ab gs [] bs data) (ap gs [] [] bs data))) =*
component-of-term ‘ Keys (fst ‘ set (gs @ bs))
using *fst-gb-schema-dummy-components-init[OF assms]* **unfolding** *gb-schema-dummy-eq-gb-schema-aux[OF assms]* **.**

corollary *gb-schema-aux-pmdl-init*:
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis (fst ‘ set gs)*
and *unique-idx (gs @ ab gs [] bs data) data* **and** *rem-comps-spec (gs @ ab gs [] bs data) (rc, data)*
shows *pmdl (fst ‘ set (gb-schema-aux (rc, data) (ab gs [] bs data) (ap gs [] [] bs data))) =*
pmdl (fst ‘ (set (gs @ bs)))
using *fst-gb-schema-dummy-pmdl-init[OF assms]* **unfolding** *gb-schema-dummy-eq-gb-schema-aux[OF assms(1)]* **.**

lemma *gb-schema-aux-isGB-init*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *is-Groebner-basis (fst ‘ set gs)*
and *unique-idx (gs @ ab gs [] bs data) data* **and** *rem-comps-spec (gs @ ab gs [] bs data) (rc, data)*
shows *is-Groebner-basis (fst ‘ set (gb-schema-aux (rc, data) (ab gs [] bs data) (ap gs [] [] bs data)))*
using *fst-gb-schema-dummy-isGB-init[OF assms]* **unfolding** *gb-schema-dummy-eq-gb-schema-aux[OF assms(1)]* **.**

end

6.2.11 Functions *gb-schema-direct* and *term gb-schema-incr*

definition *gb-schema-direct* :: *('t, 'b, 'c, 'd) selT ⇒ ('t, 'b, 'c, 'd) apT ⇒ ('t, 'b, 'c, 'd) abT ⇒*
('t, 'b, 'c, 'd) complT ⇒ ('t, 'b, 'c) pdata' list ⇒ 'd ⇒
('t, 'b::field, 'c::default) pdata' list
where *gb-schema-direct sel ap ab compl bs0 data0 =*
(let data = (length bs0, data0); bs1 = fst (add-indices (bs0, data0) (0, data0));
bs = ab [] [] bs1 data in

map (λ(*f*, -, *d*). (*f*, *d*))
(*gb-schema-aux sel ap ab compl* [] (*count-rem-components bs*, *data*)
bs (*ap* [] [] [] *bs1 data*))
)

primrec *gb-schema-incr* :: ('*t*, '*b*, '*c*, '*d*) *selT* ⇒ ('*t*, '*b*, '*c*, '*d*) *apT* ⇒ ('*t*, '*b*, '*c*, '*d*) *abT* ⇒
('*t*, '*b*, '*c*, '*d*) *complT* ⇒
(('*t*, '*b*, '*c*) *pdata list* ⇒ ('*t*, '*b*, '*c*) *pdata* ⇒ '*d* ⇒ '*d*) ⇒
('*t*, '*b*, '*c*) *pdata' list* ⇒ '*d* ⇒ ('*t*, '*b*::*field*, '*c*::*default*) *pdata' list*
where
gb-schema-incr - - - - - [] - = []|
gb-schema-incr sel ap ab compl upd (*b0* # *bs*) *data* =
(*let* (*gs*, *n*, *data'*) = *add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*);
b = (*fst b0*, *n*, *snd b0*); *data''* = *upd gs b data'* *in*
map (λ(*f*, -, *d*). (*f*, *d*))
(*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b* # *gs*), *Suc n*, *data''*)
(*ab gs* [] [*b*] (*Suc n*, *data''*)) (*ap gs* [] [] [*b*] (*Suc n*, *data''*)))
)

lemma (**in** −) *fst-set-drop-indices*:
fst ‘ (λ(*f*, -, *d*). (*f*, *d*)) ‘ *A* = *fst* ‘ *A* **for** *A*::('*x* × '*y* × '*z*) *set*
by (*simp add*: *image-image*, *rule image-cong*, *fact refl*, *simp add*: *prod.case-eq-if*)

lemma *fst-gb-schema-direct*:
fst ‘ *set* (*gb-schema-direct sel ap ab compl bs0 data0*) =
(*let data* = (*length bs0*, *data0*); *bs1* = *fst* (*add-indices* (*bs0*, *data0*) (*0*, *data0*)); *bs* = *ab* [] [] *bs1 data in*
fst ‘ *set* (*gb-schema-aux sel ap ab compl* [] (*count-rem-components bs*, *data*)
bs (*ap* [] [] [] *bs1 data*))
)
by (*simp add*: *gb-schema-direct-def Let-def fst-set-drop-indices*)

lemma *gb-schema-direct-dgrad-p-set*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl* **and** *fst* ‘ *set bs* ⊆ *dgrad-p-set d m*
shows *fst* ‘ *set* (*gb-schema-direct sel ap ab compl bs data*) ⊆ *dgrad-p-set d m*
unfolding *fst-gb-schema-direct Let-def* **using** *assms*(*1*, *2*)
proof (*rule gb-schema-aux-dgrad-p-set-init*)
show *fst* ‘ (*set* [] ∪ *set* (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*)))) ⊆ *dgrad-p-set d m*
using *assms*(*3*) **by** (*simp add*: *image-Un fst-set-add-indices*)
qed

theorem *gb-schema-direct-isGB*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl*

shows *is-Groebner-basis* (*fst* ‘ *set* (*gb-schema-direct sel ap ab compl bs data*))
unfolding *fst-gb-schema-direct Let-def* **using** *assms*
proof (*rule gb-schema-aux-isGB-init*)
from *is-Groebner-basis-empty* **show** *is-Groebner-basis* (*fst* ‘ *set* []) **by** *simp*
next
let *?data* = (*length bs*, *data*)
from *assms*(*1*) **have** *ab-spec ab* **by** (*rule struct-specD*)
moreover have *unique-idx* ([] @ []) (*0*, *data*) **by** (*simp add*: *unique-idx-Nil*)
ultimately show *unique-idx* ([] @ *ab* [] [] (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*))) *?data*) *?data*
proof (*rule unique-idx-ab*)
show (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*)), *length bs*, *data*) = *add-indices* (*bs*, *data*) (*0*, *data*)
by (*simp add*: *add-indices-def*)
qed
qed (*simp add*: *rem-comps-spec-count-rem-components*)

theorem *gb-schema-direct-pmdl*:
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl*
shows *pmdl* (*fst* ‘ *set* (*gb-schema-direct sel ap ab compl bs data*)) = *pmdl* (*fst* ‘ *set bs*)
proof −
have *pmdl* (*fst* ‘ *set* (*gb-schema-direct sel ap ab compl bs data*)) =
pmdl (*fst* ‘ *set* ([] @ (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*)))))
unfolding *fst-gb-schema-direct Let-def* **using** *assms*
proof (*rule gb-schema-aux-pmdl-init*)
from *is-Groebner-basis-empty* **show** *is-Groebner-basis* (*fst* ‘ *set* []) **by** *simp*
next
let *?data* = (*length bs*, *data*)
from *assms*(*1*) **have** *ab-spec ab* **by** (*rule struct-specD*)
moreover have *unique-idx* ([] @ []) (*0*, *data*) **by** (*simp add*: *unique-idx-Nil*)
ultimately show *unique-idx* ([] @ *ab* [] [] (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*))) *?data*) *?data*
proof (*rule unique-idx-ab*)
show (*fst* (*add-indices* (*bs*, *data*) (*0*, *data*)), *length bs*, *data*) = *add-indices* (*bs*, *data*) (*0*, *data*)
by (*simp add*: *add-indices-def*)
qed
qed (*simp add*: *rem-comps-spec-count-rem-components*)
thus *?thesis* **by** (*simp add*: *fst-set-add-indices*)
qed

lemma *fst-gb-schema-incr*:
fst ‘ *set* (*gb-schema-incr sel ap ab compl upd* (*b0* # *bs*) *data*) =
(*let* (*gs*, *n*, *data′*) = *add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*);
b = (*fst b0*, *n*, *snd b0*); *data″* = *upd gs b data′ in*
fst ‘ *set* (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b* # *gs*), *Suc n*, *data″*)

(ab gs [] [b] (Suc n, data'')) (ap gs [] [] [b] (Suc n, data'')))
)
by (*simp only*: *gb-schema-incr.simps Let-def prod.case-distrib[of set] prod.case-distrib[of image fst] set-map fst-set-drop-indices*)

lemma *gb-schema-incr-dgrad-p-set*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
and *fst ‘ set bs ⊆ dgrad-p-set d m*
shows *fst ‘ set (gb-schema-incr sel ap ab compl upd bs data) ⊆ dgrad-p-set d m*
using *assms(3)*
proof (*induct bs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons b0 bs*)
from *Cons(2)* **have** *1*: *fst b0 ∈ dgrad-p-set d m* **and** *2*: *fst ‘ set bs ⊆ dgrad-p-set d m* **by** *simp-all*
show *?case*
proof (*simp only*: *fst-gb-schema-incr Let-def split*: *prod.splits*, *simp*, *intro allI impI*)
fix *gs n data'*
assume *add-indices (gb-schema-incr sel ap ab compl upd bs data, data) (0, data) = (gs, n, data')*
hence *gs*: *gs = fst (add-indices (gb-schema-incr sel ap ab compl upd bs data, data) (0, data))* **by** *simp*
define *b* **where** *b = (fst b0, n, snd b0)*
define *data''* **where** *data'' = upd gs b data'*
from *assms(1, 2)*
show *fst ‘ set (gb-schema-aux sel ap ab compl gs (count-rem-components (b # gs), Suc n, data'') (ab gs [] [b] (Suc n, data'')) (ap gs [] [] [b] (Suc n, data''))) ⊆ dgrad-p-set d m*
proof (*rule gb-schema-aux-dgrad-p-set-init*)
from *1 Cons(1)[OF 2]* **show** *fst ‘ (set gs ∪ set [b]) ⊆ dgrad-p-set d m*
by (*simp add*: *gs fst-set-add-indices b-def*)
qed
qed
qed

theorem *gb-schema-incr-dgrad-p-set-isGB*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl*
shows *is-Groebner-basis (fst ‘ set (gb-schema-incr sel ap ab compl upd bs data))*
proof (*induct bs*)
case *Nil*
from *is-Groebner-basis-empty* **show** *?case* **by** *simp*
next
case (*Cons b0 bs*)
show *?case*
proof (*simp only*: *fst-gb-schema-incr Let-def split*: *prod.splits*, *simp*, *intro allI*

impI)
  fix *gs n data'*
  assume ∗: *add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*) = (*gs*, *n*, *data'*)
  hence *gs*: *gs* = *fst* (*add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*)) **by** *simp*
  define *b* **where** *b* = (*fst b0*, *n*, *snd b0*)
  define *data''* **where** *data''* = *upd gs b data'*
  from *assms*(*1*) **have** *ab*: *ab-spec ab* **by** (*rule struct-specD3*)
  from *Cons* **have** *is-Groebner-basis* (*fst ' set gs*) **by** (*simp add*: *gs fst-set-add-indices*)
  with *assms*
  show *is-Groebner-basis* (*fst ' set* (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b # gs*), *Suc n*, *data''*)
  (*ab gs* [] [*b*] (*Suc n*, *data''*)) (*ap gs* [] [] [*b*] (*Suc n*, *data''*))))
  proof (*rule gb-schema-aux-isGB-init*)
   from *ab* **show** *unique-idx* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data''*)) (*Suc n*, *data''*)
   proof (*rule unique-idx-ab*)
    from *unique-idx-Nil* ∗[*symmetric*] **have** *unique-idx* ([] @ *gs*) (*n*, *data'*)
     by (*rule unique-idx-append*)
    thus *unique-idx* (*gs* @ []) (*n*, *data'*) **by** *simp*
   next
    show ([*b*], *Suc n*, *data''*) = *add-indices* ([*b0*], *data''*) (*n*, *data'*)
     by (*simp add*: *add-indices-def b-def*)
   qed
  next
   have *rem-comps-spec* (*b # gs*) (*count-rem-components* (*b # gs*), *Suc n*, *data''*)
    by (*fact rem-comps-spec-count-rem-components*)
   moreover have *set* (*b # gs*) = *set* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data''*))
    by (*simp add*: *ab-specD1*[*OF ab*])
   ultimately show *rem-comps-spec* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data''*))
    (*count-rem-components* (*b # gs*), *Suc n*, *data''*)
    by (*simp only*: *rem-comps-spec-def*)
  qed
 qed
qed

theorem *gb-schema-incr-pmdl*:
 assumes *struct-spec sel ap ab compl* **and** *compl-conn compl compl-pmdl compl*
 shows *pmdl* (*fst ' set* (*gb-schema-incr sel ap ab compl upd bs data*)) = *pmdl* (*fst ' set bs*)
proof (*induct bs*)
 case *Nil*
 show *?case* **by** *simp*
next
 case (*Cons b0 bs*)
 show *?case*
 proof (*simp only*: *fst-gb-schema-incr Let-def split*: *prod.splits*, *simp*, *intro allI impI*)
  fix *gs n data'*

assume ∗: *add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*) = (*gs*, *n*, *data′*)
hence *gs*: *gs* = *fst* (*add-indices* (*gb-schema-incr sel ap ab compl upd bs data*, *data*) (*0*, *data*)) **by** *simp*
define *b* **where** *b* = (*fst b0*, *n*, *snd b0*)
define *data′′* **where** *data′′* = *upd gs b data′*
from *assms*(*1*) **have** *ab*: *ab-spec ab* **by** (*rule struct-specD3*)
from *assms*(*1*, *2*) **have** *is-Groebner-basis* (*fst ‘ set gs*) **unfolding** *gs fst-conv fst-set-add-indices*
by (*rule gb-schema-incr-dgrad-p-set-isGB*)
with *assms*(*1*, *3*)
have *eq*: *pmdl* (*fst ‘ set* (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b* # *gs*), *Suc n*, *data′′*)
(*ab gs* [] [*b*] (*Suc n*, *data′′*)) (*ap gs* [] [] [*b*] (*Suc n*, *data′′*)))) =
pmdl (*fst ‘ set* (*gs* @ [*b*]))
proof (*rule gb-schema-aux-pmdl-init*)
from *ab* **show** *unique-idx* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data′′*)) (*Suc n*, *data′′*)
proof (*rule unique-idx-ab*)
from *unique-idx-Nil* ∗[*symmetric*] **have** *unique-idx* ([] @ *gs*) (*n*, *data′*)
by (*rule unique-idx-append*)
thus *unique-idx* (*gs* @ []) (*n*, *data′*) **by** *simp*
next
show ([*b*], *Suc n*, *data′′*) = *add-indices* ([*b0*], *data′′*) (*n*, *data′*)
by (*simp add*: *add-indices-def b-def*)
qed
next
have *rem-comps-spec* (*b* # *gs*) (*count-rem-components* (*b* # *gs*), *Suc n*, *data′′*)
by (*fact rem-comps-spec-count-rem-components*)
moreover have *set* (*b* # *gs*) = *set* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data′′*))
by (*simp add*: *ab-specD1*[*OF ab*])
ultimately show *rem-comps-spec* (*gs* @ *ab gs* [] [*b*] (*Suc n*, *data′′*))
(*count-rem-components* (*b* # *gs*), *Suc n*, *data′′*)
by (*simp only*: *rem-comps-spec-def*)
qed
also have ... = *pmdl* (*insert* (*fst b*) (*fst ‘ set gs*)) **by** *simp*
also from *Cons* **have** ... = *pmdl* (*insert* (*fst b*) (*fst ‘ set bs*))
unfolding *gs fst-conv fst-set-add-indices* **by** (*rule pmdl.span-insert-cong*)
finally show *pmdl* (*fst ‘ set* (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b* # *gs*), *Suc n*, *data′′*)
(*ab gs* [] [*b*] (*Suc n*, *data′′*)) (*ap gs* [] [] [*b*] (*Suc n*, *data′′*))))
=
pmdl (*insert* (*fst b0*) (*fst ‘ set bs*)) **by** (*simp add*: *b-def*)
qed
qed

6.3 Suitable Instances of the *add-pairs* Parameter

6.3.1 Specification of the *crit* parameters

type-synonym (**in** −) (*'t*, *'b*, *'c*, *'d*) *icritT* = *nat* × *'d* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata* ⇒ (*'t*, *'b*, *'c*) *pdata* ⇒ *bool*

type-synonym (**in** −) (*'t*, *'b*, *'c*, *'d*) *ncritT* = *nat* × *'d* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*'t*, *'b*, *'c*) *pdata list* ⇒ *bool* ⇒
(*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list* ⇒ (*'t*, *'b*, *'c*) *pdata* ⇒
(*'t*, *'b*, *'c*) *pdata* ⇒ *bool*

type-synonym (**in** −) (*'t*, *'b*, *'c*, *'d*) *ocritT* = *nat* × *'d* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list* ⇒ (*'t*, *'b*, *'c*) *pdata* ⇒
(*'t*, *'b*, *'c*) *pdata* ⇒ *bool*

definition *icrit-spec* :: (*'t*, *'b::field*, *'c*, *'d*) *icritT* ⇒ *bool*
where *icrit-spec crit* ⟷
(∀ *d m data gs bs hs p q*. *dickson-grading d* ⟶
fst ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m* ⟶ *unique-idx* (*gs* @ *bs* @ *hs*) *data* ⟶
is-Groebner-basis (*fst* ‘ *set gs*) ⟶ *p* ∈ *set hs* ⟶ *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟶
fst p ≠ *0* ⟶ *fst q* ≠ *0* ⟶ *crit data gs bs hs p q* ⟶
crit-pair-cbelow-on d m (*fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*))

Criteria satisfying *icrit-spec* can be used for discarding pairs *instantly*, without reference to any other pairs. The product criterion for scalar polynomials satisfies *icrit-spec*, and so does the component criterion (which checks whether the component-indices of the leading terms of two polynomials are identical).

definition *ncrit-spec* :: (*'t*, *'b::field*, *'c*, *'d*) *ncritT* ⇒ *bool*
where *ncrit-spec crit* ⟷
(∀ *d m data gs bs hs ps B q-in-bs p q*. *dickson-grading d* ⟶ *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B* ⟶
fst ‘ *B* ⊆ *dgrad-p-set d m* ⟶ *snd* ‘ *set ps* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⟶
unique-idx (*gs* @ *bs* @ *hs*) *data* ⟶ *is-Groebner-basis* (*fst* ‘ *set gs*) ⟶
(*q-in-bs* ⟶ (*q* ∈ *set gs* ∪ *set bs*)) ⟶
(∀ *p' q'*. (*p'*, *q'*) $\in_p$ *snd* ‘ *set ps* ⟶ *fst p'* ≠ *0* ⟶ *fst q'* ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst p'*) (*fst q'*)) ⟶
(∀ *p' q'*. *p'* ∈ *set gs* ∪ *set bs* ⟶ *q'* ∈ *set gs* ∪ *set bs* ⟶ *fst p'* ≠ *0* ⟶ *fst q'* ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst p'*) (*fst q'*)) ⟶

p ∈ *set hs* ⟶ *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟶ *fst p* ≠ *0* ⟶ *fst q* ≠ *0* ⟶
crit data gs bs hs q-in-bs ps p q ⟶
crit-pair-cbelow-on d m (*fst* ' *B*) (*fst p*) (*fst q*))

definition *ocrit-spec* :: ('*t*, '*b*::*field*, '*c*, '*d*) *ocritT* ⇒ *bool*
where *ocrit-spec crit* ⟷
(∀ *d m data hs ps B p q*. *dickson-grading d* ⟶ *set hs* ⊆ *B* ⟶ *fst* ' *B* ⊆ *dgrad-p-set d m* ⟶
unique-idx (*p* # *q* # *hs* @ (*map* (*fst* ∘ *snd*) *ps*) @ (*map* (*snd* ∘ *snd*) *ps*)) *data* ⟶
(∀ *p*′ *q*′. (*p*′, *q*′) $\in_p$ *snd* ' *set ps* ⟶ *fst p*′ ≠ *0* ⟶ *fst q*′ ≠ *0* ⟶
crit-pair-cbelow-on d m (*fst* ' *B*) (*fst p*′) (*fst q*′)) ⟶
p ∈ *B* ⟶ *q* ∈ *B* ⟶ *fst p* ≠ *0* ⟶ *fst q* ≠ *0* ⟶
crit data hs ps p q ⟶ *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst p*) (*fst q*))

Criteria satisfying *ncrit-spec* can be used for discarding new pairs by reference to new and old elements, whereas criteria satisfying *ocrit-spec* can be used for discarding old pairs by reference to new elements *only* (no existing ones!). The chain criterion satisfies both *ncrit-spec* and *ocrit-spec*.

lemma *icrit-specI*:
assumes ⋀*d m data gs bs hs p q*.
dickson-grading d ⟹ *fst* ' (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m* ⟹
unique-idx (*gs* @ *bs* @ *hs*) *data* ⟹ *is-Groebner-basis* (*fst* ' *set gs*) ⟹
p ∈ *set hs* ⟹ *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* ⟹ *fst p* ≠ *0* ⟹ *fst q* ≠ *0* ⟹
crit data gs bs hs p q ⟹
crit-pair-cbelow-on d m (*fst* ' (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
shows *icrit-spec crit*
unfolding *icrit-spec-def* **using** *assms* **by** *auto*

lemma *icrit-specD*:
assumes *icrit-spec crit* **and** *dickson-grading d*
and *fst* ' (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m* **and** *unique-idx* (*gs* @ *bs* @ *hs*) *data*
and *is-Groebner-basis* (*fst* ' *set gs*) **and** *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs*
and *fst p* ≠ *0* **and** *fst q* ≠ *0* **and** *crit data gs bs hs p q*
shows *crit-pair-cbelow-on d m* (*fst* ' (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
using *assms* **unfolding** *icrit-spec-def* **by** *blast*

lemma *ncrit-specI*:
assumes ⋀*d m data gs bs hs ps B q-in-bs p q*.
dickson-grading d ⟹ *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B* ⟹
fst ' *B* ⊆ *dgrad-p-set d m* ⟹ *snd* ' *set ps* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⟹
unique-idx (*gs* @ *bs* @ *hs*) *data* ⟹ *is-Groebner-basis* (*fst* ' *set gs*) ⟹
(*q-in-bs* ⟶ *q* ∈ *set gs* ∪ *set bs*) ⟹

($\bigwedge p'\ q'.\ (p',\ q') \in_p$ *snd* ‘ *set ps* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')) $\Longrightarrow$
($\bigwedge p'\ q'.\ p' \in$ *set gs* $\cup$ *set bs* $\Longrightarrow q' \in$ *set gs* $\cup$ *set bs* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')) $\Longrightarrow$
$p \in$ *set hs* $\Longrightarrow q \in$ *set gs* $\cup$ *set bs* $\cup$ *set hs* $\Longrightarrow$ *fst* $p \neq 0 \Longrightarrow$ *fst* $q \neq 0$ $\Longrightarrow$
crit data gs bs hs q-in-bs ps p q $\Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst p*) (*fst q*)
shows *ncrit-spec crit*
unfolding *ncrit-spec-def* **by** (*intro allI impI*, *rule assms*, *assumption+*, *meson*, *meson*, *assumption+*)

lemma *ncrit-specD*:
assumes *ncrit-spec crit* **and** *dickson-grading d* **and** *set gs* $\cup$ *set bs* $\cup$ *set hs* $\subseteq B$
and *fst* ‘ $B \subseteq$ *dgrad-p-set d m* **and** *snd* ‘ *set ps* $\subseteq$ *set hs* $\times$ (*set gs* $\cup$ *set bs* $\cup$ *set hs*)
and *unique-idx* (*gs* @ *bs* @ *hs*) *data* **and** *is-Groebner-basis* (*fst* ‘ *set gs*)
and *q-in-bs* $\Longrightarrow q \in$ *set gs* $\cup$ *set bs*
and $\bigwedge p'\ q'.\ (p',\ q') \in_p$ *snd* ‘ *set ps* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')
and $\bigwedge p'\ q'.\ p' \in$ *set gs* $\cup$ *set bs* $\Longrightarrow q' \in$ *set gs* $\cup$ *set bs* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')
and $p \in$ *set hs* **and** $q \in$ *set gs* $\cup$ *set bs* $\cup$ *set hs* **and** *fst* $p \neq 0$ **and** *fst* $q \neq 0$
and *crit data gs bs hs q-in-bs ps p q*
shows *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
using *assms* **unfolding** *ncrit-spec-def* **by** *blast*

lemma *ocrit-specI*:
assumes $\bigwedge d\ m$ *data hs ps B p q*.
dickson-grading d $\Longrightarrow$ *set hs* $\subseteq B \Longrightarrow$ *fst* ‘ $B \subseteq$ *dgrad-p-set d m* $\Longrightarrow$
unique-idx (p # q # *hs* @ (*map* (*fst* $\circ$ *snd*) *ps*) @ (*map* (*snd* $\circ$ *snd*) *ps*)) *data* $\Longrightarrow$
($\bigwedge p'\ q'.\ (p',\ q') \in_p$ *snd* ‘ *set ps* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')) $\Longrightarrow$
$p \in B \Longrightarrow q \in B \Longrightarrow$ *fst* $p \neq 0 \Longrightarrow$ *fst* $q \neq 0 \Longrightarrow$
crit data hs ps p q $\Longrightarrow$ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
shows *ocrit-spec crit*
unfolding *ocrit-spec-def* **by** (*intro allI impI*, *rule assms*, *assumption+*, *meson*, *assumption+*)

lemma *ocrit-specD*:
assumes *ocrit-spec crit* **and** *dickson-grading d* **and** *set hs* $\subseteq B$ **and** *fst* ‘ $B \subseteq$ *dgrad-p-set d m*
and *unique-idx* (p # q # *hs* @ (*map* (*fst* $\circ$ *snd*) *ps*) @ (*map* (*snd* $\circ$ *snd*) *ps*)) *data*
and $\bigwedge p'\ q'.\ (p',\ q') \in_p$ *snd* ‘ *set ps* $\Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')

and *p* ∈ *B* **and** *q* ∈ *B* **and** *fst p* ≠ *0* **and** *fst q* ≠ *0*
and *crit data hs ps p q*
shows *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst p*) (*fst q*)
using *assms* **unfolding** *ocrit-spec-def* **by** *blast*

6.3.2 Suitable instances of the *crit* parameters

definition *component-crit* :: ('*t*, '*b*::*zero*, '*c*, '*d*) *icritT*
where *component-crit data gs bs hs p q* ⟷ (*component-of-term* (*lt* (*fst p*)) ≠ *component-of-term* (*lt* (*fst q*)))

lemma *icrit-spec-component-crit*: *icrit-spec* (*component-crit*::('*t*, '*b*::*field*, '*c*, '*d*) *icritT*)
proof (*rule icrit-specI*)
fix *d m* **and** *data*::*nat* × '*d* **and** *gs bs hs* **and** *p q*::('*t*, '*b*, '*c*) *pdata*
assume *component-crit data gs bs hs p q*
hence *component-of-term* (*lt* (*fst p*)) ≠ *component-of-term* (*lt* (*fst q*))
by (*simp add*: *component-crit-def*)
thus *crit-pair-cbelow-on d m* (*fst* ' (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
by (*rule crit-pair-cbelow-distinct-component*)
qed

The product criterion is only applicable to scalar polynomials.

definition *product-crit* :: ('*a*, '*b*::*zero*, '*c*, '*d*) *icritT*
where *product-crit data gs bs hs p q* ⟷ (*gcs* (*punit.lt* (*fst p*)) (*punit.lt* (*fst q*)) = *0*)

lemma (**in** *gd-term*) *icrit-spec-product-crit*: *punit.icrit-spec* (*product-crit*::('*a*, '*b*::*field*, '*c*, '*d*) *icritT*)
proof (*rule punit.icrit-specI*)
fix *d m* **and** *data*::*nat* × '*d* **and** *gs bs hs* **and** *p q*::('*a*, '*b*, '*c*) *pdata*
assume *product-crit data gs bs hs p q*
hence ∗: *gcs* (*punit.lt* (*fst p*)) (*punit.lt* (*fst q*)) = *0* **by** (*simp only*: *product-crit-def*)
assume *p* ∈ *set hs* **and** *q-in*: *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* (**is** - ∈ *?B*)
assume *dickson-grading d* **and** *sub*: *fst* ' (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *punit.dgrad-p-set d m*
moreover from ‹*p* ∈ *set hs*› **have** *fst p* ∈ *fst* ' *?B* **by** *simp*
moreover from *q-in* **have** *fst q* ∈ *fst* ' *?B* **by** *simp*
moreover assume *fst p* ≠ *0* **and** *fst q* ≠ *0*
ultimately show *punit.crit-pair-cbelow-on d m* (*fst* ' *?B*) (*fst p*) (*fst q*)
using ∗ **by** (*rule product-criterion*)
qed

component-crit and *product-crit* ignore the *data* parameter.

fun (**in** −) *pair-in-list* :: (*bool* × ('*a*, '*b*, '*c*) *pdata-pair*) *list* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
pair-in-list [] - - = *False*
|*pair-in-list* ((-, (-, *i*', -), (-, *j*', -)) # *ps*) *i j* =

$((i = i' \wedge j = j') \vee (i = j' \wedge j = i') \vee$ *pair-in-list ps i j*)

lemma (**in** −) *pair-in-listE*:
assumes *pair-in-list ps i j*
obtains *p q a b* **where** $((p, i, a), (q, j, b)) \in_p$ *snd ' set ps*
using *assms*
proof (*induct ps i j arbitrary*: *thesis rule*: *pair-in-list.induct*)
case (*1 i j*)
from *1*(*2*) **show** *?case* **by** *simp*
next
case (*2 c p i' a q j' b ps i j*)
from *2*(*3*) **have** $(i = i' \wedge j = j') \vee (i = j' \wedge j = i') \vee$ *pair-in-list ps i j* **by** *simp*
thus *?case*
proof (*elim disjE conjE*)
assume $i = i'$ **and** $j = j'$
have $((p, i, a), (q, j, b)) \in_p$ *snd ' set* ((*c*, (*p*, *i'*, *a*), *q*, *j'*, *b*) # *ps*)
unfolding ‹$i = i'$› ‹$j = j'$› *in-pair-iff* **by** *fastforce*
thus *?thesis* **by** (*rule 2*(*2*))
next
assume $i = j'$ **and** $j = i'$
have $((q, i, b), (p, j, a)) \in_p$ *snd ' set* ((*c*, (*p*, *i'*, *a*), *q*, *j'*, *b*) # *ps*)
unfolding ‹$i = j'$› ‹$j = i'$› *in-pair-iff* **by** *fastforce*
thus *?thesis* **by** (*rule 2*(*2*))
next
assume *pair-in-list ps i j*
obtain *p' q' a' b'* **where** $((p', i, a'), (q', j, b')) \in_p$ *snd ' set ps*
by (*rule 2*(*1*), *assumption*, *rule* ‹*pair-in-list ps i j*›)
also have ... $\subseteq$ *snd ' set* ((*c*, (*p*, *i'*, *a*), *q*, *j'*, *b*) # *ps*) **by** *auto*
finally show *?thesis* **by** (*rule 2*(*2*))
qed
qed

definition *chain-ncrit* :: (*'t*, *'b*::*zero*, *'c*, *'d*) *ncritT*
where *chain-ncrit data gs bs hs q-in-bs ps p q* ⟷
(*let v* = *lt* (*fst p*); *l* = *term-of-pair* (*lcs* (*pp-of-term v*) (*lp* (*fst q*)), *component-of-term v*);
i = *fst* (*snd p*); *j* = *fst* (*snd q*) *in*
($\exists$ *r*$\in$*set gs*. *let k* = *fst* (*snd r*) *in*
$k \neq i \wedge k \neq j \wedge$ *lt* (*fst r*) $adds_t$ $l \wedge$ *pair-in-list ps i k* $\wedge$ (*q-in-bs* $\vee$ *pair-in-list ps j k*) $\wedge$ *fst r* $\neq$ *0*) $\vee$
($\exists$ *r*$\in$*set bs*. *let k* = *fst* (*snd r*) *in*
$k \neq i \wedge k \neq j \wedge$ *lt* (*fst r*) $adds_t$ $l \wedge$ *pair-in-list ps i k* $\wedge$ (*q-in-bs* $\vee$ *pair-in-list ps j k*) $\wedge$ *fst r* $\neq$ *0*) $\vee$
($\exists$ *h*$\in$*set hs*. *let k* = *fst* (*snd h*) *in*
$k \neq i \wedge k \neq j \wedge$ *lt* (*fst h*) $adds_t$ $l \wedge$ *pair-in-list ps i k* $\wedge$ *pair-in-list ps j k* $\wedge$ *fst h* $\neq$ *0*))

definition *chain-ocrit* :: (*'t*, *'b*::*zero*, *'c*, *'d*) *ocritT*
where *chain-ocrit data hs ps p q* ⟷

(*let v = lt (fst p); l = term-of-pair (lcs (pp-of-term v) (lp (fst q)), component-of-term v);*
i = fst (snd p); j = fst (snd q) in
(∃ h∈set hs. let k = fst (snd h) in
k ≠ i ∧ k ≠ j ∧ lt (fst h) adds$_t$ l ∧ pair-in-list ps i k ∧ pair-in-list ps j k ∧ fst h ≠ 0))

chain-ncrit and *chain-ocrit* ignore the *data* parameter.

lemma *chain-ncritE*:
assumes *chain-ncrit data gs bs hs q-in-bs ps p q* **and** *snd ‘ set ps ⊆ set hs × (set gs ∪ set bs ∪ set hs)*
and *unique-idx (gs @ bs @ hs) data* **and** *p ∈ set hs* **and** *q ∈ set gs ∪ set bs ∪ set hs*
obtains *r* **where** *r ∈ set gs ∪ set bs ∪ set hs* **and** *fst r ≠ 0* **and** *r ≠ p* **and** *r ≠ q*
and *lt (fst r) adds$_t$ term-of-pair (lcs (lp (fst p)) (lp (fst q)), component-of-term (lt (fst p)))*
and *(p, r) ∈$_p$ snd ‘ set ps* **and** *(r ∈ set gs ∪ set bs ∧ q-in-bs) ∨ (q, r) ∈$_p$ snd ‘ set ps*
proof −
let *?l = term-of-pair (lcs (lp (fst p)) (lp (fst q)), component-of-term (lt (fst p)))*
let *?i = fst (snd p)*
let *?j = fst (snd q)*
let *?xs = gs @ bs @ hs*
have *3*: *x ∈ set ?xs* **if** *(x, y) ∈$_p$ snd ‘ set ps* **for** *x y*
proof −
note *that*
also have *snd ‘ set ps ⊆ set hs × (set gs ∪ set bs ∪ set hs)* **by** *(fact assms(2))*
also have *... ⊆ (set gs ∪ set bs ∪ set hs) × (set gs ∪ set bs ∪ set hs)* **by** *fastforce*
finally have *(x, y) ∈ (set gs ∪ set bs ∪ set hs) × (set gs ∪ set bs ∪ set hs)*
by *(simp only: in-pair-same)*
thus *?thesis* **by** *simp*
qed
have *4*: *x ∈ set ?xs* **if** *(y, x) ∈$_p$ snd ‘ set ps* **for** *x y*
proof −
from *that* **have** *(x, y) ∈$_p$ snd ‘ set ps* **by** *(simp add: in-pair-iff disj-commute)*
thus *?thesis* **by** *(rule 3)*
qed

from *assms(1)* **have**
∃ r ∈ set gs ∪ set bs ∪ set hs. let k = fst (snd r) in
k ≠ ?i ∧ k ≠ ?j ∧ lt (fst r) adds$_t$?l ∧ pair-in-list ps ?i k ∧
((r ∈ set gs ∪ set bs ∧ q-in-bs) ∨ pair-in-list ps ?j k) ∧ fst r ≠ 0
by *(smt (verit) Un-iff chain-ncrit-def)*

then obtain *r* **where** *r-in*: *r ∈ set gs ∪ set bs ∪ set hs* **and** *fst r ≠ 0* **and** *rp*: *fst (snd r) ≠ ?i*
and *rq*: *fst (snd r) ≠ ?j* **and** *lt (fst r) adds$_t$?l*

and *1*: *pair-in-list ps ?i* (*fst* (*snd r*))
and *2*: (*r* ∈ *set gs* ∪ *set bs* ∧ *q-in-bs*) ∨ *pair-in-list ps ?j* (*fst* (*snd r*))
unfolding *Let-def* **by** *blast*
let *?k* = *fst* (*snd r*)
note *r-in* ‹*fst r* ≠ *0*›
moreover from *rp* **have** $r \neq p$ **by** *auto*
moreover from *rq* **have** $r \neq q$ **by** *auto*
ultimately show *?thesis* **using** ‹*lt* (*fst r*) $adds_t$ *?l*›
proof
from *1* **obtain** p' r' *a b* **where** ∗: ((p', *?i*, *a*), (r', *?k*, *b*)) $\in_p$ *snd* ‘ *set ps*
by (*rule pair-in-listE*)

note *assms*(*3*)
moreover from ∗ **have** (p', *?i*, *a*) ∈ *set ?xs* **by** (*rule 3*)
moreover from *assms*(*4*) **have** *p* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (p', *?i*, *a*)) = *?i* **by** *simp*
ultimately have p': (p', *?i*, *a*) = *p* **by** (*rule unique-idxD1*)

note *assms*(*3*)
moreover from ∗ **have** (r', *?k*, *b*) ∈ *set ?xs* **by** (*rule 4*)
moreover from *r-in* **have** *r* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (r', *?k*, *b*)) = *?k* **by** *simp*
ultimately have r': (r', *?k*, *b*) = *r* **by** (*rule unique-idxD1*)

from ∗ **show** (*p*, *r*) $\in_p$ *snd* ‘ *set ps* **by** (*simp only*: p' r')
next
from *2* **show** (*r* ∈ *set gs* ∪ *set bs* ∧ *q-in-bs*) ∨ (*q*, *r*) $\in_p$ *snd* ‘ *set ps*
proof
assume *r* ∈ *set gs* ∪ *set bs* ∧ *q-in-bs*
thus *?thesis* **..**
next
assume *pair-in-list ps ?j ?k*
then obtain q' r' *a b* **where** ∗: ((q', *?j*, *a*), (r', *?k*, *b*)) $\in_p$ *snd* ‘ *set ps*
by (*rule pair-in-listE*)

note *assms*(*3*)
moreover from ∗ **have** (q', *?j*, *a*) ∈ *set ?xs* **by** (*rule 3*)
moreover from *assms*(*5*) **have** *q* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (q', *?j*, *a*)) = *?j* **by** *simp*
ultimately have q': (q', *?j*, *a*) = *q* **by** (*rule unique-idxD1*)

note *assms*(*3*)
moreover from ∗ **have** (r', *?k*, *b*) ∈ *set ?xs* **by** (*rule 4*)
moreover from *r-in* **have** *r* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (r', *?k*, *b*)) = *?k* **by** *simp*
ultimately have r': (r', *?k*, *b*) = *r* **by** (*rule unique-idxD1*)

from ∗ **have** (*q*, *r*) $\in_p$ *snd* ‘ *set ps* **by** (*simp only*: q' r')
thus *?thesis* **..**

qed
qed
qed

lemma *chain-ocritE*:
assumes *chain-ocrit data hs ps p q*
and *unique-idx* (*p* # *q* # *hs* @ (*map* (*fst* ∘ *snd*) *ps*) @ (*map* (*snd* ∘ *snd*) *ps*)) *data* (**is** *unique-idx ?xs* -)
obtains *h* **where** $h \in$ *set hs* **and** *fst h* $\neq$ *0* **and** $h \neq p$ **and** $h \neq q$
and *lt* (*fst h*) $adds_t$ *term-of-pair* (*lcs* (*lp* (*fst p*)) (*lp* (*fst q*)), *component-of-term* (*lt* (*fst p*)))
and (*p*, *h*) $\in_p$ *snd* ' *set ps* **and** (*q*, *h*) $\in_p$ *snd* ' *set ps*
proof −
let *?l* = *term-of-pair* (*lcs* (*lp* (*fst p*)) (*lp* (*fst q*)), *component-of-term* (*lt* (*fst p*)))
have *3*: $x \in$ *set ?xs* **if** (*x*, *y*) $\in_p$ *snd* ' *set ps* **for** *x y*
proof −
from *that* **have** (*x*, *y*) ∈ *snd* ' *set ps* ∨ (*y*, *x*) ∈ *snd* ' *set ps* **by** (*simp only*: *in-pair-iff*)
thus *?thesis*
proof
assume (*x*, *y*) ∈ *snd* ' *set ps*
hence *fst* (*x*, *y*) ∈ *fst* ' *snd* ' *set ps* **by** *fastforce*
thus *?thesis* **by** (*simp add*: *image-comp*)
next
assume (*y*, *x*) ∈ *snd* ' *set ps*
hence *snd* (*y*, *x*) ∈ *snd* ' *snd* ' *set ps* **by** *fastforce*
thus *?thesis* **by** (*simp add*: *image-comp*)
qed
qed
have *4*: $x \in$ *set ?xs* **if** (*y*, *x*) $\in_p$ *snd* ' *set ps* **for** *x y*
proof −
from *that* **have** (*x*, *y*) $\in_p$ *snd* ' *set ps* **by** (*simp add*: *in-pair-iff disj-commute*)
thus *?thesis* **by** (*rule 3*)
qed

from *assms*(*1*) **obtain** *h* **where** $h \in$ *set hs* **and** *fst h* $\neq$ *0* **and** *hp*: *fst* (*snd h*) $\neq$ *fst* (*snd p*)
and *hq*: *fst* (*snd h*) $\neq$ *fst* (*snd q*) **and** *lt* (*fst h*) $adds_t$ *?l*
and *1*: *pair-in-list ps* (*fst* (*snd p*)) (*fst* (*snd h*)) **and** *2*: *pair-in-list ps* (*fst* (*snd q*)) (*fst* (*snd h*))
unfolding *chain-ocrit-def Let-def* **by** *blast*
let *?i* = *fst* (*snd p*)
let *?j* = *fst* (*snd q*)
let *?k* = *fst* (*snd h*)
note ‹$h \in$ *set hs*› ‹*fst h* $\neq$ *0*›
moreover from *hp* **have** $h \neq p$ **by** *auto*
moreover from *hq* **have** $h \neq q$ **by** *auto*
ultimately show *?thesis* **using** ‹*lt* (*fst h*) $adds_t$ *?l*›
proof

from *1* **obtain** *p′ h′ a b* **where** ∗: ((*p′*, *?i*, *a*), (*h′*, *?k*, *b*)) $\in_p$ *snd ‘ set ps*
by (*rule pair-in-listE*)

note *assms*(*2*)
moreover from ∗ **have** (*p′*, *?i*, *a*) ∈ *set ?xs* **by** (*rule 3*)
moreover have *p* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (*p′*, *?i*, *a*)) = *?i* **by** *simp*
ultimately have *p′*: (*p′*, *?i*, *a*) = *p* **by** (*rule unique-idxD1*)

note *assms*(*2*)
moreover from ∗ **have** (*h′*, *?k*, *b*) ∈ *set ?xs* **by** (*rule 4*)
moreover from ‹*h* ∈ *set hs*› **have** *h* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (*h′*, *?k*, *b*)) = *?k* **by** *simp*
ultimately have *h′*: (*h′*, *?k*, *b*) = *h* **by** (*rule unique-idxD1*)

from ∗ **show** (*p*, *h*) $\in_p$ *snd ‘ set ps* **by** (*simp only*: *p′ h′*)
next
from *2* **obtain** *q′ h′ a b* **where** ∗: ((*q′*, *?j*, *a*), (*h′*, *?k*, *b*)) $\in_p$ *snd ‘ set ps*
by (*rule pair-in-listE*)

note *assms*(*2*)
moreover from ∗ **have** (*q′*, *?j*, *a*) ∈ *set ?xs* **by** (*rule 3*)
moreover have *q* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (*q′*, *?j*, *a*)) = *?j* **by** *simp*
ultimately have *q′*: (*q′*, *?j*, *a*) = *q* **by** (*rule unique-idxD1*)

note *assms*(*2*)
moreover from ∗ **have** (*h′*, *?k*, *b*) ∈ *set ?xs* **by** (*rule 4*)
moreover from ‹*h* ∈ *set hs*› **have** *h* ∈ *set ?xs* **by** *simp*
moreover have *fst* (*snd* (*h′*, *?k*, *b*)) = *?k* **by** *simp*
ultimately have *h′*: (*h′*, *?k*, *b*) = *h* **by** (*rule unique-idxD1*)

from ∗ **show** (*q*, *h*) $\in_p$ *snd ‘ set ps* **by** (*simp only*: *q′ h′*)
qed
qed

lemma *ncrit-spec-chain-ncrit*: *ncrit-spec* (*chain-ncrit*::(′*t*, ′*b*::*field*, ′*c*, ′*d*) *ncritT*)
proof (*rule ncrit-specI*)
fix *d m* **and** *data*::*nat* × ′*d* **and** *gs bs hs* **and** *ps*::(*bool* × (′*t*, ′*b*, ′*c*) *pdata-pair*) *list*
and *B q-in-bs* **and** *p q*::(′*t*, ′*b*, ′*c*) *pdata*
assume *dg*: *dickson-grading d* **and** *B-sup*: *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B*
and *B-sub*: *fst ‘ B* ⊆ *dgrad-p-set d m* **and** *q-in-bs*: *q-in-bs* ⟶ *q* ∈ *set gs* ∪ *set bs*
and *1*: ⋀*p′ q′*. (*p′*, *q′*) $\in_p$ *snd ‘ set ps* ⟹ *fst p′* ≠ *0* ⟹ *fst q′* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst ‘ B*) (*fst p′*) (*fst q′*)
and *2*: ⋀*p′ q′*. *p′* ∈ *set gs* ∪ *set bs* ⟹ *q′* ∈ *set gs* ∪ *set bs* ⟹ *fst p′* ≠ *0* ⟹ *fst q′* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst ‘ B*) (*fst p′*) (*fst q′*)

and *fst p* ≠ *0* **and** *fst q* ≠ *0*
let *?l* = *term-of-pair* (*lcs* (*lp* (*fst p*)) (*lp* (*fst q*)), *component-of-term* (*lt* (*fst p*)))
assume *chain-ncrit data gs bs hs q-in-bs ps p q* **and** *snd ‘ set ps* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **and**
unique-idx (*gs* @ *bs* @ *hs*) *data* **and** *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs*
then obtain *r* **where** *r* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *fst r* ≠ *0* **and** *r* ≠ *p* **and** *r* ≠ *q*
and *adds*: *lt* (*fst r*) $adds_t$ *?l* **and** (*p*, *r*) $\in_p$ *snd ‘ set ps*
and *disj*: (*r* ∈ *set gs* ∪ *set bs* ∧ *q-in-bs*) ∨ (*q*, *r*) $\in_p$ *snd ‘ set ps* **by** (*rule chain-ncritE*)
note *dg B-sub*
moreover from ‹*p* ∈ *set hs*› ‹*q* ∈ *set gs* ∪ *set bs* ∪ *set hs*› *B-sup*
have *fst p* ∈ *fst ‘ B* **and** *fst q* ∈ *fst ‘ B*
by *auto*
moreover note ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*›
moreover from *adds* **have** *lp* (*fst r*) *adds lcs* (*lp* (*fst p*)) (*lp* (*fst q*))
by (*simp add*: *adds-term-def term-simps*)
moreover from *adds* **have** *component-of-term* (*lt* (*fst r*)) = *component-of-term* (*lt* (*fst p*))
by (*simp add*: *adds-term-def term-simps*)
ultimately show *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst p*) (*fst q*)
proof (*rule chain-criterion*)
from ‹(*p*, *r*) $\in_p$ *snd ‘ set ps*› ‹*fst p* ≠ *0*› ‹*fst r* ≠ *0*›
show *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst p*) (*fst r*) **by** (*rule 1*)
next
from *disj* **show** *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst r*) (*fst q*)
proof
assume *r* ∈ *set gs* ∪ *set bs* ∧ *q-in-bs*
hence *r* ∈ *set gs* ∪ *set bs* **and** *q-in-bs* **by** *simp-all*
from *q-in-bs this*(*2*) **have** *q* ∈ *set gs* ∪ *set bs* **..**
with ‹*r* ∈ *set gs* ∪ *set bs*› **show** *?thesis* **using** ‹*fst r* ≠ *0*› ‹*fst q* ≠ *0*› **by** (*rule 2*)
next
assume (*q*, *r*) $\in_p$ *snd ‘ set ps*
hence (*r*, *q*) $\in_p$ *snd ‘ set ps* **by** (*simp only*: *in-pair-iff disj-commute*)
thus *?thesis* **using** ‹*fst r* ≠ *0*› ‹*fst q* ≠ *0*› **by** (*rule 1*)
qed
qed
qed

lemma *ocrit-spec-chain-ocrit*: *ocrit-spec* (*chain-ocrit*::(*'t*, *'b*::*field*, *'c*, *'d*) *ocritT*)
proof (*rule ocrit-specI*)
fix *d m* **and** *data*::*nat* × *'d* **and** *hs*::(*'t*, *'b*, *'c*) *pdata list* **and** *ps*::(*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list*
and *B* **and** *p q*::(*'t*, *'b*, *'c*) *pdata*
assume *dg*: *dickson-grading d* **and** *B-sup*: *set hs* ⊆ *B*
and *B-sub*: *fst ‘ B* ⊆ *dgrad-p-set d m*
and *1*: ⋀*p' q'*. (*p'*, *q'*) $\in_p$ *snd ‘ set ps* ⟹ *fst p'* ≠ *0* ⟹ *fst q'* ≠ *0* ⟹
crit-pair-cbelow-on d m (*fst ‘ B*) (*fst p'*) (*fst q'*)

and *fst p* ≠ *0* **and** *fst q* ≠ *0* **and** *p* ∈ *B* **and** *q* ∈ *B*
let *?l* = *term-of-pair* (*lcs* (*lp* (*fst p*)) (*lp* (*fst q*)), *component-of-term* (*lt* (*fst p*)))
assume *chain-ocrit data hs ps p q* **and** *unique-idx* (*p* # *q* # *hs* @ *map* (*fst* ∘ *snd*) *ps* @ *map* (*snd* ∘ *snd*) *ps*) *data*
then obtain *h* **where** *h* ∈ *set hs* **and** *fst h* ≠ *0* **and** *h* ≠ *p* **and** *h* ≠ *q*
and *adds*: *lt* (*fst h*) $adds_t$ *?l* **and** (*p*, *h*) $\in_p$ *snd* ' *set ps* **and** (*q*, *h*) $\in_p$ *snd* ' *set ps*
by (*rule chain-ocritE*)
note *dg B-sub*
moreover from ‹*p* ∈ *B*› ‹*q* ∈ *B*› *B-sup*
have *fst p* ∈ *fst* ' *B* **and** *fst q* ∈ *fst* ' *B* **by** *auto*
moreover note ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*›
moreover from *adds* **have** *lp* (*fst h*) *adds lcs* (*lp* (*fst p*)) (*lp* (*fst q*))
by (*simp add*: *adds-term-def term-simps*)
moreover from *adds* **have** *component-of-term* (*lt* (*fst h*)) = *component-of-term* (*lt* (*fst p*))
by (*simp add*: *adds-term-def term-simps*)
ultimately show *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst p*) (*fst q*)
proof (*rule chain-criterion*)
from ‹(*p*, *h*) $\in_p$ *snd* ' *set ps*› ‹*fst p* ≠ *0*› ‹*fst h* ≠ *0*›
show *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst p*) (*fst h*) **by** (*rule 1*)
next
from ‹(*q*, *h*) $\in_p$ *snd* ' *set ps*› **have** (*h*, *q*) $\in_p$ *snd* ' *set ps* **by** (*simp only*: *in-pair-iff disj-commute*)
thus *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst h*) (*fst q*) **using** ‹*fst h* ≠ *0*› ‹*fst q* ≠ *0*› **by** (*rule 1*)
qed
qed

lemma *icrit-spec-no-crit*: *icrit-spec* ((λ- - - - - -. *False*)::(*'t*, *'b*::*field*, *'c*, *'d*) *icritT*)
by (*rule icrit-specI*, *simp*)

lemma *ncrit-spec-no-crit*: *ncrit-spec* ((λ- - - - - - - -. *False*)::(*'t*, *'b*::*field*, *'c*, *'d*) *ncritT*)
by (*rule ncrit-specI*, *simp*)

lemma *ocrit-spec-no-crit*: *ocrit-spec* ((λ- - - - -. *False*)::(*'t*, *'b*::*field*, *'c*, *'d*) *ocritT*)
by (*rule ocrit-specI*, *simp*)

6.3.3 Creating Initial List of New Pairs

type-synonym (**in** −) (*'t*, *'b*, *'c*) *apsT* = *bool* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*'t*, *'b*, *'c*) *pdata* ⇒ (*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list* ⇒
(*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list*

type-synonym (**in** −) (*'t*, *'b*, *'c*, *'d*) *npT* = (*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒

$('t, 'b, 'c)$ *pdata list* ⇒ *nat* × *'d* ⇒
(*bool* × $('t, 'b, 'c)$ *pdata-pair*) *list*

definition *np-spec* :: $('t, 'b, 'c, 'd)$ *npT* ⇒ *bool*
where *np-spec np* ⟷ (∀ *gs bs hs data*.
snd ‘ *set* (*np gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∧
set hs × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*np gs bs hs data*) ∧
(∀ *a b*. *a* ∈ *set hs* ⟶ *b* ∈ *set hs* ⟶ *a* ≠ *b* ⟶ (*a*, *b*) $\in_p$ *snd* ‘ *set* (*np gs bs hs data*)) ∧
(∀ *p q*. (*True*, *p*, *q*) ∈ *set* (*np gs bs hs data*) ⟶ *q* ∈ *set gs* ∪ *set bs*))

lemma *np-specI*:
assumes ⋀*gs bs hs data*.
snd ‘ *set* (*np gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∧
set hs × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*np gs bs hs data*) ∧
(∀ *a b*. *a* ∈ *set hs* ⟶ *b* ∈ *set hs* ⟶ *a* ≠ *b* ⟶ (*a*, *b*) $\in_p$ *snd* ‘ *set* (*np gs bs hs data*)) ∧
(∀ *p q*. (*True*, *p*, *q*) ∈ *set* (*np gs bs hs data*) ⟶ *q* ∈ *set gs* ∪ *set bs*)
shows *np-spec np*
unfolding *np-spec-def* **using** *assms* **by** *meson*

lemma *np-specD1*:
assumes *np-spec np*
shows *snd* ‘ *set* (*np gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
using *assms*[*unfolded np-spec-def*, *rule-format*, *of gs bs hs data*] **..**

lemma *np-specD2*:
assumes *np-spec np*
shows *set hs* × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*np gs bs hs data*)
using *assms*[*unfolded np-spec-def*, *rule-format*, *of gs bs hs data*] **by** *auto*

lemma *np-specD3*:
assumes *np-spec np* **and** *a* ∈ *set hs* **and** *b* ∈ *set hs* **and** *a* ≠ *b*
shows (*a*, *b*) $\in_p$ *snd* ‘ *set* (*np gs bs hs data*)
using *assms*(*1*)[*unfolded np-spec-def*, *rule-format*, *of gs bs hs data*] *assms*(*2*,*3*,*4*) **by** *blast*

lemma *np-specD4*:
assumes *np-spec np* **and** (*True*, *p*, *q*) ∈ *set* (*np gs bs hs data*)
shows *q* ∈ *set gs* ∪ *set bs*
using *assms*(*1*)[*unfolded np-spec-def*, *rule-format*, *of gs bs hs data*] *assms*(*2*) **by** *blast*

lemma *np-specE*:
assumes *np-spec np* **and** *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* **and** *p* ≠ *q*
assumes *1*: ⋀*q-in-bs*. (*q-in-bs*, *p*, *q*) ∈ *set* (*np gs bs hs data*) ⟹ *thesis*
assumes *2*: ⋀*p-in-bs*. (*p-in-bs*, *q*, *p*) ∈ *set* (*np gs bs hs data*) ⟹ *thesis*

shows *thesis*
proof (*cases* *q* ∈ *set gs* ∪ *set bs*)
case *True*
with *assms*(*2*) **have** (*p*, *q*) ∈ *set hs* × (*set gs* ∪ *set bs*) **by** *simp*
also from *assms*(*1*) **have** ... ⊆ *snd* ' *set* (*np gs bs hs data*) **by** (*rule np-specD2*)
finally obtain *q-in-bs* **where** (*q-in-bs*, *p*, *q*) ∈ *set* (*np gs bs hs data*) **by** *fastforce*
thus *?thesis* **by** (*rule 1*)
next
case *False*
with *assms*(*3*) **have** *q* ∈ *set hs* **by** *simp*
from *assms*(*1*,*2*) *this assms*(*4*) **have** (*p*, *q*) $\in_p$ *snd* ' *set* (*np gs bs hs data*) **by** (*rule np-specD3*)
hence (*p*, *q*) ∈ *snd* ' *set* (*np gs bs hs data*) ∨ (*q*, *p*) ∈ *snd* ' *set* (*np gs bs hs data*)
by (*simp only*: *in-pair-iff*)
thus *?thesis*
proof
assume (*p*, *q*) ∈ *snd* ' *set* (*np gs bs hs data*)
then obtain *q-in-bs* **where** (*q-in-bs*, *p*, *q*) ∈ *set* (*np gs bs hs data*) **by** *fastforce*
thus *?thesis* **by** (*rule 1*)
next
assume (*q*, *p*) ∈ *snd* ' *set* (*np gs bs hs data*)
then obtain *p-in-bs* **where** (*p-in-bs*, *q*, *p*) ∈ *set* (*np gs bs hs data*) **by** *fastforce*
thus *?thesis* **by** (*rule 2*)
qed
qed

definition *add-pairs-single-naive* :: *'d* ⇒ (*'t*, *'b*::*zero*, *'c*) *apsT*
where *add-pairs-single-naive data flag gs bs h ps* = *ps* @ (*map* (λ*g*. (*flag*, *h*, *g*)) *gs*) @ (*map* (λ*b*. (*flag*, *h*, *b*)) *bs*)

lemma *set-add-pairs-single-naive*:
set (*add-pairs-single-naive data flag gs bs h ps*) = *set ps* ∪ *Pair flag* ' ({*h*} × (*set gs* ∪ *set bs*))
by (*auto simp add*: *add-pairs-single-naive-def Let-def*)

fun *add-pairs-single-sorted* :: ((*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) ⇒ (*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) ⇒ *bool*) ⇒
(*'t*, *'b*::*zero*, *'c*) *apsT* **where**
add-pairs-single-sorted - - [] [] - *ps* = *ps*|
add-pairs-single-sorted rel flag [] (*b* # *bs*) *h ps* =
add-pairs-single-sorted rel flag [] *bs h* (*insort-wrt rel* (*flag*, *h*, *b*) *ps*)|
add-pairs-single-sorted rel flag (*g* # *gs*) *bs h ps* =
add-pairs-single-sorted rel flag gs bs h (*insort-wrt rel* (*flag*, *h*, *g*) *ps*)

lemma *set-add-pairs-single-sorted*:
set (*add-pairs-single-sorted rel flag gs bs h ps*) = *set ps* ∪ *Pair flag* ' ({*h*} × (*set gs* ∪ *set bs*))
proof (*induct gs arbitrary*: *ps*)
case *Nil*

show *?case*
proof (*induct bs arbitrary*: *ps*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons b bs*)
show *?case* **by** (*simp add*: *Cons*)
qed
next
case (*Cons g gs*)
show *?case* **by** (*simp add*: *Cons*)
qed

primrec (**in** −) *pairs* :: ('*t*, '*b*, '*c*) *apsT* ⇒ *bool* ⇒ ('*t*, '*b*, '*c*) *pdata list* ⇒ (*bool* × ('*t*, '*b*, '*c*) *pdata-pair*) *list*
where
pairs - - [] = []|
pairs aps flag (*x* # *xs*) = *aps flag* [] *xs x* (*pairs aps flag xs*)

lemma *pairs-subset*:
assumes ⋀*gs bs h ps*. *set* (*aps flag gs bs h ps*) = *set ps* ∪ *Pair flag* ‘ ({*h*} × (*set gs* ∪ *set bs*))
shows *set* (*pairs aps flag xs*) ⊆ *Pair flag* ‘ (*set xs* × *set xs*)
proof (*induct xs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons* **have** *set* (*pairs aps flag xs*) ⊆ *Pair flag* ‘ (*set* (*x* # *xs*) × *set* (*x* # *xs*)) **by** *fastforce*
moreover have {*x*} × *set xs* ⊆ *set* (*x* # *xs*) × *set* (*x* # *xs*) **by** *fastforce*
ultimately show *?case* **by** (*auto simp add*: *assms*)
qed

lemma *in-pairsI*:
assumes ⋀*gs bs h ps*. *set* (*aps flag gs bs h ps*) = *set ps* ∪ *Pair flag* ‘ ({*h*} × (*set gs* ∪ *set bs*))
and *a* ≠ *b* **and** *a* ∈ *set xs* **and** *b* ∈ *set xs*
shows (*flag*, *a*, *b*) ∈ *set* (*pairs aps flag xs*) ∨ (*flag*, *b*, *a*) ∈ *set* (*pairs aps flag xs*)
using *assms*(*3*, *4*)
proof (*induct xs*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons*(*3*) **have** *d*: *b* = *x* ∨ *b* ∈ *set xs* **by** *simp*
from *Cons*(*2*) **have** *a* = *x* ∨ *a* ∈ *set xs* **by** *simp*
thus *?case*
proof

assume *a* = *x*
with *assms*(*2*) **have** *b* ≠ *x* **by** *simp*
with *d* **have** *b* ∈ *set xs* **by** *simp*
hence (*flag*, *a*, *b*) ∈ *set* (*pairs aps flag* (*x* # *xs*)) **by** (*simp add*: ‹*a* = *x*› *assms*(*1*))
thus *?thesis* **by** *simp*
next
assume *a* ∈ *set xs*
from *d* **show** *?thesis*
proof
assume *b* = *x*
from ‹*a* ∈ *set xs*› **have** (*flag*, *b*, *a*) ∈ *set* (*pairs aps flag* (*x* # *xs*)) **by** (*simp add*: ‹*b* = *x*› *assms*(*1*))
thus *?thesis* **by** *simp*
next
assume *b* ∈ *set xs*
with ‹*a* ∈ *set xs*› **have** (*flag*, *a*, *b*) ∈ *set* (*pairs aps flag xs*) ∨ (*flag*, *b*, *a*) ∈ *set* (*pairs aps flag xs*)
by (*rule Cons*(*1*))
thus *?thesis* **by** (*auto simp*: *assms*(*1*))
qed
qed
qed

corollary *in-pairsI′*:
assumes ⋀*gs bs h ps*. *set* (*aps flag gs bs h ps*) = *set ps* ∪ *Pair flag* ‘ ({*h*} × (*set gs* ∪ *set bs*))
and *a* ∈ *set xs* **and** *b* ∈ *set xs* **and** *a* ≠ *b*
shows (*a*, *b*) $\in_p$ *snd* ‘ *set* (*pairs aps flag xs*)
proof −
from *assms*(*1*,*4*,*2*,*3*) **have** (*flag*, *a*, *b*) ∈ *set* (*pairs aps flag xs*) ∨ (*flag*, *b*, *a*) ∈ *set* (*pairs aps flag xs*)
by (*rule in-pairsI*)
thus *?thesis*
proof
assume (*flag*, *a*, *b*) ∈ *set* (*pairs aps flag xs*)
hence *snd* (*flag*, *a*, *b*) ∈ *snd* ‘ *set* (*pairs aps flag xs*) **by** *fastforce*
thus *?thesis* **by** (*simp add*: *in-pair-iff*)
next
assume (*flag*, *b*, *a*) ∈ *set* (*pairs aps flag xs*)
hence *snd* (*flag*, *b*, *a*) ∈ *snd* ‘ *set* (*pairs aps flag xs*) **by** *fastforce*
thus *?thesis* **by** (*simp add*: *in-pair-iff*)
qed
qed

definition *new-pairs-naive* :: (′*t*, ′*b*::*zero*, ′*c*, ′*d*) *npT*
where *new-pairs-naive gs bs hs data* =
fold (*add-pairs-single-naive data True gs bs*) *hs* (*pairs* (*add-pairs-single-naive data*) *False hs*)

definition *new-pairs-sorted* :: (*nat* × *'d* ⇒ (*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) ⇒ (*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) ⇒ *bool*) ⇒
(*'t*, *'b*::*zero*, *'c*, *'d*) *npT*
where *new-pairs-sorted rel gs bs hs data* =
fold (*add-pairs-single-sorted* (*rel data*) *True gs bs*) *hs* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)

lemma *set-fold-aps*:
assumes ⋀*gs bs h ps*. *set* (*aps flag gs bs h ps*) = *set ps* ∪ *Pair flag* ‘ ({*h*} × (*set gs* ∪ *set bs*))
shows *set* (*fold* (*aps flag gs bs*) *hs ps*) = *Pair flag* ‘ (*set hs* × (*set gs* ∪ *set bs*)) ∪ *set ps*
proof (*induct hs arbitrary*: *ps*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons h hs*)
show *?case* **by** (*auto simp add*: *Cons assms*)
qed

lemma *set-new-pairs-naive*:
set (*new-pairs-naive gs bs hs data*) =
Pair True ‘ (*set hs* × (*set gs* ∪ *set bs*)) ∪ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*)
proof −
have *set* (*new-pairs-naive gs bs hs data*) =
Pair True ‘ (*set hs* × (*set gs* ∪ *set bs*)) ∪ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*)
unfolding *new-pairs-naive-def* **by** (*rule set-fold-aps*, *fact set-add-pairs-single-naive*)
thus *?thesis* **by** (*simp add*: *ac-simps*)
qed

lemma *set-new-pairs-sorted*:
set (*new-pairs-sorted rel gs bs hs data*) =
Pair True ‘ (*set hs* × (*set gs* ∪ *set bs*)) ∪ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)
proof −
have *set* (*new-pairs-sorted rel gs bs hs data*) =
Pair True ‘ (*set hs* × (*set gs* ∪ *set bs*)) ∪ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)
unfolding *new-pairs-sorted-def* **by** (*rule set-fold-aps*, *fact set-add-pairs-single-sorted*)
thus *?thesis* **by** (*simp add*: *set-merge-wrt ac-simps*)
qed

lemma (**in** −) *fst-snd-Pair* [*simp*]:
shows *fst* ∘ *Pair x* = (λ-. *x*) **and** *snd* ∘ *Pair x* = *id*
by *auto*

lemma *np-spec-new-pairs-naive*: *np-spec new-pairs-naive*
proof (*rule np-specI*)
fix *gs bs hs* :: (*'t*, *'b*, *'c*) *pdata list* **and** *data*::*nat* × *'d*
have *1*: *set hs* × (*set gs* ∪ *set bs*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** *fastforce*
have *set* (*pairs* (*add-pairs-single-naive data*) *False hs*) ⊆ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule pairs-subset*, *simp add*: *set-add-pairs-single-naive*)
hence *snd* ‘ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*) ⊆ *snd* ‘ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule image-mono*)
also have ... = *set hs* × *set hs* **by** (*simp add*: *image-comp*)
finally have *2*: *snd* ‘ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by *fastforce*

show *snd* ‘ *set* (*new-pairs-naive gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∧
set hs × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*new-pairs-naive gs bs hs data*) ∧
(∀ *a b*. *a* ∈ *set hs* ⟶ *b* ∈ *set hs* ⟶ *a* ≠ *b* ⟶ (*a*, *b*) $\in_p$ *snd* ‘ *set* (*new-pairs-naive gs bs hs data*)) ∧
(∀ *p q*. (*True*, *p*, *q*) ∈ *set* (*new-pairs-naive gs bs hs data*) ⟶ *q* ∈ *set gs* ∪ *set bs*)
proof (*intro conjI allI impI*)
show *snd* ‘ *set* (*new-pairs-naive gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*simp add*: *set-new-pairs-naive image-Un image-comp 1 2*)
next
show *set hs* × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*new-pairs-naive gs bs hs data*)
by (*simp add*: *set-new-pairs-naive image-Un image-comp*)
next
fix *a b*
assume *a* ∈ *set hs* **and** *b* ∈ *set hs* **and** *a* ≠ *b*
with *set-add-pairs-single-naive*
have (*a*, *b*) $\in_p$ *snd* ‘ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*)
by (*rule in-pairsI'*)
thus (*a*, *b*) $\in_p$ *snd* ‘ *set* (*new-pairs-naive gs bs hs data*)
by (*simp add*: *set-new-pairs-naive image-Un*)
next
fix *p q*
assume (*True*, *p*, *q*) ∈ *set* (*new-pairs-naive gs bs hs data*)
hence *q* ∈ *set gs* ∪ *set bs* ∨ (*True*, *p*, *q*) ∈ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*)
by (*auto simp*: *set-new-pairs-naive*)
thus *q* ∈ *set gs* ∪ *set bs*
proof
assume (*True*, *p*, *q*) ∈ *set* (*pairs* (*add-pairs-single-naive data*) *False hs*)
also from *set-add-pairs-single-naive* **have** ... ⊆ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule pairs-subset*)

finally show *?thesis* **by** *auto*
qed
qed
qed

lemma *np-spec-new-pairs-sorted*: *np-spec* (*new-pairs-sorted rel*)
proof (*rule np-specI*)
fix *gs bs hs* :: ($'t$, $'b$, $'c$) *pdata list* **and** *data*::*nat* × $'d$
have *1*: *set hs* × (*set gs* ∪ *set bs*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** *fastforce*
have *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*) ⊆ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule pairs-subset*, *simp add*: *set-add-pairs-single-sorted*)
hence *snd* ‘ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*) ⊆ *snd* ‘ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule image-mono*)
also have ... = *set hs* × *set hs* **by** (*simp add*: *image-comp*)
finally have *2*: *snd* ‘ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by *fastforce*

show *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∧
set hs × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*) ∧
(∀ *a b*. *a* ∈ *set hs* ⟶ *b* ∈ *set hs* ⟶ *a* ≠ *b* ⟶ (*a*, *b*) $\in_p$ *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*)) ∧
(∀ *p q*. (*True*, *p*, *q*) ∈ *set* (*new-pairs-sorted rel gs bs hs data*) ⟶ *q* ∈ *set gs* ∪ *set bs*)
proof (*intro conjI allI impI*)
show *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*) ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*simp add*: *set-new-pairs-sorted image-Un image-comp 1 2*)
next
show *set hs* × (*set gs* ∪ *set bs*) ⊆ *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*)
by (*simp add*: *set-new-pairs-sorted image-Un image-comp*)
next
fix *a b*
assume *a* ∈ *set hs* **and** *b* ∈ *set hs* **and** *a* ≠ *b*
with *set-add-pairs-single-sorted*
have (*a*, *b*) $\in_p$ *snd* ‘ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)
by (*rule in-pairsI′*)
thus (*a*, *b*) $\in_p$ *snd* ‘ *set* (*new-pairs-sorted rel gs bs hs data*)
by (*simp add*: *set-new-pairs-sorted image-Un*)
next
fix *p q*
assume (*True*, *p*, *q*) ∈ *set* (*new-pairs-sorted rel gs bs hs data*)
hence *q* ∈ *set gs* ∪ *set bs* ∨ (*True*, *p*, *q*) ∈ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)
by (*auto simp*: *set-new-pairs-sorted*)

thus *q* ∈ *set gs* ∪ *set bs*
proof
assume (*True*, *p*, *q*) ∈ *set* (*pairs* (*add-pairs-single-sorted* (*rel data*)) *False hs*)
also from *set-add-pairs-single-sorted* **have** ... ⊆ *Pair False* ‘ (*set hs* × *set hs*)
by (*rule pairs-subset*)
finally show *?thesis* **by** *auto*
qed
qed
qed

new-pairs-naive gs bs hs data and *new-pairs-sorted rel gs bs hs data* return lists of triples (*q-in-bs*, *p*, *q*), where *q-in-bs* indicates whether *q* is contained in the list *gs* @ *bs* or in the list *hs*. *p* is always contained in *hs*.

definition *canon-pair-order-aux* :: (*'t*, *'b*::*zero*, *'c*) *pdata-pair* ⇒ (*'t*, *'b*, *'c*) *pdata-pair* ⇒ *bool*
where *canon-pair-order-aux p q* ⟷
(*lcs* (*lp* (*fst* (*fst p*))) (*lp* (*fst* (*snd p*))) ⪯ *lcs* (*lp* (*fst* (*fst q*))) (*lp* (*fst* (*snd q*))))

abbreviation *canon-pair-order data p q* ≡ *canon-pair-order-aux* (*snd p*) (*snd q*)

abbreviation *canon-pair-comb* ≡ *merge-wrt canon-pair-order-aux*

6.3.4 Applying Criteria to New Pairs

definition *apply-icrit* :: (*'t*, *'b*, *'c*, *'d*) *icritT* ⇒ (*nat* × *'d*) ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*'t*, *'b*, *'c*) *pdata list* ⇒ (*'t*, *'b*, *'c*) *pdata list* ⇒
(*bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list* ⇒
(*bool* × *bool* × (*'t*, *'b*, *'c*) *pdata-pair*) *list*
where *apply-icrit crit data gs bs hs ps* = (*let c* = *crit data gs bs hs in map* (λ(*q-in-bs*, *p*, *q*). (*c p q*, *q-in-bs*, *p*, *q*)) *ps*)

lemma *fst-apply-icrit*:
assumes *icrit-spec crit* **and** *dickson-grading d*
and *fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m* **and** *unique-idx* (*gs* @ *bs* @ *hs*) *data*
and *is-Groebner-basis* (*fst* ‘ *set gs*) **and** *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs*
and *fst p* ≠ *0* **and** *fst q* ≠ *0* **and** (*True*, *q-in-bs*, *p*, *q*) ∈ *set* (*apply-icrit crit data gs bs hs ps*)
shows *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
proof −
from *assms*(*10*) **have** *crit data gs bs hs p q* **by** (*auto simp*: *apply-icrit-def*)
with *assms*(*1*−*9*) **show** *?thesis* **by** (*rule icrit-specD*)
qed

lemma *snd-apply-icrit* [*simp*]: *map snd* (*apply-icrit crit data gs bs hs ps*) = *ps*
by (*auto simp add*: *apply-icrit-def case-prod-beta'* *intro*: *nth-equalityI*)

lemma *set-snd-apply-icrit* [*simp*]: *snd ‘ set* (*apply-icrit crit data gs bs hs ps*) = *set ps*
proof −
have *snd ‘ set* (*apply-icrit crit data gs bs hs ps*) = *set* (*map snd* (*apply-icrit crit data gs bs hs ps*))
by (*simp del*: *snd-apply-icrit*)
also have ... = *set ps* **by** (*simp only*: *snd-apply-icrit*)
finally show *?thesis* **.**
qed

definition *apply-ncrit* :: (′*t*, ′*b*, ′*c*, ′*d*) *ncritT* ⇒ (*nat* × ′*d*) ⇒ (′*t*, ′*b*, ′*c*) *pdata list* ⇒
(′*t*, ′*b*, ′*c*) *pdata list* ⇒ (′*t*, ′*b*, ′*c*) *pdata list* ⇒
(*bool* × *bool* × (′*t*, ′*b*, ′*c*) *pdata-pair*) *list* ⇒
(*bool* × (′*t*, ′*b*, ′*c*) *pdata-pair*) *list*
where *apply-ncrit crit data gs bs hs ps* =
(*let c* = *crit data gs bs hs in*
rev (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*). λ*ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps* []))

lemma *apply-ncrit-append*:
apply-ncrit crit data gs bs hs (*xs* @ *ys*) =
rev (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*). λ*ps*′. *if* ¬ *ic* ∧ *crit data gs bs hs q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ys*
(*rev* (*apply-ncrit crit data gs bs hs xs*)))
by (*simp add*: *apply-ncrit-def Let-def*)

lemma *fold-superset*:
set acc ⊆
set (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*). λ*ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps acc*)
proof (*induct ps arbitrary*: *acc*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x ps*)
obtain *ic*′ *q-in-bs*′ *p*′ *q*′ **where** *x*: *x* = (*ic*′, *q-in-bs*′, *p*′, *q*′) **using** *prod-cases4* **by** *blast*
have *1*: *set acc0* ⊆ *set* (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps acc0*)
for *acc0* **by** (*rule Cons*)
have *set acc* ⊆ *set* ((*ic*′, *p*′, *q*′) # *acc*) **by** *fastforce*
also have ... ⊆ *set* (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps*
((*ic*′, *p*′, *q*′) # *acc*)) **by** (*fact 1*)
finally have *2*: *set acc* ⊆ *set* (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps*
((*ic*′, *p*′, *q*′) # *acc*)) **.**

show *?case* **by** (*simp add*: *x 1 2*)
qed

lemma *apply-ncrit-superset*:
set (*apply-ncrit crit data gs bs hs ps*) ⊆ *set* (*apply-ncrit crit data gs bs hs* (*ps* @ *qs*)) (**is** *?l* ⊆ *?r*)
proof −
have *?l* = *set* (*rev* (*apply-ncrit crit data gs bs hs ps*)) **by** *simp*
also have ... ⊆ *set* (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps*′.
if ¬ *ic* ∧ *crit data gs bs hs q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′)
qs (*rev* (*apply-ncrit crit data gs bs hs ps*))) **by** (*fact fold-superset*)
also have ... = *?r* **by** (*simp add*: *apply-ncrit-append*)
finally show *?thesis* **.**
qed

lemma *apply-ncrit-subset-aux*:
assumes (*ic*, *p*, *q*) ∈ *set* (*fold*
(λ(*ic*, *q-in-bs*, *p*, *q*). λ*ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps acc*)
shows (*ic*, *p*, *q*) ∈ *set acc* ∨ (∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*)
using *assms*
proof (*induct ps arbitrary*: *acc*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x ps*)
obtain *ic*′ *q-in-bs*′ *p*′ *q*′ **where** *x*: *x* = (*ic*′, *q-in-bs*′, *p*′, *q*′) **using** *prod-cases4* **by** *blast*
from *Cons*(*2*) **have** (*ic*, *p*, *q*) ∈
set (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps*′. *if* ¬ *ic* ∧ *c q-in-bs ps*′ *p q then ps*′ *else* (*ic*, *p*, *q*) # *ps*′) *ps*
(*if* ¬ *ic*′ ∧ *c q-in-bs*′ *acc p*′ *q*′ *then acc else* (*ic*′, *p*′, *q*′) # *acc*)) **by** (*simp add*: *x*)
hence (*ic*, *p*, *q*) ∈ *set* (*if* ¬ *ic*′ ∧ *c q-in-bs*′ *acc p*′ *q*′ *then acc else* (*ic*′, *p*′, *q*′) # *acc*) ∨
(∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*) **by** (*rule Cons*(*1*))
hence (*ic*, *p*, *q*) ∈ *set acc* ∨ (*ic*, *p*, *q*) = (*ic*′, *p*′, *q*′) ∨ (∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*)
by (*auto split*: *if-splits*)
thus *?case*
proof (*elim disjE*)
assume (*ic*, *p*, *q*) ∈ *set acc*
thus *?thesis* **..**
next
assume (*ic*, *p*, *q*) = (*ic*′, *p*′, *q*′)
hence *x* = (*ic*, *q-in-bs*′, *p*, *q*) **by** (*simp add*: *x*)
thus *?thesis* **by** *auto*
next

assume ∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*
then obtain *q-in-bs* **where** (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps* ..
thus *?thesis* **by** *auto*
qed
qed

corollary *apply-ncrit-subset*:
assumes (*ic*, *p*, *q*) ∈ *set* (*apply-ncrit crit data gs bs hs ps*)
obtains *q-in-bs* **where** (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*
proof −
from *assms*
have (*ic*, *p*, *q*) ∈ *set* (*fold*
(λ(*ic*, *q-in-bs*, *p*, *q*). λ*ps'*. *if* ¬ *ic* ∧ *crit data gs bs hs q-in-bs ps' p q then ps' else* (*ic*, *p*, *q*) # *ps'*) *ps* [])
by (*simp add*: *apply-ncrit-def*)
hence (*ic*, *p*, *q*) ∈ *set* [] ∨ (∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps*)
by (*rule apply-ncrit-subset-aux*)
hence ∃ *q-in-bs*. (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps* **by** *simp*
then obtain *q-in-bs* **where** (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps* ..
thus *?thesis* ..
qed

corollary *apply-ncrit-subset'*: *snd* ' *set* (*apply-ncrit crit data gs bs hs ps*) ⊆ *snd* ' *snd* ' *set ps*
proof
fix *p q*
assume (*p*, *q*) ∈ *snd* ' *set* (*apply-ncrit crit data gs bs hs ps*)
then obtain *ic* **where** (*ic*, *p*, *q*) ∈ *set* (*apply-ncrit crit data gs bs hs ps*) **by** *fastforce*
then obtain *q-in-bs* **where** (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps* **by** (*rule apply-ncrit-subset*)
thus (*p*, *q*) ∈ *snd* ' *snd* ' *set ps* **by** *force*
qed

lemma *not-in-apply-ncrit*:
assumes (*ic*, *p*, *q*) ∉ *set* (*apply-ncrit crit data gs bs hs* (*xs* @ ((*ic*, *q-in-bs*, *p*, *q*) # *ys*)))
shows *crit data gs bs hs q-in-bs* (*rev* (*apply-ncrit crit data gs bs hs xs*)) *p q*
using *assms*
proof (*simp add*: *apply-ncrit-append split*: *if-splits*)
assume (*ic*, *p*, *q*) ∉
set (*fold* (λ(*ic*, *q-in-bs*, *p*, *q*) *ps'*. *if* ¬ *ic* ∧ *crit data gs bs hs q-in-bs ps' p q then ps' else* (*ic*, *p*, *q*) # *ps'*)
ys ((*ic*, *p*, *q*) # *rev* (*apply-ncrit crit data gs bs hs xs*))) (**is** - ∉ *?A*)
have (*ic*, *p*, *q*) ∈ *set* ((*ic*, *p*, *q*) # *rev* (*apply-ncrit crit data gs bs hs xs*)) **by** *simp*
also have ... ⊆ *?A* **by** (*rule fold-superset*)
finally have (*ic*, *p*, *q*) ∈ *?A* .
with ‹(*ic*, *p*, *q*) ∉ *?A*› **show** *?thesis* ..
qed

lemma (**in** −) *setE*:
  assumes $x \in set\ xs$
  obtains *ys zs* **where** $xs = ys\ @\ (x\ \#\ zs)$
  using *assms*
proof (*induct xs arbitrary*: *thesis*)
  case *Nil*
  from *Nil*(*2*) **show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  from *Cons*(*3*) **have** $x = a \vee x \in set\ xs$ **by** *simp*
  thus *?case*
  proof
    assume $x = a$
    show *?thesis* **by** (*rule Cons*(*2*)[*of* [] *xs*], *simp add*: ‹$x = a$›)
  next
    assume $x \in set\ xs$
    then obtain *ys zs* **where** $xs = ys\ @\ (x\ \#\ zs)$ **by** (*meson Cons*(*1*))
    show *?thesis* **by** (*rule Cons*(*2*)[*of a* # *ys zs*], *simp add*: ‹$xs = ys\ @\ (x\ \#\ zs)$›)
  qed
qed

lemma *apply-ncrit-connectible*:
  assumes *ncrit-spec crit* **and** *dickson-grading d*
    and $set\ gs \cup set\ bs \cup set\ hs \subseteq B$ **and** *fst* ‘ $B \subseteq$ *dgrad-p-set d m*
    and *snd* ‘ *snd* ‘ *set ps* $\subseteq$ *set hs* $\times$ (*set gs* $\cup$ *set bs* $\cup$ *set hs*) **and** *unique-idx* (*gs* @ *bs* @ *hs*) *data*
    and *is-Groebner-basis* (*fst* ‘ *set gs*)
    and $\bigwedge p'\ q'.\ (p', q') \in$ *snd* ‘ *set* (*apply-ncrit crit data gs bs hs ps*) $\Longrightarrow$
                fst $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst* p') (*fst* q')
    and $\bigwedge p'\ q'.\ p' \in set\ gs \cup set\ bs \Longrightarrow q' \in set\ gs \cup set\ bs \Longrightarrow$ *fst* $p' \neq 0 \Longrightarrow$ *fst* $q' \neq 0 \Longrightarrow$
                crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst* p') (*fst* q')
  assumes (*ic*, *q-in-bs*, *p*, *q*) $\in$ *set ps* **and** *fst* $p \neq 0$ **and** *fst* $q \neq 0$
    and *q-in-bs* $\Longrightarrow$ ($q \in set\ gs \cup set\ bs$)
  shows *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
proof (*cases* (*p*, *q*) $\in$ *snd* ‘ *set* (*apply-ncrit crit data gs bs hs ps*))
  case *True*
  thus *?thesis* **using** *assms*(*11*,*12*) **by** (*rule assms*(*8*))
next
  case *False*
  from *assms*(*10*) **have** (*p*, *q*) $\in$ *snd* ‘ *snd* ‘ *set ps* **by** *force*
  also have ... $\subseteq$ *set hs* $\times$ (*set gs* $\cup$ *set bs* $\cup$ *set hs*) **by** (*fact assms*(*5*))
  finally have $p \in set\ hs$ **and** $q \in set\ gs \cup set\ bs \cup set\ hs$ **by** *simp-all*
  from ‹(*ic*, *q-in-bs*, *p*, *q*) $\in$ *set ps*› **obtain** *xs ys* **where** *ps*: *ps* = *xs* @ ((*ic*, *q-in-bs*, *p*, *q*) # *ys*)
    by (*rule setE*)

  let *?ps* = *rev* (*apply-ncrit crit data gs bs hs xs*)

have *snd ‘ set ?ps* ⊆ *snd ‘ snd ‘ set xs* **by** (*simp add*: *apply-ncrit-subset′*)
also have ... ⊆ *snd ‘ snd ‘ set ps* **unfolding** *ps* **by** *fastforce*
finally have *sub*: *snd ‘ set ?ps* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
using *assms*(*5*) **by** (*rule subset-trans*)
from *False* **have** (*p*, *q*) ∉ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*) **by** (*simp add*: *in-pair-iff*)
hence (*ic*, *p*, *q*) ∉ *set* (*apply-ncrit crit data gs bs hs* (*xs* @ ((*ic*, *q-in-bs*, *p*, *q*) # *ys*)))
unfolding *ps* **by** *force*
hence *crit data gs bs hs q-in-bs ?ps p q* **by** (*rule not-in-apply-ncrit*)
with *assms*(*1*−*4*) *sub assms*(*6*,*7*,*13*) - - ‹*p* ∈ *set hs*› ‹*q* ∈ *set gs* ∪ *set bs* ∪ *set hs*› *assms*(*11*,*12*)
show *?thesis*
proof (*rule ncrit-specD*)
fix *p′ q′*
assume (*p′*, *q′*) $\in_p$ *snd ‘ set ?ps*
also have ... ⊆ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*)
by (*rule image-mono*, *simp add*: *ps apply-ncrit-superset*)
finally have *disj*: (*p′*, *q′*) ∈ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*) ∨
(*q′*, *p′*) ∈ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*) **by** (*simp only*: *in-pair-iff*)
assume *fst p′* ≠ *0* **and** *fst q′* ≠ *0*
from *disj* **show** *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst p′*) (*fst q′*)
proof
assume (*p′*, *q′*) ∈ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*)
thus *?thesis* **using** ‹*fst p′* ≠ *0*› ‹*fst q′* ≠ *0*› **by** (*rule assms*(*8*))
next
assume (*q′*, *p′*) ∈ *snd ‘ set* (*apply-ncrit crit data gs bs hs ps*)
hence *crit-pair-cbelow-on d m* (*fst ‘ B*) (*fst q′*) (*fst p′*)
using ‹*fst q′* ≠ *0*› ‹*fst p′* ≠ *0*› **by** (*rule assms*(*8*))
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed (*assumption*, *fact assms*(*9*))
qed

6.3.5 Applying Criteria to Old Pairs

definition *apply-ocrit* :: (*′t*, *′b*, *′c*, *′d*) *ocritT* ⇒ (*nat* × *′d*) ⇒ (*′t*, *′b*, *′c*) *pdata list* ⇒
(*bool* × (*′t*, *′b*, *′c*) *pdata-pair*) *list* ⇒ (*′t*, *′b*, *′c*) *pdata-pair list* ⇒
(*′t*, *′b*, *′c*) *pdata-pair list*
where *apply-ocrit crit data hs ps′ ps* = (*let c* = *crit data hs ps′ in* [(*p*, *q*)←*ps* . ¬ *c p q*])

lemma *set-apply-ocrit*:
set (*apply-ocrit crit data hs ps′ ps*) = {(*p*, *q*) | *p q*. (*p*, *q*) ∈ *set ps* ∧ ¬ *crit data hs ps′ p q*}
by (*auto simp*: *apply-ocrit-def*)

corollary *set-apply-ocrit-iff*:
$(p, q) \in$ *set* (*apply-ocrit crit data hs ps' ps*) $\longleftrightarrow$ $((p, q) \in$ *set ps* $\wedge$ $\neg$ *crit data hs ps' p q*)
by (*auto simp*: *apply-ocrit-def*)

lemma *apply-ocrit-connectible*:
assumes *ocrit-spec crit* **and** *dickson-grading d* **and** *set hs* $\subseteq$ *B* **and** *fst* ‘ *B* $\subseteq$ *dgrad-p-set d m*
and *unique-idx* (*p* # *q* # *hs* @ (*map* (*fst* ∘ *snd*) *ps'*) @ (*map* (*snd* ∘ *snd*) *ps'*)) *data*
and $\bigwedge p'\ q'$. $(p', q') \in$ *snd* ‘ *set ps'* $\Longrightarrow$ *fst p'* $\neq$ *0* $\Longrightarrow$ *fst q'* $\neq$ *0* $\Longrightarrow$
crit-pair-cbelow-on d m (*fst* ‘ *B*) (*fst p'*) (*fst q'*)
assumes $p \in B$ **and** $q \in B$ **and** *fst p* $\neq$ *0* **and** *fst q* $\neq$ *0*
and $(p, q) \in$ *set ps* **and** $(p, q) \notin$ *set* (*apply-ocrit crit data hs ps' ps*)
shows *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
proof −
from *assms*(*11*,*12*) **have** *crit data hs ps' p q* **by** (*simp add*: *set-apply-ocrit-iff*)
with *assms*(*1*−*5*) - *assms*(*7*−*10*) **show** *?thesis*
proof (*rule ocrit-specD*)
fix *p' q'*
assume $(p', q') \in_p$ *snd* ‘ *set ps'*
hence *disj*: $(p', q') \in$ *snd* ‘ *set ps'* $\vee$ $(q', p') \in$ *snd* ‘ *set ps'* **by** (*simp only*: *in-pair-iff*)
assume *fst p'* $\neq$ *0* **and** *fst q'* $\neq$ *0*
from *disj* **show** *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p'*) (*fst q'*)
proof
assume $(p', q') \in$ *snd* ‘ *set ps'*
thus *?thesis* **using** ‹*fst p'* $\neq$ *0*› ‹*fst q'* $\neq$ *0*› **by** (*rule assms*(*6*))
next
assume $(q', p') \in$ *snd* ‘ *set ps'*
hence *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst q'*) (*fst p'*) **using** ‹*fst q'* $\neq$ *0*› ‹*fst p'* $\neq$ *0*›
by (*rule assms*(*6*))
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed
qed

6.3.6 Creating Final List of Pairs

context
fixes *np*::('*t*, '*b*::*field*, '*c*, '*d*) *npT*
and *icrit*::('*t*, '*b*, '*c*, '*d*) *icritT*
and *ncrit*::('*t*, '*b*, '*c*, '*d*) *ncritT*
and *ocrit*::('*t*, '*b*, '*c*, '*d*) *ocritT*
and *comb*::('*t*, '*b*, '*c*) *pdata-pair list* ⇒ ('*t*, '*b*, '*c*) *pdata-pair list* ⇒ ('*t*, '*b*, '*c*) *pdata-pair list*
begin

definition *add-pairs* :: ('t, 'b, 'c, 'd) *apT*
where *add-pairs gs bs ps hs data* =
(*let ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*));
ps2 = *apply-ocrit ocrit data hs ps1 ps in comb* (*map snd* [*x*←*ps1* . ¬ *fst x*]) *ps2*)

lemma *set-add-pairs*:
assumes ⋀*xs ys*. *set* (*comb xs ys*) = *set xs* ∪ *set ys*
assumes *ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
shows *set* (*add-pairs gs bs ps hs data*) =
{(*p*, *q*) | *p q*. (*False*, *p*, *q*) ∈ *set ps1* ∨ ((*p*, *q*) ∈ *set ps* ∧ ¬ *ocrit data hs ps1 p q*)}
proof −
have *eq*: *snd* ' {*x* ∈ *set ps1*. ¬ *fst x*} = {(*p*, *q*) | *p q*. (*False*, *p*, *q*) ∈ *set ps1*} **by** *force*
thus *?thesis* **by** (*auto simp*: *add-pairs-def Let-def assms*(*1*) *assms*(*2*)[*symmetric*] *set-apply-ocrit*)
qed

lemma *set-add-pairs-iff*:
assumes ⋀*xs ys*. *set* (*comb xs ys*) = *set xs* ∪ *set ys*
assumes *ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
shows ((*p*, *q*) ∈ *set* (*add-pairs gs bs ps hs data*)) ⟷
((*False*, *p*, *q*) ∈ *set ps1* ∨ ((*p*, *q*) ∈ *set ps* ∧ ¬ *ocrit data hs ps1 p q*))
proof −
from *assms* **have** *eq*: *set* (*add-pairs gs bs ps hs data*) =
{(*p*, *q*) | *p q*. (*False*, *p*, *q*) ∈ *set ps1* ∨ ((*p*, *q*) ∈ *set ps* ∧ ¬ *ocrit data hs ps1 p q*)}
by (*rule set-add-pairs*)
obtain *a aa b* **where** *p*: *p* = (*a*, *aa*, *b*) **using** *prod-cases3* **by** *blast*
obtain *ab ac ba* **where** *q*: *q* = (*ab*, *ac*, *ba*) **using** *prod-cases3* **by** *blast*
show *?thesis* **by** (*simp add*: *eq p q*)
qed

lemma *ap-spec-add-pairs*:
assumes *np-spec np* **and** *icrit-spec icrit* **and** *ncrit-spec ncrit* **and** *ocrit-spec ocrit*
and ⋀*xs ys*. *set* (*comb xs ys*) = *set xs* ∪ *set ys*
shows *ap-spec add-pairs*
proof (*rule ap-specI*)
fix *gs bs* :: ('t, 'b, 'c) *pdata list* **and** *ps hs* **and** *data*::*nat* × '*d*
define *ps1* **where** *ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
show *set* (*add-pairs gs bs ps hs data*) ⊆ *set ps* ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
proof

fix *p q*
assume (*p*, *q*) ∈ *set* (*add-pairs gs bs ps hs data*)
with *assms*(*5*) *ps1-def* **have** (*False*, *p*, *q*) ∈ *set ps1* ∨ ((*p*, *q*) ∈ *set ps* ∧ ¬ *ocrit data hs ps1 p q*)
by (*simp add*: *set-add-pairs-iff*)
thus (*p*, *q*) ∈ *set ps* ∪ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
proof
assume (*False*, *p*, *q*) ∈ *set ps1*
hence *snd* (*False*, *p*, *q*) ∈ *snd* ' *set ps1* **by** *fastforce*
hence (*p*, *q*) ∈ *snd* ' *set ps1* **by** *simp*
also have ... ⊆ *snd* ' *snd* ' *set* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
unfolding *ps1-def* **by** (*fact apply-ncrit-subset'*)
also have ... = *snd* ' *set* (*np gs bs hs data*) **by** *simp*
also from *assms*(*1*) **have** ... ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** (*rule np-specD1*)
finally show *?thesis* **..**
next
assume (*p*, *q*) ∈ *set ps* ∧ ¬ *ocrit data hs ps1 p q*
thus *?thesis* **by** *simp*
qed
qed
next
fix *gs bs* :: (*'t*, *'b*, *'c*) *pdata list* **and** *ps hs* **and** *data*::*nat* × *'d* **and** *B* **and** *d*::*'a* ⇒ *nat* **and** *m h g*
assume *dg*: *dickson-grading d* **and** *B-sup*: *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B*
and *B-sub*: *fst* ' *B* ⊆ *dgrad-p-set d m* **and** *h-in*: *h* ∈ *set hs* **and** *g-in*: *g* ∈ *set gs* ∪ *set bs* ∪ *set hs*
and *ps-sub*: *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and *uid*: *unique-idx* (*gs* @ *bs* @ *hs*) *data* **and** *gb*: *is-Groebner-basis* (*fst* ' *set gs*) **and** *h* ≠ *g*
and *fst h* ≠ *0* **and** *fst g* ≠ *0*
assume *a*: $\bigwedge$*a b*. (*a*, *b*) $\in_p$ *set* (*add-pairs gs bs ps hs data*) ⟹
fst a ≠ *0* ⟹ *fst b* ≠ *0* ⟹ *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst a*) (*fst b*)
assume *b*: $\bigwedge$*a b*. *a* ∈ *set gs* ∪ *set bs* ⟹
b ∈ *set gs* ∪ *set bs* ⟹
fst a ≠ *0* ⟹ *fst b* ≠ *0* ⟹ *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst a*) (*fst b*)
define *ps0* **where** *ps0* = *apply-icrit icrit data gs bs hs* (*np gs bs hs data*)
define *ps1* **where** *ps1* = *apply-ncrit ncrit data gs bs hs ps0*

have *snd* ' *snd* ' *set ps0* = *snd* ' *set* (*np gs bs hs data*) **by** (*simp add*: *ps0-def*)
also from *assms*(*1*) **have** ... ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** (*rule np-specD1*)
finally have *ps0-sub*: *snd* ' *snd* ' *set ps0* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **.**

have *crit-pair-cbelow-on d m* (*fst* ' *B*) (*fst p*) (*fst q*)
if (*p*, *q*) ∈ *snd* ' *set ps1* **and** *fst p* ≠ *0* **and** *fst q* ≠ *0* **for** *p q*

proof −
from ‹(*p*, *q*) ∈ *snd* ‘ *set ps1*› **obtain** *ic* **where** (*ic*, *p*, *q*) ∈ *set ps1* **by** *fastforce*
show *?thesis*
proof (*cases ic*)
case *True*
from ‹(*ic*, *p*, *q*) ∈ *set ps1*› **obtain** *q-in-bs* **where** (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps0*
unfolding *ps1-def* **by** (*rule apply-ncrit-subset*)
with *True* **have** (*True*, *q-in-bs*, *p*, *q*) ∈ *set ps0* **by** *simp*
hence *snd* (*snd* (*True*, *q-in-bs*, *p*, *q*)) ∈ *snd* ‘ *snd* ‘ *set ps0* **by** *fastforce*
hence (*p*, *q*) ∈ *snd* ‘ *snd* ‘ *set ps0* **by** *simp*
also have ... ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** (*fact ps0-sub*)
finally have *p* ∈ *set hs* **and** *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
from *B-sup* **have** *B-sup'*: *fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *fst* ‘ *B* **by** (*rule image-mono*)
hence *fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m* **using** *B-sub* **by** (*rule subset-trans*)
from *assms*(*2*) *dg this uid gb* ‹*p* ∈ *set hs*› ‹*q* ∈ *set gs* ∪ *set bs* ∪ *set hs*› ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*›
‹(*True*, *q-in-bs*, *p*, *q*) ∈ *set ps0*›
have *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
unfolding *ps0-def* **by** (*rule fst-apply-icrit*)
thus *?thesis* **using** *B-sup'* **by** (*rule crit-pair-cbelow-mono*)
next
case *False*
with ‹(*ic*, *p*, *q*) ∈ *set ps1*› **have** (*False*, *p*, *q*) ∈ *set ps1* **by** *simp*
with *assms*(*5*) *ps1-def* **have** (*p*, *q*) ∈ *set* (*add-pairs gs bs ps hs data*)
by (*simp add*: *set-add-pairs-iff ps0-def*)
hence (*p*, *q*) $\in_p$ *set* (*add-pairs gs bs ps hs data*) **by** (*simp add*: *in-pair-iff*)
thus *?thesis* **using** ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*› **by** (*rule a*)
qed
qed
with *assms*(*3*) *dg B-sup B-sub ps0-sub uid gb*
have ∗: (*ic*, *q-in-bs*, *p*, *q*) ∈ *set ps0* ⟹ *fst p* ≠ *0* ⟹ *fst q* ≠ *0* ⟹
(*q-in-bs* ⟹ *q* ∈ *set gs* ∪ *set bs*) ⟹ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
for *ic q-in-bs p q* **using** *b* **unfolding** *ps1-def* **by** (*rule apply-ncrit-connectible*)

show *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst h*) (*fst g*)
proof (*cases h* = *g*)
case *True*
from *g-in B-sup* **have** *g* ∈ *B* **..**
hence *fst g* ∈ *fst* ‘ *B* **by** *simp*
hence *fst g* ∈ *dgrad-p-set d m* **using** *B-sub* **..**
with *dg* **show** *?thesis* **unfolding** *True* **by** (*rule crit-pair-cbelow-same*)
next
case *False*
with *assms*(*1*) *h-in g-in* **show** *?thesis*
proof (*rule np-specE*)
fix *g-in-bs*

assume (*g-in-bs*, *h*, *g*) ∈ *set* (*np gs bs hs data*)
also have ... = *snd* ‘ *set ps0* **by** (*simp add*: *ps0-def*)
finally obtain *ic* **where** (*ic*, *g-in-bs*, *h*, *g*) ∈ *set ps0* **by** *fastforce*
moreover note ‹*fst h* ≠ *0*› ‹*fst g* ≠ *0*›
moreover from *assms*(*1*) **have** *g* ∈ *set gs* ∪ *set bs* **if** *g-in-bs*
proof (*rule np-specD4*)
from ‹(*g-in-bs*, *h*, *g*) ∈ *set* (*np gs bs hs data*)› *that* **show** (*True*, *h*, *g*) ∈ *set* (*np gs bs hs data*)
by *simp*
qed
ultimately show *?thesis* **by** (*rule* ∗)
next
fix *h-in-bs*
assume (*h-in-bs*, *g*, *h*) ∈ *set* (*np gs bs hs data*)
also have ... = *snd* ‘ *set ps0* **by** (*simp add*: *ps0-def*)
finally obtain *ic* **where** (*ic*, *h-in-bs*, *g*, *h*) ∈ *set ps0* **by** *fastforce*
moreover note ‹*fst g* ≠ *0*› ‹*fst h* ≠ *0*›
moreover from *assms*(*1*) **have** *h* ∈ *set gs* ∪ *set bs* **if** *h-in-bs*
proof (*rule np-specD4*)
from ‹(*h-in-bs*, *g*, *h*) ∈ *set* (*np gs bs hs data*)› *that* **show** (*True*, *g*, *h*) ∈ *set* (*np gs bs hs data*)
by *simp*
qed
ultimately have *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst g*) (*fst h*) **by** (*rule* ∗)
thus *?thesis* **by** (*rule crit-pair-cbelow-sym*)
qed
qed
next
fix *gs bs* :: (′*t*, ′*b*, ′*c*) *pdata list* **and** *ps hs* **and** *data*::*nat* × ′*d* **and** *B* **and** *d*::′*a* ⇒ *nat* **and** *m h g*
define *ps1* **where** *ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
assume (*h*, *g*) ∈ *set ps* $-_p$ *set* (*add-pairs gs bs ps hs data*)
hence (*h*, *g*) ∈ *set ps* **and** (*h*, *g*) $\notin_p$ *set* (*add-pairs gs bs ps hs data*) **by** *simp-all*
from *this*(*2*) **have** (*h*, *g*) ∉ *set* (*add-pairs gs bs ps hs data*) **by** (*simp add*: *in-pair-iff*)
assume *dg*: *dickson-grading d* **and** *B-sup*: *set gs* ∪ *set bs* ∪ *set hs* ⊆ *B* **and** *B-sub*: *fst* ‘ *B* ⊆ *dgrad-p-set d m*
and *ps-sub*: *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and (*set gs* ∪ *set bs*) ∩ *set hs* = {} — unused
and *uid*: *unique-idx* (*gs* @ *bs* @ *hs*) *data* **and** *gb*: *is-Groebner-basis* (*fst* ‘ *set gs*)
and *h* ≠ *g* **and** *fst h* ≠ *0* **and** *fst g* ≠ *0*
assume ∗: ⋀*a b*. (*a*, *b*) $\in_p$ *set* (*add-pairs gs bs ps hs data*) ⟹
(*a*, *b*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ⟹
fst a ≠ *0* ⟹ *fst b* ≠ *0* ⟹ *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst a*) (*fst b*)

have *snd* ‘ *set ps1* ⊆ *snd* ‘ *snd* ‘ *set* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))

unfolding *ps1-def* **by** (*rule apply-ncrit-subset'*)
also have ... = *snd* ‘ *set* (*np gs bs hs data*) **by** *simp*
also from *assms*(*1*) **have** ... ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **by** (*rule np-specD1*)
finally have *ps1-sub*: *snd* ‘ *set ps1* ⊆ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **.**

from ‹(*h*, *g*) ∈ *set ps*› *ps-sub* **have** *h-in*: *h* ∈ *set gs* ∪ *set bs* **and** *g-in*: *g* ∈ *set gs* ∪ *set bs*
by *fastforce*+
with *B-sup* **have** *h* ∈ *B* **and** *g* ∈ *B* **by** *auto*
with *assms*(*4*) *dg* - *B-sub* - - **show** *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst h*) (*fst g*)
using ‹*fst h* ≠ *0*› ‹*fst g* ≠ *0*› ‹(*h*, *g*) ∈ *set ps*›
proof (*rule apply-ocrit-connectible*)
from *B-sup* **show** *set hs* ⊆ *B* **by** *simp*
next
from *ps1-sub h-in g-in*
have *set* (*h* # *g* # *hs* @ *map* (*fst* ∘ *snd*) *ps1* @ *map* (*snd* ∘ *snd*) *ps1*) ⊆ *set* (*gs* @ *bs* @ *hs*)
by *fastforce*
with *uid* **show** *unique-idx* (*h* # *g* # *hs* @ *map* (*fst* ∘ *snd*) *ps1* @ *map* (*snd* ∘ *snd*) *ps1*) *data*
by (*rule unique-idx-subset*)
next
fix *p q*
assume (*p*, *q*) ∈ *snd* ‘ *set ps1*
hence *pq-in*: (*p*, *q*) ∈ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) **using** *ps1-sub* **..**
hence *p-in*: *p* ∈ *set hs* **and** *q-in*: *q* ∈ *set gs* ∪ *set bs* ∪ *set hs* **by** *simp-all*
assume *fst p* ≠ *0* **and** *fst q* ≠ *0*
from ‹(*p*, *q*) ∈ *snd* ‘ *set ps1*› **obtain** *ic* **where** (*ic*, *p*, *q*) ∈ *set ps1* **by** *fastforce*
show *crit-pair-cbelow-on d m* (*fst* ‘ *B*) (*fst p*) (*fst q*)
proof (*cases ic*)
case *True*
hence *ic* = *True* **by** *simp*
from *B-sup* **have** *B-sup'*: *fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *fst* ‘ *B* **by** (*rule image-mono*)
note *assms*(*2*) *dg*
moreover from *B-sup' B-sub* **have** *fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*) ⊆ *dgrad-p-set d m*
by (*rule subset-trans*)
moreover note *uid gb p-in q-in* ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*›
moreover from ‹(*ic*, *p*, *q*) ∈ *set ps1*› **obtain** *q-in-bs*
where (*True*, *q-in-bs*, *p*, *q*) ∈ *set* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))
unfolding *ps1-def* ‹*ic* = *True*› **by** (*rule apply-ncrit-subset*)
ultimately have *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set bs* ∪ *set hs*)) (*fst p*) (*fst q*)
by (*rule fst-apply-icrit*)
thus *?thesis* **using** *B-sup'* **by** (*rule crit-pair-cbelow-mono*)

next
case *False*
with ‹(*ic*, *p*, *q*) ∈ *set ps1*› **have** (*False*, *p*, *q*) ∈ *set ps1* **by** *simp*
with *assms*(*5*) *ps1-def* **have** (*p*, *q*) ∈ *set* (*add-pairs gs bs ps hs data*)
by (*simp add*: *set-add-pairs-iff*)
hence (*p*, *q*) $\in_p$ *set* (*add-pairs gs bs ps hs data*) **by** (*simp add*: *in-pair-iff*)
moreover from *pq-in* **have** (*p*, *q*) $\in_p$ *set hs* × (*set gs* ∪ *set bs* ∪ *set hs*)
by (*simp add*: *in-pair-iff*)
ultimately show *?thesis* **using** ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*› **by** (*rule* ∗)
qed
next
show (*h*, *g*) ∉ *set* (*apply-ocrit ocrit data hs ps1 ps*)
proof
assume (*h*, *g*) ∈ *set* (*apply-ocrit ocrit data hs ps1 ps*)
hence (*h*, *g*) ∈ *set* (*add-pairs gs bs ps hs data*)
by (*simp add*: *add-pairs-def assms*(*5*) *Let-def ps1-def*)
with ‹(*h*, *g*) ∉ *set* (*add-pairs gs bs ps hs data*)› **show** *False* **..**
qed
qed
qed

end

abbreviation *add-pairs-canon* ≡
add-pairs (*new-pairs-sorted canon-pair-order*) *component-crit chain-ncrit chain-ocrit canon-pair-comb*

lemma *ap-spec-add-pairs-canon*: *ap-spec add-pairs-canon*
using *np-spec-new-pairs-sorted icrit-spec-component-crit ncrit-spec-chain-ncrit ocrit-spec-chain-ocrit set-merge-wrt*
by (*rule ap-spec-add-pairs*)

6.4 Suitable Instances of the *completion* Parameter

definition *rcp-spec* :: (′*t*, ′*b*::*field*, ′*c*, ′*d*) *complT* ⇒ *bool*
where *rcp-spec rcp* ⟷
(∀ *gs bs ps sps data*.
0 ∉ *fst* ‘ *set* (*fst* (*rcp gs bs ps sps data*)) ∧
(∀ *h b*. *h* ∈ *set* (*fst* (*rcp gs bs ps sps data*)) ⟶ *b* ∈ *set gs* ∪ *set bs* ⟶ *fst b* ≠ *0* ⟶
¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)) ∧
(∀ *d*. *dickson-grading d* ⟶
dgrad-p-set-le d (*fst* ‘ *set* (*fst* (*rcp gs bs ps sps data*))) (*args-to-set* (*gs*, *bs*, *sps*))) ∧
component-of-term ‘ *Keys* (*fst* ‘ (*set* (*fst* (*rcp gs bs ps sps data*)))) ⊆
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *sps*)) ∧
(*is-Groebner-basis* (*fst* ‘ *set gs*) ⟶ *unique-idx* (*gs* @ *bs*) *data* ⟶
(*fst* ‘ *set* (*fst* (*rcp gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) ∧

(∀ (p, q)∈set sps. set sps ⊆ set bs × (set gs ∪ set bs) ⟶
(red (fst ‘ (set gs ∪ set bs) ∪ fst ‘ set (fst (rcp gs bs ps sps data))))** (spoly (fst p) (fst q)) 0))))

Informally, *rcp-spec rcp* expresses that, for suitable *gs*, *bs* and *sps*, the value of *rcp gs bs ps sps*

- is a list consisting exclusively of non-zero polynomials contained in the module generated by *set bs* ∪ *set gs*, whose leading terms are not divisible by the leading term of any non-zero *b* ∈ *set bs*, and
- contains sufficiently many new polynomials such that all S-polynomials originating from *sps* can be reduced to *0* modulo the enlarged list of polynomials.

lemma *rcp-specI*:
assumes ⋀gs bs ps sps data. 0 ∉ fst ‘ set (fst (rcp gs bs ps sps data))
assumes ⋀gs bs ps sps h b data. h ∈ set (fst (rcp gs bs ps sps data)) ⟹ b ∈ set gs ∪ set bs ⟹ fst b ≠ 0 ⟹
¬ lt (fst b) $adds_t$ lt (fst h)
assumes ⋀gs bs ps sps d data. dickson-grading d ⟹
dgrad-p-set-le d (fst ‘ set (fst (rcp gs bs ps sps data))) (args-to-set (gs, bs, sps))
assumes ⋀gs bs ps sps data. component-of-term ‘ Keys (fst ‘ (set (fst (rcp gs bs ps sps data)))) ⊆
component-of-term ‘ Keys (args-to-set (gs, bs, sps))
assumes ⋀gs bs ps sps data. is-Groebner-basis (fst ‘ set gs) ⟹ unique-idx (gs @ bs) data ⟹
(fst ‘ set (fst (rcp gs bs ps sps data)) ⊆ pmdl (args-to-set (gs, bs, sps)) ∧
(∀ (p, q)∈set sps. set sps ⊆ set bs × (set gs ∪ set bs) ⟶
(red (fst ‘ (set gs ∪ set bs) ∪ fst ‘ set (fst (rcp gs bs ps sps data))))** (spoly (fst p) (fst q)) 0))
shows *rcp-spec rcp*
unfolding *rcp-spec-def* **using** *assms* **by** *auto*

lemma *rcp-specD1*:
assumes *rcp-spec rcp*
shows 0 ∉ fst ‘ set (fst (rcp gs bs ps sps data))
using *assms* **unfolding** *rcp-spec-def* **by** (*elim allE conjE*)

lemma *rcp-specD2*:
assumes *rcp-spec rcp*
and h ∈ set (fst (rcp gs bs ps sps data)) **and** b ∈ set gs ∪ set bs **and** fst b ≠ 0
shows ¬ lt (fst b) $adds_t$ lt (fst h)
using *assms* **unfolding** *rcp-spec-def* **by** (*elim allE conjE*, *blast*)

lemma *rcp-specD3*:
assumes *rcp-spec rcp* **and** *dickson-grading d*

shows *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*rcp gs bs ps sps data*))) (*args-to-set* (*gs*, *bs*, *sps*))
using *assms* **unfolding** *rcp-spec-def* **by** (*elim allE conjE*, *blast*)

lemma *rcp-specD4*:
assumes *rcp-spec rcp*
shows *component-of-term* ' *Keys* (*fst* ' (*set* (*fst* (*rcp gs bs ps sps data*)))) ⊆ *component-of-term* ' *Keys* (*args-to-set* (*gs*, *bs*, *sps*))
using *assms* **unfolding** *rcp-spec-def* **by** (*elim allE conjE*)

lemma *rcp-specD5*:
assumes *rcp-spec rcp* **and** *is-Groebner-basis* (*fst* ' *set gs*) **and** *unique-idx* (*gs* @ *bs*) *data*
shows *fst* ' *set* (*fst* (*rcp gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*))
using *assms* **unfolding** *rcp-spec-def* **by** *blast*

lemma *rcp-specD6*:
assumes *rcp-spec rcp* **and** *is-Groebner-basis* (*fst* ' *set gs*) **and** *unique-idx* (*gs* @ *bs*) *data*
and *set sps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
and (*p*, *q*) ∈ *set sps*
shows (*red* (*fst* ' (*set gs* ∪ *set bs*) ∪ *fst* ' *set* (*fst* (*rcp gs bs ps sps data*))))** (*spoly* (*fst p*) (*fst q*)) *0*
using *assms* **unfolding** *rcp-spec-def* **by** *blast*

lemma *compl-struct-rcp*:
assumes *rcp-spec rcp*
shows *compl-struct rcp*
proof (*rule compl-structI*)
fix *d*::'*a* ⇒ *nat* **and** *gs bs ps* **and** *sps*::('*t*, '*b*, '*c*) *pdata-pair list* **and** *data*::*nat* × '*d*
assume *dickson-grading d* **and** *set sps* ⊆ *set ps*
from *assms this*(*1*) **have** *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*)))
(*args-to-set* (*gs*, *bs*, *sps*))
by (*rule rcp-specD3*)
also have *dgrad-p-set-le d* ... (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule dgrad-p-set-le-subset*, *rule args-to-set-subset3*, *fact* ‹*set sps* ⊆ *set ps*›)
finally show *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*)))
(*args-to-set* (*gs*, *bs*, *ps*)) **.**
next
fix *gs bs ps* **and** *sps*::('*t*, '*b*, '*c*) *pdata-pair list* **and** *data*::*nat* × '*d*
from *assms* **show** *0* ∉ *fst* ' *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*))
by (*rule rcp-specD1*)
next
fix *gs bs ps sps h b data*
assume *h* ∈ *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*))
and *b* ∈ *set gs* ∪ *set bs* **and** *fst b* ≠ *0*
with *assms* **show** ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **by** (*rule rcp-specD2*)

next
fix *gs bs ps* **and** *sps*::('*t*, '*b*, '*c*) *pdata-pair list* **and** *data*::*nat* × '*d*
assume *set sps* ⊆ *set ps*
from *assms*
have *component-of-term* ' *Keys* (*fst* ' *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*))) ⊆
component-of-term ' *Keys* (*args-to-set* (*gs*, *bs*, *sps*))
by (*rule rcp-specD4*)
also have ... ⊆ *component-of-term* ' *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *rule args-to-set-subset3*, *fact* ‹*set sps* ⊆ *set ps*›)
finally show *component-of-term* ' *Keys* (*fst* ' *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*))) ⊆
component-of-term ' *Keys* (*args-to-set* (*gs*, *bs*, *ps*)) **.**
qed

lemma *compl-pmdl-rcp*:
assumes *rcp-spec rcp*
shows *compl-pmdl rcp*
proof (*rule compl-pmdlI*)
fix *gs bs* :: ('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* :: ('*t*, '*b*, '*c*) *pdata-pair list* **and** *data*::*nat* × '*d*
assume *gb*: *is-Groebner-basis* (*fst* ' *set gs*) **and** *set sps* ⊆ *set ps*
and *un*: *unique-idx* (*gs* @ *bs*) *data*
let *?res* = *fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*)
from *assms gb un* **have** *fst* ' *set ?res* ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*))
by (*rule rcp-specD5*)
also have ... ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*))
by (*rule pmdl.span-mono*, *rule args-to-set-subset3*, *fact* ‹*set sps* ⊆ *set ps*›)
finally show *fst* ' *set ?res* ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *ps*)) **.**
qed

lemma *compl-conn-rcp*:
assumes *rcp-spec rcp*
shows *compl-conn rcp*
proof (*rule compl-connI*)
fix *d*::'*a* ⇒ *nat* **and** *m gs bs ps sps p* **and** *q*::('*t*, '*b*, '*c*) *pdata* **and** *data*::*nat* × '*d*
assume *dg*: *dickson-grading d* **and** *gs-sub*: *fst* ' *set gs* ⊆ *dgrad-p-set d m*
and *gb*: *is-Groebner-basis* (*fst* ' *set gs*) **and** *bs-sub*: *fst* ' *set bs* ⊆ *dgrad-p-set d m*
and *ps-sub*: *set ps* ⊆ *set bs* × (*set gs* ∪ *set bs*) **and** *set sps* ⊆ *set ps*
and *uid*: *unique-idx* (*gs* @ *bs*) *data*
and (*p*, *q*) ∈ *set sps* **and** *fst p* ≠ *0* **and** *fst q* ≠ *0*

from ‹*set sps* ⊆ *set ps*› *ps-sub* **have** *sps-sub*: *set sps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
by (*rule subset-trans*)

let *?res* = *fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*)
have *fst* ' *set ?res* ⊆ *dgrad-p-set d m*

proof (*rule dgrad-p-set-le-dgrad-p-set*, *rule rcp-specD3*, *fact*+)
show *args-to-set* (*gs*, *bs*, *sps*) ⊆ *dgrad-p-set d m*
by (*simp add*: *args-to-set-subset-Times*[*OF sps-sub*], *rule*, *fact*+)
qed
moreover have *gs-bs-sub*: *fst* ‘ (*set gs* ∪ *set bs*) ⊆ *dgrad-p-set d m* **by** (*simp add*: *image-Un*, *rule*, *fact*+)
ultimately have *res-sub*: *fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set ?res* ⊆ *dgrad-p-set d m* **by** *simp*

from ‹(*p*, *q*) ∈ *set sps*› ‹*set sps* ⊆ *set ps*› *ps-sub*
have *fst p* ∈ *fst* ‘ *set bs* **and** *fst q* ∈ *fst* ‘ (*set gs* ∪ *set bs*) **by** *auto*
with ‹*fst* ‘ *set bs* ⊆ *dgrad-p-set d m*› *gs-bs-sub*
have *fst p* ∈ *dgrad-p-set d m* **and** *fst q* ∈ *dgrad-p-set d m* **by** *auto*

with *dg res-sub* **show** *crit-pair-cbelow-on d m* (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set ?res*) (*fst p*) (*fst q*)
using ‹*fst p* ≠ *0*› ‹*fst q* ≠ *0*›
proof (*rule spoly-red-zero-imp-crit-pair-cbelow-on*)
from *assms gb uid sps-sub* ‹(*p*, *q*) ∈ *set sps*›
show (*red* (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*rcp gs bs* (*ps* −− *sps*) *sps data*))))**
(*spoly* (*fst p*) (*fst q*)) *0*
by (*rule rcp-specD6*)
qed
qed

end

6.5 Suitable Instances of the *add-basis* Parameter

definition *add-basis-naive* :: ('*a*, '*b*, '*c*, '*d*) *abT*
where *add-basis-naive gs bs ns data* = *bs* @ *ns*

lemma *ab-spec-add-basis-naive*: *ab-spec add-basis-naive*
by (*rule ab-specI*, *simp-all add*: *add-basis-naive-def*)

definition *add-basis-sorted* :: (*nat* × '*d* ⇒ ('*a*, '*b*, '*c*) *pdata* ⇒ ('*a*, '*b*, '*c*) *pdata* ⇒ *bool*) ⇒ ('*a*, '*b*, '*c*, '*d*) *abT*
where *add-basis-sorted rel gs bs ns data* = *merge-wrt* (*rel data*) *bs ns*

lemma *ab-spec-add-basis-sorted*: *ab-spec* (*add-basis-sorted rel*)
by (*rule ab-specI*, *simp-all add*: *add-basis-sorted-def set-merge-wrt*)

definition *card-keys* :: ('*a* $\Rightarrow_0$ '*b*::*zero*) ⇒ *nat*
where *card-keys* = *card* ∘ *keys*

definition (**in** *ordered-term*) *canon-basis-order* :: '*d* ⇒ ('*t*, '*b*::*zero*, '*c*) *pdata* ⇒ ('*t*, '*b*, '*c*) *pdata* ⇒ *bool*
where *canon-basis-order data p q* ⟷

(*let cp* = *card-keys* (*fst p*); *cq* = *card-keys* (*fst q*) *in*
cp < *cq* ∨ (*cp* = *cq* ∧ *lt* (*fst p*) $\prec_t$ *lt* (*fst q*)))

abbreviation (**in** *ordered-term*) *add-basis-canon* ≡ *add-basis-sorted canon-basis-order*

6.6 Special Case: Scalar Polynomials

context *gd-powerprod*
begin

lemma *remdups-map-component-of-term-punit*:
remdups (*map* (λ-. ()) (*punit.Keys-to-list* (*map fst bs*))) =
(*if* (∀ *b*∈*set bs*. *fst b* = *0*) *then* [] *else* [()])
proof (*split if-split*, *intro conjI impI*)
assume ∀ *b*∈*set bs*. *fst b* = *0*
hence *fst* ‘ *set bs* ⊆ {*0*} **by** *blast*
hence *Keys* (*fst* ‘ *set bs*) = {} **by** (*metis Keys-empty Keys-zero subset-singleton-iff*)
hence *punit.Keys-to-list* (*map fst bs*) = []
by (*simp add*: *set-empty*[*symmetric*] *punit.set-Keys-to-list del*: *set-empty*)
thus *remdups* (*map* (λ-. ()) (*punit.Keys-to-list* (*map fst bs*))) = [] **by** *simp*
next
assume ¬ (∀ *b*∈*set bs*. *fst b* = *0*)
hence ∃ *b*∈*set bs*. *fst b* ≠ *0* **by** *simp*
then obtain *b* **where** *b* ∈ *set bs* **and** *fst b* ≠ *0* **..**
hence *Keys* (*fst* ‘ *set bs*) ≠ {} **by** (*meson Keys-not-empty* ‹*fst b* ≠ *0*› *imageI*)
hence *set* (*punit.Keys-to-list* (*map fst bs*)) ≠ {} **by** (*simp add*: *punit.set-Keys-to-list*)
hence *punit.Keys-to-list* (*map fst bs*) ≠ [] **by** *simp*
thus *remdups* (*map* (λ-. ()) (*punit.Keys-to-list* (*map fst bs*))) = [()]
by (*metis* (*full-types*) *remdups-adj.cases old.unit.exhaust Nil-is-map-conv* ‹*punit.Keys-to-list* (*map fst bs*) ≠ []› *distinct-length-2-or-more distinct-remdups remdups-eq-nil-right-iff*)
qed

lemma *count-const-lt-components-punit* [*code*]:
punit.count-const-lt-components hs =
(*if* (∃ *h*∈*set hs*. *punit.const-lt-component* (*fst h*) = *Some* ()) *then 1 else 0*)
proof (*simp add*: *punit.count-const-lt-components-def cong del*: *image-cong-simp*,
simp add: *card-set* [*symmetric*] *cong del*: *image-cong-simp*, *rule*)
assume ∃ *h*∈*set hs*. *punit.const-lt-component* (*fst h*) = *Some* ()
then obtain *h* **where** *h* ∈ *set hs* **and** *punit.const-lt-component* (*fst h*) = *Some* () **..**
from *this*(*2*) **have** (*punit.const-lt-component* ∘ *fst*) *h* = *Some* () **by** *simp*
with ‹*h* ∈ *set hs*› **have** *Some* () ∈ (*punit.const-lt-component* ∘ *fst*) ‘ *set hs*
by (*metis rev-image-eqI*)
hence {*x*. *x* = *Some* () ∧ *x* ∈ (*punit.const-lt-component* ∘ *fst*) ‘ *set hs*} = {*Some* ()} **by** *auto*
thus *card* {*x*. *x* = *Some* () ∧ *x* ∈ (*punit.const-lt-component* ∘ *fst*) ‘ *set hs*} = *Suc 0* **by** *simp*
qed

lemma *count-rem-components-punit* [*code*]:
punit.count-rem-components bs =
(*if* (∀ *b*∈*set bs. fst b* = *0*) *then 0*
else
if (∃ *b*∈*set bs. fst b* ≠ *0* ∧ *punit.const-lt-component* (*fst b*) = *Some* ()) *then*
0 else 1)
proof (*cases* ∀ *b*∈*set bs. fst b* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *punit.count-rem-components-def remdups-map-component-of-term-punit*)
next
case *False*
have *eq*: (∃ *b*∈*set* [*b*←*bs* . *fst b* ≠ *0*]. *punit.const-lt-component* (*fst b*) = *Some* ())
=
(∃ *b*∈*set bs. fst b* ≠ *0* ∧ *punit.const-lt-component* (*fst b*) = *Some* ())
by (*metis* (*mono-tags*, *lifting*) *filter-set member-filter*)
show *?thesis*
by (*simp only*: *False punit.count-rem-components-def eq if-False*
remdups-map-component-of-term-punit count-const-lt-components-punit punit-component-of-term,
simp)
qed

lemma *full-gb-punit* [*code*]:
punit.full-gb bs = (*if* (∀ *b*∈*set bs. fst b* = *0*) *then* [] *else* [(*1*, *0*, *default*)])
by (*simp add*: *punit.full-gb-def remdups-map-component-of-term-punit*)

abbreviation *add-pairs-punit-canon* ≡
punit.add-pairs (*punit.new-pairs-sorted punit.canon-pair-order*) *punit.product-crit*
punit.chain-ncrit
punit.chain-ocrit punit.canon-pair-comb

lemma *ap-spec-add-pairs-punit-canon*: *punit.ap-spec add-pairs-punit-canon*
using *punit.np-spec-new-pairs-sorted punit.icrit-spec-product-crit punit.ncrit-spec-chain-ncrit*
punit.ocrit-spec-chain-ocrit set-merge-wrt
by (*rule punit.ap-spec-add-pairs*)

end

end

7 Buchberger's Algorithm

theory *Buchberger*
imports *Algorithm-Schema*
begin

context *gd-term*
begin

7.1 Reduction

definition *trdsp*::($'t \Rightarrow_0 'b$) *list* ⇒ ($'t$, $'b$, $'c$) *pdata-pair* ⇒ ($'t \Rightarrow_0 'b$::*field*)
  where *trdsp bs p* ≡ *trd bs* (*spoly* (*fst* (*fst p*)) (*fst* (*snd p*)))

lemma *trdsp-alt*: *trdsp bs* (*p*, *q*) = *trd bs* (*spoly* (*fst p*) (*fst q*))
  by (*simp add*: *trdsp-def*)

lemma *trdsp-in-pmdl*: *trdsp bs* (*p*, *q*) ∈ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*)))
  unfolding *trdsp-alt*
proof (*rule pmdl-closed-trd*)
  have *spoly* (*fst p*) (*fst q*) ∈ *pmdl* {*fst p*, *fst q*}
  proof (*rule pmdl-closed-spoly*)
    show *fst p* ∈ *pmdl* {*fst p*, *fst q*} **by** (*rule pmdl.span-base*, *simp*)
  next
    show *fst q* ∈ *pmdl* {*fst p*, *fst q*} **by** (*rule pmdl.span-base*, *simp*)
  qed
  also have ... ⊆ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*)))
    by (*rule pmdl.span-mono*, *simp*)
  finally show *spoly* (*fst p*) (*fst q*) ∈ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*)))
.
next
  have *set bs* ⊆ *insert* (*fst p*) (*insert* (*fst q*) (*set bs*)) **by** *blast*
  also have ... ⊆ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*)))
    by (*fact pmdl.span-superset*)
  finally show *set bs* ⊆ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*))) **.**
qed

lemma *dgrad-p-set-le-trdsp*:
  assumes *dickson-grading d*
  shows *dgrad-p-set-le d* {*trdsp bs* (*p*, *q*)} (*insert* (*fst p*) (*insert* (*fst q*) (*set bs*)))
proof −
  let *?h* = *trdsp bs* (*p*, *q*)
  have (*red* (*set bs*))** (*spoly* (*fst p*) (*fst q*)) *?h* **unfolding** *trdsp-alt* **by** (*rule trd-red-rtrancl*)
  with *assms* **have** *dgrad-p-set-le d* {*?h*} (*insert* (*spoly* (*fst p*) (*fst q*)) (*set bs*))
    by (*rule dgrad-p-set-le-red-rtrancl*)
  also have *dgrad-p-set-le d* ... ({*fst p*, *fst q*} ∪ *set bs*)
  proof (*rule dgrad-p-set-leI-insert*)
    show *dgrad-p-set-le d* (*set bs*) ({*fst p*, *fst q*} ∪ *set bs*) **by** (*rule dgrad-p-set-le-subset*, *blast*)
  next
    from *assms* **have** *dgrad-p-set-le d* {*spoly* (*fst p*) (*fst q*)} {*fst p*, *fst q*}
      by (*rule dgrad-p-set-le-spoly*)
    also have *dgrad-p-set-le d* ... ({*fst p*, *fst q*} ∪ *set bs*)
      by (*rule dgrad-p-set-le-subset*, *blast*)
    finally show *dgrad-p-set-le d* {*spoly* (*fst p*) (*fst q*)} ({*fst p*, *fst q*} ∪ *set bs*) **.**
  qed
  finally show *?thesis* **by** *simp*

qed

lemma *components-trdsp-subset*:
component-of-term ' keys (trdsp bs (p, q)) ⊆ *component-of-term ' Keys (insert (fst p) (insert (fst q) (set bs)))*
proof −
let *?h* = *trdsp bs (p, q)*
have *(red (set bs))*** *(spoly (fst p) (fst q)) ?h* **unfolding** *trdsp-alt* **by** *(rule trd-red-rtrancl)*
hence *component-of-term ' keys ?h* ⊆
component-of-term ' keys (spoly (fst p) (fst q)) ∪ *component-of-term ' Keys (set bs)*
by *(rule components-red-rtrancl-subset)*
also have ... ⊆ *component-of-term ' Keys {fst p, fst q}* ∪ *component-of-term ' Keys (set bs)*
using *components-spoly-subset* **by** *force*
also have ... = *component-of-term ' Keys (insert (fst p) (insert (fst q) (set bs)))*
by *(simp add: Keys-insert image-Un Un-assoc)*
finally show *?thesis* **.**
qed

definition *gb-red-aux* :: *('t, 'b::field, 'c) pdata list* ⇒ *('t, 'b, 'c) pdata-pair list* ⇒
('t $\Rightarrow_0$ *'b) list*
where *gb-red-aux bs ps* =
(let bs' = map fst bs in
filter (λh. h ≠ 0) (map (trdsp bs') ps)
)

Actually, *gb-red-aux* is only called on singleton lists.

lemma *set-gb-red-aux*: *set (gb-red-aux bs ps) = (trdsp (map fst bs)) ' set ps − {0}*
by *(simp add: gb-red-aux-def, blast)*

lemma *in-set-gb-red-auxI*:
assumes *(p, q) ∈ set ps* **and** *h = trdsp (map fst bs) (p, q)* **and** *h ≠ 0*
shows *h ∈ set (gb-red-aux bs ps)*
using *assms(1, 3)* **unfolding** *set-gb-red-aux assms(2)* **by** *force*

lemma *in-set-gb-red-auxE*:
assumes *h ∈ set (gb-red-aux bs ps)*
obtains *p q* **where** *(p, q) ∈ set ps* **and** *h = trdsp (map fst bs) (p, q)*
using *assms* **unfolding** *set-gb-red-aux* **by** *force*

lemma *gb-red-aux-not-zero*: *0 ∉ set (gb-red-aux bs ps)*
by *(simp add: set-gb-red-aux)*

lemma *gb-red-aux-irredudible*:
assumes *h ∈ set (gb-red-aux bs ps)* **and** *b ∈ set bs* **and** *fst b ≠ 0*
shows ¬ *lt (fst b)* $adds_t$ *lt h*
proof

assume *lt* (*fst b*) $adds_t$ (*lt h*)
from *assms*(*1*) **obtain** *p q* :: (′*t*, ′*b*, ′*c*) *pdata* **where** *h*: *h* = *trdsp* (*map fst bs*) (*p*, *q*)
by (*rule in-set-gb-red-auxE*)
have ¬ *is-red* (*set* (*map fst bs*)) *h* **unfolding** *h trdsp-def* **by** (*rule trd-irred*)
moreover have *is-red* (*set* (*map fst bs*)) *h*
proof (*rule is-red-addsI*)
from *assms*(*2*) **show** *fst b* ∈ *set* (*map fst bs*) **by** (*simp*)
next
from *assms*(*1*) **have** *h* ≠ *0* **by** (*simp add*: *set-gb-red-aux*)
thus *lt h* ∈ *keys h* **by** (*rule lt-in-keys*)
qed *fact*+
ultimately show *False* **..**
qed

lemma *gb-red-aux-dgrad-p-set-le*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* (*set* (*gb-red-aux bs ps*)) (*args-to-set* ([], *bs*, *ps*))
proof (*rule dgrad-p-set-leI*)
fix *h*
assume *h* ∈ *set* (*gb-red-aux bs ps*)
then obtain *p q* **where** (*p*, *q*) ∈ *set ps* **and** *h*: *h* = *trdsp* (*map fst bs*) (*p*, *q*)
by (*rule in-set-gb-red-auxE*)
from *assms* **have** *dgrad-p-set-le d* {*h*} (*insert* (*fst p*) (*insert* (*fst q*) (*set* (*map fst bs*))))
unfolding *h* **by** (*rule dgrad-p-set-le-trdsp*)
also have *dgrad-p-set-le d* ... (*args-to-set* ([], *bs*, *ps*))
proof (*rule dgrad-p-set-le-subset*, *intro insert-subsetI*)
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst p* ∈ *fst* ‘ *fst* ‘ *set ps* **by** *force*
thus *fst p* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst q* ∈ *fst* ‘ *snd* ‘ *set ps* **by** *force*
thus *fst q* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
show *set* (*map fst bs*) ⊆ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
qed
finally show *dgrad-p-set-le d* {*h*} (*args-to-set* ([], *bs*, *ps*)) **.**
qed

lemma *components-gb-red-aux-subset*:
component-of-term ‘ *Keys* (*set* (*gb-red-aux bs ps*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
proof
fix *k*
assume *k* ∈ *component-of-term* ‘ *Keys* (*set* (*gb-red-aux bs ps*))
then obtain *v* **where** *v* ∈ *Keys* (*set* (*gb-red-aux bs ps*)) **and** *k*: *k* = *component-of-term v* **..**
from *this*(*1*) **obtain** *h* **where** *h* ∈ *set* (*gb-red-aux bs ps*) **and** *v* ∈ *keys h* **by** (*rule in-KeysE*)

from *this*(*1*) **obtain** *p q* **where** (*p*, *q*) ∈ *set ps* **and** *h*: *h* = *trdsp* (*map fst bs*) (*p*, *q*)
by (*rule in-set-gb-red-auxE*)
from ‹*v* ∈ *keys h*› **have** *k* ∈ *component-of-term* ‘ *keys h* **by** (*simp add*: *k*)
have *component-of-term* ‘ *keys h* ⊆ *component-of-term* ‘ *Keys* (*insert* (*fst p*) (*insert* (*fst q*) (*set* (*map fst bs*))))
unfolding *h* **by** (*rule components-trdsp-subset*)
also have ... ⊆ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
proof (*rule image-mono*, *rule Keys-mono*, *intro insert-subsetI*)
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst p* ∈ *fst* ‘ *fst* ‘ *set ps* **by** *force*
thus *fst p* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst q* ∈ *fst* ‘ *snd* ‘ *set ps* **by** *force*
thus *fst q* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
show *set* (*map fst bs*) ⊆ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
qed
finally have *component-of-term* ‘ *keys h* ⊆ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*)) **.**
with ‹*k* ∈ *component-of-term* ‘ *keys h*› **show** *k* ∈ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*)) **..**
qed

lemma *pmdl-gb-red-aux*: *set* (*gb-red-aux bs ps*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
proof
fix *h*
assume *h* ∈ *set* (*gb-red-aux bs ps*)
then obtain *p q* **where** (*p*, *q*) ∈ *set ps* **and** *h*: *h* = *trdsp* (*map fst bs*) (*p*, *q*)
by (*rule in-set-gb-red-auxE*)
have *h* ∈ *pmdl* (*insert* (*fst p*) (*insert* (*fst q*) (*set* (*map fst bs*)))) **unfolding** *h*
by (*fact trdsp-in-pmdl*)
also have ... ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
proof (*rule pmdl.span-mono*, *intro insert-subsetI*)
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst p* ∈ *fst* ‘ *fst* ‘ *set ps* **by** *force*
thus *fst p* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
from ‹(*p*, *q*) ∈ *set ps*› **have** *fst q* ∈ *fst* ‘ *snd* ‘ *set ps* **by** *force*
thus *fst q* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
next
show *set* (*map fst bs*) ⊆ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
qed
finally show *h* ∈ *pmdl* (*args-to-set* ([], *bs*, *ps*)) **.**
qed

lemma *gb-red-aux-spoly-reducible*:
assumes (*p*, *q*) ∈ *set ps*
shows (*red* (*fst* ‘ *set bs* ∪ *set* (*gb-red-aux bs ps*)))** (*spoly* (*fst p*) (*fst q*)) *0*
proof −
define *h* **where** *h* = *trdsp* (*map fst bs*) (*p*, *q*)

from *trd-red-rtrancl*[*of map fst bs spoly* (*fst p*) (*fst q*)]
have (*red* (*set* (*map fst bs*)))** (*spoly* (*fst p*) (*fst q*)) *h*
by (*simp only*: *h-def trdsp-alt*)
hence (*red* (*fst* ' *set bs* ∪ *set* (*gb-red-aux bs ps*)))** (*spoly* (*fst p*) (*fst q*)) *h*
proof (*rule red-rtrancl-subset*)
show *set* (*map fst bs*) ⊆ *fst* ' *set bs* ∪ *set* (*gb-red-aux bs ps*) **by** *simp*
qed
moreover have (*red* (*fst* ' *set bs* ∪ *set* (*gb-red-aux bs ps*)))** *h 0*
proof (*cases h* = *0*)
case *True*
show *?thesis* **unfolding** *True* **..**
next
case *False*
hence *red* {*h*} *h 0* **by** (*rule red-self*)
hence *red* (*fst* ' *set bs* ∪ *set* (*gb-red-aux bs ps*)) *h 0*
proof (*rule red-subset*)
from *assms h-def False* **have** *h* ∈ *set* (*gb-red-aux bs ps*) **by** (*rule in-set-gb-red-auxI*)
thus {*h*} ⊆ *fst* ' *set bs* ∪ *set* (*gb-red-aux bs ps*) **by** *simp*
qed
thus *?thesis* **..**
qed
ultimately show *?thesis* **by** *simp*
qed

definition *gb-red* :: ('*t*, '*b*::*field*, '*c*::*default*, '*d*) *complT*
where *gb-red gs bs ps sps data* = (*map* (λ*h*. (*h*, *default*)) (*gb-red-aux* (*gs* @ *bs*) *sps*), *snd data*)

lemma *fst-set-fst-gb-red*: *fst* ' *set* (*fst* (*gb-red gs bs ps sps data*)) = *set* (*gb-red-aux* (*gs* @ *bs*) *sps*)
by (*simp add*: *gb-red-def*, *force*)

lemma *rcp-spec-gb-red*: *rcp-spec gb-red*
proof (*rule rcp-specI*)
fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
from *gb-red-aux-not-zero* **show** *0* ∉ *fst* ' *set* (*fst* (*gb-red gs bs ps sps data*))
unfolding *fst-set-fst-gb-red* **.**
next
fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps h b* **and** *data*::*nat* × '*d*
assume *h* ∈ *set* (*fst* (*gb-red gs bs ps sps data*)) **and** *b* ∈ *set gs* ∪ *set bs*
from *this*(*1*) **have** *fst h* ∈ *fst* ' *set* (*fst* (*gb-red gs bs ps sps data*)) **by** *simp*
hence *fst h* ∈ *set* (*gb-red-aux* (*gs* @ *bs*) *sps*) **by** (*simp only*: *fst-set-fst-gb-red*)
moreover from ‹*b* ∈ *set gs* ∪ *set bs*› **have** *b* ∈ *set* (*gs* @ *bs*) **by** *simp*
moreover assume *fst b* ≠ *0*
ultimately show ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **by** (*rule gb-red-aux-irredudible*)
next
fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *d*::'*a* ⇒ *nat* **and** *data*::*nat* × '*d*
assume *dickson-grading d*
hence *dgrad-p-set-le d* (*set* (*gb-red-aux* (*gs* @ *bs*) *sps*)) (*args-to-set* ([], *gs* @ *bs*,

sps))
  by (*rule gb-red-aux-dgrad-p-set-le*)
 also have ... = *args-to-set* (*gs*, *bs*, *sps*) **by** (*simp add*: *args-to-set-alt image-Un*)
 finally show *dgrad-p-set-le d* (*fst* ‘ *set* (*fst* (*gb-red gs bs ps sps data*))) (*args-to-set* (*gs*, *bs*, *sps*))
  by (*simp only*: *fst-set-fst-gb-red*)
next
 fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
 have *component-of-term* ‘ *Keys* (*set* (*gb-red-aux* (*gs* @ *bs*) *sps*)) ⊆
   component-of-term ‘ *Keys* (*args-to-set* ([], *gs* @ *bs*, *sps*))
  by (*rule components-gb-red-aux-subset*)
 also have ... = *component-of-term* ‘ *Keys* (*args-to-set* (*gs*, *bs*, *sps*))
  by (*simp add*: *args-to-set-alt image-Un*)
 finally show *component-of-term* ‘ *Keys* (*fst* ‘ *set* (*fst* (*gb-red gs bs ps sps data*))) ⊆
   component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *sps*)) **by** (*simp only*: *fst-set-fst-gb-red*)
next
 fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
 have *set* (*gb-red-aux* (*gs* @ *bs*) *sps*) ⊆ *pmdl* (*args-to-set* ([], *gs* @ *bs*, *sps*))
  by (*fact pmdl-gb-red-aux*)
 also have ... = *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) **by** (*simp add*: *args-to-set-alt image-Un*)
 finally have *fst* ‘ *set* (*fst* (*gb-red gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*))
  by (*simp only*: *fst-set-fst-gb-red*)
 moreover {
  fix *p q* :: ('*t*, '*b*, '*c*) *pdata*
  assume (*p*, *q*) ∈ *set sps*
  hence (*red* (*fst* ‘ *set* (*gs* @ *bs*) ∪ *set* (*gb-red-aux* (*gs* @ *bs*) *sps*)))** (*spoly* (*fst p*) (*fst q*)) *0*
   by (*rule gb-red-aux-spoly-reducible*)
 }
 ultimately show
  fst ‘ *set* (*fst* (*gb-red gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) ∧
   (∀ (*p*, *q*)∈*set sps*.
    set sps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟶
     (*red* (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*gb-red gs bs ps sps data*))))** (*spoly* (*fst p*) (*fst q*)) *0*)
  by (*auto simp add*: *image-Un fst-set-fst-gb-red*)
qed

lemmas *compl-struct-gb-red* = *compl-struct-rcp*[*OF rcp-spec-gb-red*]
lemmas *compl-pmdl-gb-red* = *compl-pmdl-rcp*[*OF rcp-spec-gb-red*]
lemmas *compl-conn-gb-red* = *compl-conn-rcp*[*OF rcp-spec-gb-red*]

7.2 Pair Selection

primrec *gb-sel* :: ('*t*, '*b*::*zero*, '*c*, '*d*) *selT* **where**

gb-sel gs bs [] *data* = []|
gb-sel gs bs (*p* # *ps*) *data* = [*p*]

lemma *sel-spec-gb-sel*: *sel-spec gb-sel*
proof (*rule sel-specI*)
fix *gs bs* :: ('*t*, '*b*, '*c*) *pdata list* **and** *ps*::('*t*, '*b*, '*c*) *pdata-pair list* **and** *data*::*nat* × '*d*
assume *ps* ≠ []
then obtain *p ps*' **where** *ps*: *ps* = *p* # *ps*' **by** (*meson list.exhaust*)
show *gb-sel gs bs ps data* ≠ [] ∧ *set* (*gb-sel gs bs ps data*) ⊆ *set ps* **by** (*simp add*: *ps*)
qed

7.3 Buchberger's Algorithm

lemma *struct-spec-gb*: *struct-spec gb-sel add-pairs-canon add-basis-canon gb-red*
using *sel-spec-gb-sel ap-spec-add-pairs-canon ab-spec-add-basis-sorted compl-struct-gb-red*
by (*rule struct-specI*)

definition *gb-aux* :: ('*t*, '*b*, '*c*) *pdata list* ⇒ *nat* × *nat* × '*d* ⇒ ('*t*, '*b*, '*c*) *pdata list* ⇒
('*t*, '*b*, '*c*) *pdata-pair list* ⇒ ('*t*, '*b*::*field*, '*c*::*default*) *pdata list*
where *gb-aux* = *gb-schema-aux gb-sel add-pairs-canon add-basis-canon gb-red*

lemmas *gb-aux-simps* [*code*] = *gb-schema-aux-simps*[*OF struct-spec-gb*, *folded gb-aux-def*]

definition *gb* :: ('*t*, '*b*, '*c*) *pdata' list* ⇒ '*d* ⇒ ('*t*, '*b*::*field*, '*c*::*default*) *pdata' list*
where *gb* = *gb-schema-direct gb-sel add-pairs-canon add-basis-canon gb-red*

lemmas *gb-simps* [*code*] = *gb-schema-direct-def*[*of gb-sel add-pairs-canon add-basis-canon gb-red*, *folded gb-def gb-aux-def*]

lemmas *gb-isGB* = *gb-schema-direct-isGB*[*OF struct-spec-gb compl-conn-gb-red*, *folded gb-def*]

lemmas *gb-pmdl* = *gb-schema-direct-pmdl*[*OF struct-spec-gb compl-pmdl-gb-red*, *folded gb-def*]

7.3.1 Special Case: *punit*

lemma (**in** *gd-term*) *struct-spec-gb-punit*: *punit.struct-spec punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*
using *punit.sel-spec-gb-sel ap-spec-add-pairs-punit-canon ab-spec-add-basis-sorted punit.compl-struct-gb-red*
by (*rule punit.struct-specI*)

definition *gb-aux-punit* :: ('*a*, '*b*, '*c*) *pdata list* ⇒ *nat* × *nat* × '*d* ⇒ ('*a*, '*b*, '*c*) *pdata list* ⇒
('*a*, '*b*, '*c*) *pdata-pair list* ⇒ ('*a*, '*b*::*field*, '*c*::*default*) *pdata list*

where *gb-aux-punit* = *punit.gb-schema-aux punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*

lemmas *gb-aux-punit-simps* [*code*] = *punit.gb-schema-aux-simps*[*OF struct-spec-gb-punit*, *folded gb-aux-punit-def*]

definition *gb-punit* :: (′*a*, ′*b*, ′*c*) *pdata′ list* ⇒ ′*d* ⇒ (′*a*, ′*b*::*field*, ′*c*::*default*) *pdata′ list*
where *gb-punit* = *punit.gb-schema-direct punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*

lemmas *gb-punit-simps* [*code*] = *punit.gb-schema-direct-def*[*of punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*, *folded gb-punit-def gb-aux-punit-def*]

lemmas *gb-punit-isGB* = *punit.gb-schema-direct-isGB*[*OF struct-spec-gb-punit punit.compl-conn-gb-red*, *folded gb-punit-def*]

lemmas *gb-punit-pmdl* = *punit.gb-schema-direct-pmdl*[*OF struct-spec-gb-punit punit.compl-pmdl-gb-red*, *folded gb-punit-def*]

end

end

8 Benchmark Problems for Computing Gröbner Bases

theory *Benchmarks*
imports *Polynomials.MPoly-Type-Class-OAlist*
begin

This theory defines various well-known benchmark problems for computing Gröbner bases. The actual tests of the different algorithms on these problems are contained in the theories whose names end with *-Examples*.

8.1 Cyclic

definition *cycl-pp* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat*, *nat*) *pp*
where *cycl-pp n d i* = $sparse_0$ (*map* (λ*k*. (*modulo* (*k* + *i*) *n*, *1*)) [*0*..<*d*])

definition *cyclic* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ ′*a*::{*zero*,*one*,*uminus*}) *list*
where *cyclic to n* =
(*let xs* = [*0*..<*n*] *in*
(*map* (λ*d*. $distr_0$ *to* (*map* (λ*i*. (*cycl-pp n d i*, *1*)) *xs*)) [*1*..<*n*]) @
[$distr_0$ *to* [(*cycl-pp n n 0*, *1*), (*0*, −*1*)]]
)

cyclic n is a system of *n* polynomials in *n* indeterminates, with maximum degree *n*.

8.2 Katsura

definition *katsura-poly* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ *'a::comm-ring-1*)
  where *katsura-poly to n i* =
      change-ord to (($\sum$*j::int*=−*int n..*<*n* + *1*. *if abs* (*i* − *j*) ≤ *n then* V_0 (*nat* (*abs j*)) ∗ V_0 (*nat* (*abs* (*i* − *j*))) *else 0*) − V_0 *i*)

definition *katsura* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ *'a::comm-ring-1*) *list*
  where *katsura to n* =
    (*let xs* = [*0..*<*n*] *in*
      (*distr*$_0$ *to* ((*sparse*$_0$ [(*0*, *1*)], *1*) # (*map* (λ*i*. (*sparse*$_0$ [(*Suc i*, *1*)], *2*)) *xs*) @ [(*0*, −*1*)])) #
     (*map* (*katsura-poly to n*) *xs*)
)

For *1* ≤ *n*, *katsura n* is a system of *n* + *1* polynomials in *n* + *1* indeterminates, with maximum degree *2*.

8.3 Eco

definition *eco-poly* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ *'a::comm-ring-1*)
  where *eco-poly to m i* =
     distr$_0$ *to* ((*sparse*$_0$ [(*i*, *1*), (*m*, *1*)], *1*) # *map* (λ*j*. (*sparse*$_0$ [(*j*, *1*), (*j* + *i* + *1*, *1*), (*m*, *1*)], *1*)) [*0..*<*m* − *i* − *1*])

definition *eco* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ *'a::comm-ring-1*) *list*
  where *eco to n* =
    (*let m* = *n* − *1 in*
     (*distr*$_0$ *to* ((*map* (λ*j*. (*sparse*$_0$ [(*j*, *1*)], *1*)) [*0..*<*m*]) @ [(*0*, *1*)])) #
     (*distr*$_0$ *to* [(*sparse*$_0$ [(*m*−*1*, *1*), (*m*,*1*)], *1*), (*0*, − *of-nat m*)]) #
     (*rev* (*map* (*eco-poly to m*) [*0..*<*m*−*1*]))
)

For (*2*::*'a*) ≤ *n*, *eco n* is a system of *n* polynomials in *n* indeterminates, with maximum degree *3*.

8.4 Noon

definition *noon-poly* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* $\Rightarrow_0$ *'a::comm-ring-1*)
  where *noon-poly to n i* =

(*let ten* = *of-nat 10*; *eleven* = − *of-nat 11 in*
distr$_0$ *to* ((*map* (λ*j*. *if j* = *i then* (*sparse*$_0$ [(*i*, *1*)], *eleven*) *else* (*sparse*$_0$ [(*j*, *2*), (*i*, *1*)], *ten*)) [*0*..<*n*]) @
[(*0*, *ten*)]))

definition *noon* :: (*nat*, *nat*) *pp nat-term-order* ⇒ *nat* ⇒ ((*nat*, *nat*) *pp* ⇒$_0$ ′*a*::*comm-ring-1*) *list*
where *noon to n* = (*noon-poly to n 1*) # (*noon-poly to n 0*) # (*map* (*noon-poly to n*) [*2*..<*n*])

For (*2*::′*a*) ≤ *n*, *noon n* is a system of *n* polynomials in *n* indeterminates, with maximum degree *3*.

end

9 Code Equations Related to the Computation of Gröbner Bases

theory *Algorithm-Schema-Impl*
imports *Algorithm-Schema Benchmarks*
begin

lemma *card-keys-MP-oalist* [*code*]: *card-keys* (*MP-oalist xs*) = *length* (*fst* (*list-of-oalist-ntm xs*))
proof −
let *?rel* = *ko.lt* (*key-order-of-nat-term-order-inv* (*snd* (*list-of-oalist-ntm xs*)))
have *irreflp ?rel* **by** (*simp add*: *irreflp-def*)
moreover have *transp ?rel* **by** (*simp add*: *lt-of-nat-term-order-alt*)
ultimately have ∗: *distinct* (*map fst* (*fst* (*list-of-oalist-ntm xs*))) **using** *oa-ntm.list-of-oalist-sorted*
by (*rule distinct-sorted-wrt-irrefl*)
have *card-keys* (*MP-oalist xs*) = *length* (*map fst* (*fst* (*list-of-oalist-ntm xs*)))
by (*simp only*: *card-keys-def keys-MP-oalist image-set o-def oa-ntm.sorted-domain-def*[*symmetric*],
rule distinct-card, *fact* ∗)
also have ... = *length* (*fst* (*list-of-oalist-ntm xs*)) **by** *simp*
finally show *?thesis* **.**
qed

end

theory *Code-Target-Rat*
imports *Complex-Main HOL−Library.Code-Target-Numeral*
begin

Mapping type *rat* to type "Rat.rat" in Isabelle/ML. Serialization for other target languages will be provided in the future.

context includes *integer.lifting* **begin**

lift-definition *rat-of-integer* :: *integer* ⇒ *rat* **is** *Rat.of-int* **.**

lift-definition *quotient-of′* :: *rat* ⇒ *integer* × *integer* **is** *quotient-of* **.**

lemma [*code*]: *Rat.of-int* (*int-of-integer x*) = *rat-of-integer x*
  by *transfer simp*

lemma [*code-unfold*]: *quotient-of* = (*λx. map-prod int-of-integer int-of-integer* (*quotient-of′ x*))
  by *transfer simp*

end

code-printing
  type-constructor *rat* ⇀
    (*SML*) *Rat.rat* |
  constant *plus* :: *rat* ⇒ - ⇒ - ⇀
    (*SML*) *Rat.add* |
  constant *minus* :: *rat* ⇒ - ⇒ - ⇀
    (*SML*) *Rat.add* ((-)) (*Rat.neg* ((-))) |
  constant *times* :: *rat* ⇒ - ⇒ - ⇀
    (*SML*) *Rat.mult* |
  constant *inverse* :: *rat* ⇒ - ⇀
    (*SML*) *Rat.inv* |
  constant *divide* :: *rat* ⇒ - ⇒ - ⇀
    (*SML*) *Rat.mult* ((-)) (*Rat.inv* ((-))) |
  constant *rat-of-integer* :: *integer* ⇒ *rat* ⇀
    (*SML*) *Rat.of′-int* |
  constant *abs* :: *rat* ⇒ - ⇀
    (*SML*) *Rat.abs* |
  constant *0* :: *rat* ⇀
    (*SML*) !(*Rat.make* (*0*, *1*)) |
  constant *1* :: *rat* ⇀
    (*SML*) !(*Rat.make* (*1*, *1*)) |
  constant *uminus* :: *rat* ⇒ *rat* ⇀
    (*SML*) *Rat.neg* |
  constant *HOL.equal* :: *rat* ⇒ - ⇀
    (*SML*) !((- : *Rat.rat*) = -) |
  constant *quotient-of′* ⇀
    (*SML*) *Rat.dest*

end

10 Sample Computations with Buchberger's Algorithm

theory *Buchberger-Examples*
  imports *Buchberger Algorithm-Schema-Impl Code-Target-Rat*
begin

lemma (**in** *gd-term*) *compute-trd-aux* [*code*]:
  trd-aux fs p r =
    (*if is-zero p then*
      r
    else
      case find-adds fs (*lt p*) *of*
        None ⇒ *trd-aux fs* (*tail p*) (*plus-monomial-less r* (*lc p*) (*lt p*))
      | *Some f* ⇒ *trd-aux fs* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *r*
)
  by (*simp only*: *trd-aux.simps*[*of fs p r*] *plus-monomial-less-def is-zero-def*)

10.1 Scalar Polynomials

global-interpretation *punit'*: *gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit cmp-term*
  rewrites *punit.adds-term* = (*adds*)
  and *punit.pp-of-term* = (λ*x*. *x*)
  and *punit.component-of-term* = (λ-. ())
  and *punit.monom-mult* = *monom-mult-punit*
  and *punit.mult-scalar* = *mult-scalar-punit*
  and *punit'.punit.min-term* = *min-term-punit*
  and *punit'.punit.lt* = *lt-punit cmp-term*
  and *punit'.punit.lc* = *lc-punit cmp-term*
  and *punit'.punit.tail* = *tail-punit cmp-term*
  and *punit'.punit.ord-p* = *ord-p-punit cmp-term*
  and *punit'.punit.ord-strict-p* = *ord-strict-p-punit cmp-term*
  for *cmp-term* :: (*'a*::*nat*, *'b*::{*nat*,*add-wellorder*}) *pp nat-term-order*

  defines *find-adds-punit* = *punit'.punit.find-adds*
  and *trd-aux-punit* = *punit'.punit.trd-aux*
  and *trd-punit* = *punit'.punit.trd*
  and *spoly-punit* = *punit'.punit.spoly*
  and *count-const-lt-components-punit* = *punit'.punit.count-const-lt-components*
  and *count-rem-components-punit* = *punit'.punit.count-rem-components*
  and *const-lt-component-punit* = *punit'.punit.const-lt-component*
  and *full-gb-punit* = *punit'.punit.full-gb*
  and *add-pairs-single-sorted-punit* = *punit'.punit.add-pairs-single-sorted*
  and *add-pairs-punit* = *punit'.punit.add-pairs*
  and *canon-pair-order-aux-punit* = *punit'.punit.canon-pair-order-aux*
  and *canon-basis-order-punit* = *punit'.punit.canon-basis-order*
  and *new-pairs-sorted-punit* = *punit'.punit.new-pairs-sorted*

and *product-crit-punit* = *punit'.punit.product-crit*
and *chain-ncrit-punit* = *punit'.punit.chain-ncrit*
and *chain-ocrit-punit* = *punit'.punit.chain-ocrit*
and *apply-icrit-punit* = *punit'.punit.apply-icrit*
and *apply-ncrit-punit* = *punit'.punit.apply-ncrit*
and *apply-ocrit-punit* = *punit'.punit.apply-ocrit*
and *trdsp-punit* = *punit'.punit.trdsp*
and *gb-sel-punit* = *punit'.punit.gb-sel*
and *gb-red-aux-punit* = *punit'.punit.gb-red-aux*
and *gb-red-punit* = *punit'.punit.gb-red*
and *gb-aux-punit* = *punit'.punit.gb-aux-punit*
and *gb-punit* = *punit'.punit.gb-punit* — Faster, because incorporates product criterion.
subgoal by (*fact gd-powerprod-ord-pp-punit*)
subgoal by (*fact punit-adds-term*)
subgoal by (*simp add*: *id-def*)
subgoal by (*fact punit-component-of-term*)
subgoal by (*simp only*: *monom-mult-punit-def*)
subgoal by (*simp only*: *mult-scalar-punit-def*)
subgoal using *min-term-punit-def* **by** *fastforce*
subgoal by (*simp only*: *lt-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *lc-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *tail-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *ord-p-punit-def ord-pp-strict-punit-alt*)
subgoal by (*simp only*: *ord-strict-p-punit-def ord-pp-strict-punit-alt*)
done

lemma *compute-spoly-punit* [*code*]:
spoly-punit to p q = (*let t1* = *lt-punit to p*; *t2* = *lt-punit to q*; *l* = *lcs t1 t2 in* (*monom-mult-punit* (*1* / *lc-punit to p*) (*l* − *t1*) *p*) − (*monom-mult-punit* (*1* / *lc-punit to q*) (*l* − *t2*) *q*))
by (*simp add*: *punit'.punit.spoly-def Let-def punit'.punit.lc-def*)

lemma *compute-trd-punit* [*code*]: *trd-punit to fs p* = *trd-aux-punit to fs p* (*change-ord to 0*)
by (*simp only*: *punit'.punit.trd-def change-ord-def*)

experiment begin interpretation *trivariate*$_0$*-rat* **.**

lemma
lt-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *sparse*$_0$ [(*0*, *2*), (*2*, *3*)]
by *eval*

lemma
lc-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *1*
by *eval*

lemma
tail-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *3* ∗ X^2 ∗ *Y*

by *eval*

lemma
ord-strict-p-punit DRLEX $(X^2 * Z \hat{} 4 - 2 * Y \hat{} 3 * Z^2)$ $(X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2)$
by *eval*

lemma
trd-punit DRLEX $[Y^2 * Z + 2 * Y * Z \hat{} 3]$ $(X^2 * Z \hat{} 4 - 2 * Y \hat{} 3 * Z \hat{} 3) =$
$X^2 * Z \hat{} 4 + Y \hat{} 4 * Z$
by *eval*

lemma
spoly-punit DRLEX $(X^2 * Z \hat{} 4 - 2 * Y \hat{} 3 * Z^2)$ $(Y^2 * Z + 2 * Z \hat{} 3) =$
$-2 * Y \hat{} 3 * Z^2 - (C_0\ (1\ /\ 2)) * X^2 * Y^2 * Z^2$
by *eval*

lemma
gb-punit DRLEX
[
$(X^2 * Z \hat{} 4 - 2 * Y \hat{} 3 * Z^2, ())$,
$(Y^2 * Z + 2 * Z \hat{} 3, ())$
] () =
[
$(-2 * Y \hat{} 3 * Z^2 - (C_0\ (1\ /\ 2)) * X^2 * Y^2 * Z^2, ())$,
$(X^2 * Z \hat{} 4 - 2 * Y \hat{} 3 * Z^2, ())$,
$(Y^2 * Z + 2 * Z \hat{} 3, ())$,
$(-\ (C_0\ (1\ /\ 2)) * X^2 * Y \hat{} 4 * Z - 2 * Y \hat{} 5 * Z, ())$
]
by *eval*

lemma
gb-punit DRLEX
[
$(X^2 * Z^2 - Y, ())$,
$(Y^2 * Z - 1, ())$
] () =
[
$(-\ (Y \hat{} 3) + X^2 * Z, ())$,
$(X^2 * Z^2 - Y, ())$,
$(Y^2 * Z - 1, ())$
]
by *eval*

lemma
gb-punit DRLEX
[
$(X \hat{} 3 - X * Y * Z^2, ())$,

$(Y^2 * Z - 1, ())$
] () =
[
$(- (X \,\hat{}\, 3 * Y) + X * Z, ()),$
$(X \,\hat{}\, 3 - X * Y * Z^2, ()),$
$(Y^2 * Z - 1, ()),$
$(- (X * Z \,\hat{}\, 3) + X \,\hat{}\, 5, ())$
]
by *eval*

lemma
gb-punit DRLEX
[
$(X^2 + Y^2 + Z^2 - 1, ()),$
$(X * Y - Z - 1, ()),$
$(Y^2 + X, ()),$
$(Z^2 + X, ())$
] () =
[
(*1*, ())
]
by *eval*

end

value [*code*] *length* (*gb-punit DRLEX* (*map* (λ*p*. (*p*, ())) ((*katsura DRLEX 2*)::(- $\Rightarrow_0$ *rat*) *list*)) ())

value [*code*] *length* (*gb-punit DRLEX* (*map* (λ*p*. (*p*, ())) ((*cyclic DRLEX 5*)::(- $\Rightarrow_0$ *rat*) *list*)) ())

10.2 Vector Polynomials

We must define the following four constants outside the global interpretation, since otherwise their types are too general.

definition *splus-pprod* :: (′*a*::*nat*, ′*b*::*nat*) *pp* ⇒ -
where *splus-pprod* = *pprod.splus*

definition *monom-mult-pprod* :: ′*c*::*semiring-0* ⇒ (′*a*::*nat*, ′*b*::*nat*) *pp* ⇒ -
where *monom-mult-pprod* = *pprod.monom-mult*

definition *mult-scalar-pprod* :: ((′*a*::*nat*, ′*b*::*nat*) *pp* $\Rightarrow_0$ ′*c*::*semiring-0*) ⇒ -
where *mult-scalar-pprod* = *pprod.mult-scalar*

definition *adds-term-pprod* :: ((′*a*::*nat*, ′*b*::*nat*) *pp* × -) ⇒ -
where *adds-term-pprod* = *pprod.adds-term*

global-interpretation *pprod*′: *gd-nat-term* λ*x*::(′*a*, ′*b*) *pp* × ′*c*. *x* λ*x*. *x* *cmp-term*
rewrites *pprod.pp-of-term* = *fst*

and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.monom-mult* = *monom-mult-pprod*
and *pprod.mult-scalar* = *mult-scalar-pprod*
and *pprod.adds-term* = *adds-term-pprod*
for *cmp-term* :: (('*a*::*nat*, '*b*::*nat*) *pp* × '*c*::{*nat*,*the-min*}) *nat-term-order*
defines *shift-map-keys-pprod* = *pprod'.shift-map-keys*
and *min-term-pprod* = *pprod'.min-term*
and *lt-pprod* = *pprod'.lt*
and *lc-pprod* = *pprod'.lc*
and *tail-pprod* = *pprod'.tail*
and *comp-opt-p-pprod* = *pprod'.comp-opt-p*
and *ord-p-pprod* = *pprod'.ord-p*
and *ord-strict-p-pprod* = *pprod'.ord-strict-p*
and *find-adds-pprod* = *pprod'.find-adds*
and *trd-aux-pprod*= *pprod'.trd-aux*
and *trd-pprod* = *pprod'.trd*
and *spoly-pprod* = *pprod'.spoly*
and *count-const-lt-components-pprod* = *pprod'.count-const-lt-components*
and *count-rem-components-pprod* = *pprod'.count-rem-components*
and *const-lt-component-pprod* = *pprod'.const-lt-component*
and *full-gb-pprod* = *pprod'.full-gb*
and *keys-to-list-pprod* = *pprod'.keys-to-list*
and *Keys-to-list-pprod* = *pprod'.Keys-to-list*
and *add-pairs-single-sorted-pprod* = *pprod'.add-pairs-single-sorted*
and *add-pairs-pprod* = *pprod'.add-pairs*
and *canon-pair-order-aux-pprod* = *pprod'.canon-pair-order-aux*
and *canon-basis-order-pprod* = *pprod'.canon-basis-order*
and *new-pairs-sorted-pprod* = *pprod'.new-pairs-sorted*
and *component-crit-pprod* = *pprod'.component-crit*
and *chain-ncrit-pprod* = *pprod'.chain-ncrit*
and *chain-ocrit-pprod* = *pprod'.chain-ocrit*
and *apply-icrit-pprod* = *pprod'.apply-icrit*
and *apply-ncrit-pprod* = *pprod'.apply-ncrit*
and *apply-ocrit-pprod* = *pprod'.apply-ocrit*
and *trdsp-pprod* = *pprod'.trdsp*
and *gb-sel-pprod* = *pprod'.gb-sel*
and *gb-red-aux-pprod* = *pprod'.gb-red-aux*
and *gb-red-pprod* = *pprod'.gb-red*
and *gb-aux-pprod* = *pprod'.gb-aux*
and *gb-pprod* = *pprod'.gb*
subgoal by (*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*: *monom-mult-pprod-def*)
subgoal by (*simp only*: *mult-scalar-pprod-def*)
subgoal by (*simp only*: *adds-term-pprod-def*)
done

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-shift-map-keys-pprod* [*code abstract*]:
list-of-oalist-ntm (*shift-map-keys-pprod t f xs*) = *map-raw* (λ(*k*, *v*). (*splus-pprod t k*, *f v*)) (*list-of-oalist-ntm xs*)
by (*simp add*: *pprod'.list-of-oalist-shift-keys case-prod-beta'*)

lemma *compute-trd-pprod* [*code*]: *trd-pprod to fs p* = *trd-aux-pprod to fs p* (*change-ord to 0*)
by (*simp only*: *pprod'.trd-def change-ord-def*)

lemmas [*code*] = *conversep-iff*

definition Vec_0 :: *nat* ⇒ ((*'a*, *nat*) *pp* $\Rightarrow_0$ *'b*) ⇒ ((*'a*::*nat*, *nat*) *pp* × *nat*) $\Rightarrow_0$ *'b*::*semiring-1* **where**
Vec_0 *i p* = *mult-scalar-pprod p* (*Poly-Mapping.single* (*0*, *i*) *1*)

experiment begin interpretation *trivariate*$_0$*-rat* **.**

lemma
ord-p-pprod (*POT DRLEX*) (Vec_0 *1* ($X^2 * Z$) + Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)) (Vec_0 *1* ($X^2 * Z^2 + 2 * Y$ ^ $3 * Z^2$))
by *eval*

lemma
tail-pprod (*POT DRLEX*) (Vec_0 *1* ($X^2 * Z$) + Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)) = Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)
by *eval*

lemma
lt-pprod (*POT DRLEX*) (Vec_0 *1* ($X^2 * Z$) + Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)) = ($sparse_0$ [(*0*, *2*), (*2*, *1*)], *1*)
by *eval*

lemma
keys (Vec_0 *0* ($X^2 * Z$ ^ *3*) + Vec_0 *1* ($2 * Y$ ^ $3 * Z^2$)) =
{($sparse_0$ [(*0*, *2*), (*2*, *3*)], *0*), ($sparse_0$ [(*1*, *3*), (*2*, *2*)], *1*)}
by *eval*

lemma
keys (Vec_0 *0* ($X^2 * Z$ ^ *3*) + Vec_0 *2* ($2 * Y$ ^ $3 * Z^2$)) =
{($sparse_0$ [(*0*, *2*), (*2*, *3*)], *0*), ($sparse_0$ [(*1*, *3*), (*2*, *2*)], *2*)}
by *eval*

lemma
Vec$_0$ *1* (X^2 ∗ *Z* ^ *7* + *2* ∗ *Y* ^ *3* ∗ Z^2) + *Vec*$_0$ *3* (X^2 ∗ *Z* ^ *4*) + *Vec*$_0$ *1* (− *2* ∗ *Y* ^ *3* ∗ Z^2) =
Vec$_0$ *1* (X^2 ∗ *Z* ^ *7*) + *Vec*$_0$ *3* (X^2 ∗ *Z* ^ *4*)
by *eval*

lemma
lookup (*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *7*) + *Vec*$_0$ *1* (*2* ∗ *Y* ^ *3* ∗ Z^2 + *2*)) (*sparse*$_0$ [(*0*, *2*), (*2*, *7*)], *0*) = *1*
by *eval*

lemma
lookup (*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *7*) + *Vec*$_0$ *1* (*2* ∗ *Y* ^ *3* ∗ Z^2 + *2*)) (*sparse*$_0$ [(*0*, *2*), (*2*, *7*)], *1*) = *0*
by *eval*

lemma
Vec$_0$ *0* (*0* ∗ *X*^*2* ∗ *Z*^*7*) + *Vec*$_0$ *1* (*0* ∗ *Y*^*3*∗Z^2) = *0*
by *eval*

lemma
monom-mult-pprod 3 (*sparse*$_0$ [(*1*, *2*::*nat*)]) (*Vec*$_0$ *0* (X^2 ∗ *Z*) + *Vec*$_0$ *1* (*2* ∗ *Y* ^ *3* ∗ Z^2)) =
Vec$_0$ *0* (*3* ∗ Y^2 ∗ *Z* ∗ X^2) + *Vec*$_0$ *1* (*6* ∗ *Y* ^ *5* ∗ Z^2)
by *eval*

lemma
trd-pprod DRLEX [*Vec*$_0$ *0* (Y^2 ∗ *Z* + *2* ∗ *Y* ∗ *Z* ^ *3*)] (*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *4* − *2* ∗ *Y* ^ *3* ∗ *Z* ^ *3*)) =
Vec$_0$ *0* (X^2 ∗ *Z* ^ *4* + *Y* ^ *4* ∗ *Z*)
by *eval*

lemma
length (*gb-pprod* (*POT DRLEX*)
[
(*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *4* − *2* ∗ *Y* ^ *3* ∗ Z^2), ()),
(*Vec*$_0$ *0* (Y^2 ∗ *Z* + *2* ∗ *Z* ^ *3*), ())
] ()) = *4*
by *eval*

end

end

11 Further Properties of Multivariate Polynomials

```
theory More-MPoly-Type-Class
  imports Polynomials.MPoly-Type-Class-Ordered General
begin
```

Some further general properties of (ordered) multivariate polynomials needed for Gröbner bases. This theory is an extension of *Polynomials.MPoly-Type-Class-Ordered*.

11.1 Modules and Linear Hulls

```
context module
begin

lemma span-listE:
  assumes p ∈ span (set bs)
  obtains qs where length qs = length bs and p = sum-list (map2 (∗s) qs bs)
proof −
  have finite (set bs) ..
   from this assms obtain q where p: p = (∑ b∈set bs. (q b) ∗s b) by (rule
span-finiteE)
  let ?qs = map-dup q (λ-. 0) bs
  show ?thesis
  proof
    show length ?qs = length bs by simp
  next
    let ?zs = zip (map q (remdups bs)) (remdups bs)
    have ∗: distinct ?zs by (rule distinct-zipI2, rule distinct-remdups)
    have inj: inj-on (λb. (q b, b)) (set bs) by (rule, simp)
    have p = (∑ (q, b)←?zs. q ∗s b)
     by (simp add: sum-list-distinct-conv-sum-set[OF ∗] set-zip-map1 p comm-monoid-add-class.sum.reindex[OF
inj])
    also have ... = (∑ (q, b)←(filter (λ(q, b). q ≠ 0) ?zs). q ∗s b)
      by (rule monoid-add-class.sum-list-map-filter[symmetric], auto)
    also have ... = (∑ (q, b)←(filter (λ(q, b). q ≠ 0) (zip ?qs bs)). q ∗s b)
      by (simp only: filter-zip-map-dup-const)
    also have ... = (∑ (q, b)←zip ?qs bs. q ∗s b)
      by (rule monoid-add-class.sum-list-map-filter, auto)
    finally show p = (∑ (q, b)←zip ?qs bs. q ∗s b) .
  qed
qed

lemma span-listI: sum-list (map2 (∗s) qs bs) ∈ span (set bs)
proof (induct qs arbitrary: bs)
  case Nil
  show ?case by (simp add: span-zero)
next
  case step: (Cons q qs)
```

show *?case*
proof (*simp add*: *zip-Cons1 span-zero split*: *list.split*, *intro allI impI*)
fix *a as*
have *sum-list* (*map2* (*∗s*) *qs as*) ∈ *span* (*insert a* (*set as*)) (**is** *?x* ∈ *?A*)
by (*rule*, *fact step*, *rule span-mono*, *auto*)
moreover have *a* ∈ *?A* **by** (*rule span-base*) *simp*
ultimately show *q ∗s a* + *?x* ∈ *?A* **by** (*intro span-add span-scale*)
qed
qed

end

lemma (**in** *term-powerprod*) *monomial-1-in-pmdlI*:
assumes (*f*::- $\Rightarrow_0$ *'b*::*field*) ∈ *pmdl F* **and** *keys f* = {*t*}
shows *monomial 1 t* ∈ *pmdl F*
proof −
define *c* **where** *c* ≡ *lookup f t*
from *assms*(*2*) **have** *f-eq*: *f* = *monomial c t* **unfolding** *c-def*
by (*metis* (*mono-tags*, *lifting*) *Diff-insert-absorb cancel-comm-monoid-add-class.add-cancel-right-right plus-except insert-absorb insert-not-empty keys-eq-empty keys-except*)
from *assms*(*2*) **have** *c* ≠ *0*
unfolding *c-def* **by** *auto*
hence *monomial 1 t* = *monom-mult* (*1* / *c*) *0 f* **by** (*simp add*: *f-eq monom-mult-monomial term-simps*)
also from *assms*(*1*) **have** ... ∈ *pmdl F* **by** (*rule pmdl-closed-monom-mult*)
finally show *?thesis* **.**
qed

11.2 Ordered Polynomials

context *ordered-term*
begin

11.2.1 Sets of Leading Terms and -Coefficients

definition *lt-set* :: (*'t*, *'b*::*zero*) *poly-mapping set* ⇒ *'t set* **where**
lt-set F = *lt* ‘ (*F* − {*0*})

definition *lc-set* :: (*'t*, *'b*::*zero*) *poly-mapping set* ⇒ *'b set* **where**
lc-set F = *lc* ‘ (*F* − {*0*})

lemma *lt-setI*:
assumes *f* ∈ *F* **and** *f* ≠ *0*
shows *lt f* ∈ *lt-set F*
unfolding *lt-set-def* **using** *assms* **by** *simp*

lemma *lt-setE*:
assumes *t* ∈ *lt-set F*
obtains *f* **where** *f* ∈ *F* **and** *f* ≠ *0* **and** *lt f* = *t*
using *assms* **unfolding** *lt-set-def* **by** *auto*

lemma *lt-set-iff*:
  shows $t \in$ *lt-set* $F \longleftrightarrow (\exists f \in F.\ f \neq 0 \wedge lt\ f = t)$
  unfolding *lt-set-def* **by** *auto*

lemma *lc-setI*:
  assumes $f \in F$ **and** $f \neq 0$
  shows *lc f* $\in$ *lc-set F*
  unfolding *lc-set-def* **using** *assms* **by** *simp*

lemma *lc-setE*:
  assumes $c \in$ *lc-set F*
  obtains *f* **where** $f \in F$ **and** $f \neq 0$ **and** *lc f* $= c$
  using *assms* **unfolding** *lc-set-def* **by** *auto*

lemma *lc-set-iff*:
  shows $c \in$ *lc-set* $F \longleftrightarrow (\exists f \in F.\ f \neq 0 \wedge lc\ f = c)$
  unfolding *lc-set-def* **by** *auto*

lemma *lc-set-nonzero*:
  shows $0 \notin$ *lc-set F*
proof
  assume $0 \in$ *lc-set F*
  then obtain *f* **where** $f \in F$ **and** $f \neq 0$ **and** *lc f* $= 0$ **by** (*rule lc-setE*)
  from ‹$f \neq 0$› **have** *lc f* $\neq 0$ **by** (*rule lc-not-0*)
  from *this* ‹*lc f* $= 0$› **show** *False* **..**
qed

lemma *lt-sum-distinct-eq-Max*:
  assumes *finite I* **and** *sum p I* $\neq 0$
    and $\bigwedge i1\ i2.\ i1 \in I \Longrightarrow i2 \in I \Longrightarrow p\ i1 \neq 0 \Longrightarrow p\ i2 \neq 0 \Longrightarrow lt\ (p\ i1) = lt$
$(p\ i2) \Longrightarrow i1 = i2$
  shows *lt* (*sum p I*) = *ord-term-lin.Max* (*lt-set* (*p* ‘ *I*))
proof −
  have ¬ *p* ‘ *I* $\subseteq \{0\}$
  proof
    assume *p* ‘ *I* $\subseteq \{0\}$
    hence *sum p I* $= 0$ **by** (*rule sum-poly-mapping-eq-zeroI*)
    with *assms*(*2*) **show** *False* **..**
  qed
  from *assms*(*1*) *this assms*(*3*) **show** *?thesis*
  proof (*induct I*)
    case *empty*
    from *empty*(*1*) **show** *?case* **by** *simp*
  next
    case (*insert x I*)
    show *?case*
    proof (*cases p* ‘ *I* $\subseteq \{0\}$)
      case *True*

hence *p ‘ I* − {*0*} = {} **by** *simp*
have *p x* ≠ *0*
proof
assume *p x* = *0*
with *True* **have** *p ‘ insert x I* ⊆ {*0*} **by** *simp*
with *insert*(*4*) **show** *False* **..**
qed
hence *insert* (*p x*) (*p ‘ I*) − {*0*} = *insert* (*p x*) (*p ‘ I* − {*0*}) **by** *auto*
hence *lt-set* (*p ‘ insert x I*) = {*lt* (*p x*)} **by** (*simp add*: *lt-set-def* ‹*p ‘ I* − {*0*} = {}›)
hence *eq1*: *ord-term-lin.Max* (*lt-set* (*p ‘ insert x I*)) = *lt* (*p x*) **by** *simp*
have *eq2*: *sum p I* = *0*
proof (*rule ccontr*)
assume *sum p I* ≠ *0*
then obtain *y* **where** *y* ∈ *I* **and** *p y* ≠ *0* **by** (*rule sum.not-neutral-contains-not-neutral*)
with *True* **show** *False* **by** *auto*
qed
show *?thesis* **by** (*simp only*: *eq1 sum.insert*[*OF insert*(*1*) *insert*(*2*)], *simp add*: *eq2*)
next
case *False*
hence *IH*: *lt* (*sum p I*) = *ord-term-lin.Max* (*lt-set* (*p ‘ I*))
proof (*rule insert*(*3*))
fix *i1 i2*
assume *i1* ∈ *I* **and** *i2* ∈ *I*
hence *i1* ∈ *insert x I* **and** *i2* ∈ *insert x I* **by** *simp-all*
moreover assume *p i1* ≠ *0* **and** *p i2* ≠ *0* **and** *lt* (*p i1*) = *lt* (*p i2*)
ultimately show *i1* = *i2* **by** (*rule insert*(*5*))
qed
show *?thesis*
proof (*cases p x* = *0*)
case *True*
hence *eq*: *lt-set* (*p ‘ insert x I*) = *lt-set* (*p ‘ I*) **by** (*simp add*: *lt-set-def*)
show *?thesis* **by** (*simp only*: *eq*, *simp add*: *sum.insert*[*OF insert*(*1*) *insert*(*2*)] *True*, *fact IH*)
next
case *False*
hence *eq1*: *lt-set* (*p ‘ insert x I*) = *insert* (*lt* (*p x*)) (*lt-set* (*p ‘ I*))
by (*auto simp add*: *lt-set-def*)
from *insert*(*1*) **have** *finite* (*lt-set* (*p ‘ I*)) **by** (*simp add*: *lt-set-def*)
moreover from ‹¬ *p ‘ I* ⊆ {*0*}› **have** *lt-set* (*p ‘ I*) ≠ {} **by** (*simp add*: *lt-set-def*)
ultimately have *eq2*: *ord-term-lin.Max* (*insert* (*lt* (*p x*)) (*lt-set* (*p ‘ I*))) =
ord-term-lin.max (*lt* (*p x*)) (*ord-term-lin.Max* (*lt-set* (*p ‘ I*)))
by (*rule ord-term-lin.Max-insert*)
show *?thesis*
proof (*simp only*: *eq1*, *simp add*: *sum.insert*[*OF insert*(*1*) *insert*(*2*)] *eq2 IH*[*symmetric*],
rule lt-plus-distinct-eq-max, *rule*)

```
        assume *: lt (p x) = lt (sum p I)
        have lt (p x) ∈ lt-set (p ` I) by (simp only: * IH, rule ord-term-lin.Max-in,
fact+)
        then obtain f where f ∈ p ` I and f ≠ 0 and ltf: lt f = lt (p x) by
(rule lt-setE)
        from this(1) obtain y where y ∈ I and f = p y ..
        from this(2) ‹f ≠ 0› ltf have p y ≠ 0 and lt-eq: lt (p y) = lt (p x) by
simp-all
        from - - this(1) ‹p x ≠ 0› this(2) have y = x
        proof (rule insert(5))
          from ‹y ∈ I› show y ∈ insert x I by simp
        next
          show x ∈ insert x I by simp
        qed
        with ‹y ∈ I› have x ∈ I by simp
        with ‹x ∉ I› show False ..
      qed
    qed
  qed
 qed
qed

lemma lt-sum-distinct-in-lt-set:
 assumes finite I and sum p I ≠ 0
  and ⋀i1 i2. i1 ∈ I ⟹ i2 ∈ I ⟹ p i1 ≠ 0 ⟹ p i2 ≠ 0 ⟹ lt (p i1) = lt
(p i2) ⟹ i1 = i2
 shows lt (sum p I) ∈ lt-set (p ` I)
proof −
 have ¬ p ` I ⊆ {0}
 proof
  assume p ` I ⊆ {0}
  hence sum p I = 0 by (rule sum-poly-mapping-eq-zeroI)
  with assms(2) show False ..
 qed
 have lt (sum p I) = ord-term-lin.Max (lt-set (p ` I))
  by (rule lt-sum-distinct-eq-Max, fact+)
 also have ... ∈ lt-set (p ` I)
 proof (rule ord-term-lin.Max-in)
  from assms(1) show finite (lt-set (p ` I)) by (simp add: lt-set-def)
 next
  from ‹¬ p ` I ⊆ {0}› show lt-set (p ` I) ≠ {} by (simp add: lt-set-def)
 qed
 finally show ?thesis .
qed
```

11.2.2 Monicity

```
definition monic :: ('t ⇒₀ 'b) ⇒ ('t ⇒₀ 'b::field) where
 monic p = monom-mult (1 / lc p) 0 p
```

definition *is-monic-set* :: (′*t* ⇒$_0$ ′*b*::*field*) *set* ⇒ *bool* **where**
  is-monic-set B ≡ (∀ *b*∈*B*. *b* ≠ *0* ⟶ *lc b* = *1*)

lemma *lookup-monic*: *lookup* (*monic p*) *v* = (*lookup p v*) / *lc p*
proof −
  have *lookup* (*monic p*) (*0* ⊕ *v*) = (*1* / *lc p*) ∗ (*lookup p v*) **unfolding** *monic-def*
    by (*rule lookup-monom-mult-plus*)
  thus *?thesis* **by** (*simp add*: *term-simps*)
qed

lemma *lookup-monic-lt*:
  assumes *p* ≠ *0*
  shows *lookup* (*monic p*) (*lt p*) = *1*
  unfolding *monic-def*
proof −
  from *assms* **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
  hence *1* / *lc p* ≠ *0* **by** *simp*
  let *?q* = *monom-mult* (*1* / *lc p*) *0 p*
  have *lookup ?q* (*0* ⊕ *lt p*) = (*1* / *lc p*) ∗ (*lookup p* (*lt p*)) **by** (*rule lookup-monom-mult-plus*)
  also have ... = (*1* / *lc p*) ∗ *lc p* **unfolding** *lc-def* **..**
  also have ... = *1* **using** ‹*lc p* ≠ *0*› **by** *simp*
  finally have *lookup ?q* (*0* ⊕ *lt p*) = *1* **.**
  thus *lookup ?q* (*lt p*) = *1* **by** (*simp add*: *term-simps*)
qed

lemma *monic-0* [*simp*]: *monic 0* = *0*
  unfolding *monic-def* **by** (*rule monom-mult-zero-right*)

lemma *monic-0-iff*: (*monic p* = *0*) ⟷ (*p* = *0*)
proof
  assume *monic p* = *0*
  show *p* = *0*
  proof (*rule ccontr*)
    assume *p* ≠ *0*
    hence *lookup* (*monic p*) (*lt p*) = *1* **by** (*rule lookup-monic-lt*)
    with ‹*monic p* = *0*› **have** *lookup 0* (*lt p*) = (*1*::′*b*) **by** *simp*
    thus *False* **by** *simp*
  qed
next
  assume *p0*: *p* = *0*
  show *monic p* = *0* **unfolding** *p0* **by** (*fact monic-0*)
qed

lemma *keys-monic* [*simp*]: *keys* (*monic p*) = *keys p*
proof (*cases p* = *0*)
  case *True*
  show *?thesis* **unfolding** *True monic-0* **..**
next

case *False*
hence *lc p* ≠ *0* **by** (*rule lc-not-0*)
show *?thesis* **by** (*rule set-eqI*, *simp add*: *in-keys-iff lookup-monic* ‹*lc p* ≠ *0*›)
qed

lemma *lt-monic* [*simp*]: *lt* (*monic p*) = *lt p*
proof (*cases p* = *0*)
case *True*
show *?thesis* **unfolding** *True monic-0* **..**
next
case *False*
have *lt* (*monom-mult* (*1* / *lc p*) *0 p*) = *0* ⊕ *lt p*
proof (*rule lt-monom-mult*)
from *False* **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
thus *1* / *lc p* ≠ *0* **by** *simp*
qed *fact*
thus *?thesis* **by** (*simp add*: *monic-def term-simps*)
qed

lemma *lc-monic*:
assumes *p* ≠ *0*
shows *lc* (*monic p*) = *1*
using *assms* **by** (*simp add*: *lc-def lookup-monic-lt*)

lemma *mult-lc-monic*:
assumes *p* ≠ *0*
shows *monom-mult* (*lc p*) *0* (*monic p*) = *p* (**is** *?q* = *p*)
proof (*rule poly-mapping-eqI*)
fix *v*
from *assms* **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
have *lookup ?q* (*0* ⊕ *v*) = (*lc p*) ∗ (*lookup* (*monic p*) *v*) **by** (*rule lookup-monom-mult-plus*)
also have ... = (*lc p*) ∗ ((*lookup p v*) / *lc p*) **by** (*simp add*: *lookup-monic*)
also have ... = *lookup p v* **using** ‹*lc p* ≠ *0*› **by** *simp*
finally show *lookup ?q v* = *lookup p v* **by** (*simp add*: *term-simps*)
qed

lemma *is-monic-setI*:
assumes ⋀*b*. *b* ∈ *B* ⟹ *b* ≠ *0* ⟹ *lc b* = *1*
shows *is-monic-set B*
unfolding *is-monic-set-def* **using** *assms* **by** *auto*

lemma *is-monic-setD*:
assumes *is-monic-set B* **and** *b* ∈ *B* **and** *b* ≠ *0*
shows *lc b* = *1*
using *assms* **unfolding** *is-monic-set-def* **by** *auto*

lemma *Keys-image-monic* [*simp*]: *Keys* (*monic* ‘ *A*) = *Keys A*
by (*simp add*: *Keys-def*)

```
lemma image-monic-is-monic-set: is-monic-set (monic ` A)
proof (rule is-monic-setI)
  fix p
  assume pin: p ∈ monic ` A and p ≠ 0
  from pin obtain p' where p-def: p = monic p' and p' ∈ A ..
  from ‹p ≠ 0› have p' ≠ 0 unfolding p-def monic-0-iff .
  thus lc p = 1 unfolding p-def by (rule lc-monic)
qed

lemma pmdl-image-monic [simp]: pmdl (monic ` B) = pmdl B
proof
  show pmdl (monic ` B) ⊆ pmdl B
  proof
    fix p
    assume p ∈ pmdl (monic ` B)
    thus p ∈ pmdl B
    proof (induct p rule: pmdl-induct)
      case base: module-0
      show ?case by (fact pmdl.span-zero)
    next
      case ind: (module-plus a b c t)
      from ind(3) obtain b' where b-def: b = monic b' and b' ∈ B ..
      have eq: b = monom-mult (1 / lc b') 0 b' by (simp only: b-def monic-def)
      show ?case unfolding eq monom-mult-assoc
        by (rule pmdl.span-add, fact, rule monom-mult-in-pmdl, fact)
    qed
  qed
next
  show pmdl B ⊆ pmdl (monic ` B)
  proof
    fix p
    assume p ∈ pmdl B
    thus p ∈ pmdl (monic ` B)
    proof (induct p rule: pmdl-induct)
      case base: module-0
      show ?case by (fact pmdl.span-zero)
    next
      case ind: (module-plus a b c t)
      show ?case
      proof (cases b = 0)
        case True
        from ind(2) show ?thesis by (simp add: True)
      next
        case False
        let ?b = monic b
        from ind(3) have ?b ∈ monic ` B by (rule imageI)
        have a + monom-mult c t (monom-mult (lc b) 0 ?b) ∈ pmdl (monic ` B)
          unfolding monom-mult-assoc
          by (rule pmdl.span-add, fact, rule monom-mult-in-pmdl, fact)
```

thus *?thesis* **unfolding** *mult-lc-monic*[*OF False*] **.**
qed
qed
qed
qed

end

end

12 Auto-reducing Lists of Polynomials

theory *Auto-Reduction*
imports *Reduction More-MPoly-Type-Class*
begin

12.1 Reduction and Monic Sets

context *ordered-term*
begin

lemma *is-red-monic*: *is-red B* (*monic p*) ⟷ *is-red B p*
unfolding *is-red-adds-iff keys-monic* **..**

lemma *red-image-monic* [*simp*]: *red* (*monic* ‘ *B*) = *red B*
proof (*rule*, *rule*)
fix *p q*
show *red* (*monic* ‘ *B*) *p q* ⟷ *red B p q*
proof
assume *red* (*monic* ‘ *B*) *p q*
then obtain *f t* **where** *f* ∈ *monic* ‘ *B* **and** ∗: *red-single p q f t* **by** (*rule red-setE*)
from *this*(*1*) **obtain** *g* **where** *g* ∈ *B* **and** *f* = *monic g* **..**
from ∗ **have** *f* ≠ *0* **by** (*simp add*: *red-single-def*)
hence *g* ≠ *0* **by** (*simp add*: *monic-0-iff* ‹*f* = *monic g*›)
hence *lc g* ≠ *0* **by** (*rule lc-not-0*)
have *eq*: *monom-mult* (*lc g*) *0 f* = *g* **by** (*simp add*: ‹*f* = *monic g*› *mult-lc-monic*[*OF* ‹*g* ≠ *0*›])
from ‹*g* ∈ *B*› **show** *red B p q*
proof (*rule red-setI*)
from ∗ ‹*lc g* ≠ *0*› **have** *red-single p q* (*monom-mult* (*lc g*) *0 f*) *t* **by** (*rule red-single-mult-const*)
thus *red-single p q g t* **by** (*simp only*: *eq*)
qed
next
assume *red B p q*
then obtain *f t* **where** *f* ∈ *B* **and** ∗: *red-single p q f t* **by** (*rule red-setE*)
from ∗ **have** *f* ≠ *0* **by** (*simp add*: *red-single-def*)
hence *lc f* ≠ *0* **by** (*rule lc-not-0*)

hence *1 / lc f* ≠ *0* **by** *simp*
from ‹*f* ∈ *B*› **have** *monic f* ∈ *monic* ‘ *B* **by** (*rule imageI*)
thus *red* (*monic* ‘ *B*) *p q*
proof (*rule red-setI*)
from ∗ ‹*1 / lc f* ≠ *0*› **show** *red-single p q* (*monic f*) *t* **unfolding** *monic-def*
by (*rule red-single-mult-const*)
qed
qed
qed

lemma *is-red-image-monic* [*simp*]: *is-red* (*monic* ‘ *B*) *p* ⟷ *is-red B p*
by (*simp add*: *is-red-def*)

12.2 Minimal Bases and Auto-reduced Bases

definition *is-auto-reduced* :: (*'t* ⇒₀ *'b*::*field*) *set* ⇒ *bool* **where**
is-auto-reduced B ≡ (∀ *b*∈*B*. ¬ *is-red* (*B* − {*b*}) *b*)

definition *is-minimal-basis* :: (*'t* $\Rightarrow_0$ *'b*::*zero*) *set* ⇒ *bool* **where**
is-minimal-basis B ⟷ (*0* ∉ *B* ∧ (∀ *p q*. *p* ∈ *B* ⟶ *q* ∈ *B* ⟶ *p* ≠ *q* ⟶ ¬ *lt p* $adds_t$ *lt q*))

lemma *is-auto-reducedD*:
assumes *is-auto-reduced B* **and** *b* ∈ *B*
shows ¬ *is-red* (*B* − {*b*}) *b*
using *assms* **unfolding** *is-auto-reduced-def* **by** *auto*

The converse of the following lemma is only true if *B* is minimal!

lemma *image-monic-is-auto-reduced*:
assumes *is-auto-reduced B*
shows *is-auto-reduced* (*monic* ‘ *B*)
unfolding *is-auto-reduced-def*
proof
fix *b*
assume *b* ∈ *monic* ‘ *B*
then obtain *b'* **where** *b-def*: *b* = *monic b'* **and** *b'* ∈ *B* **..**
from *assms* ‹*b'* ∈ *B*› **have** *nred*: ¬ *is-red* (*B* − {*b'*}) *b'* **by** (*rule is-auto-reducedD*)
show ¬ *is-red* ((*monic* ‘ *B*) − {*b*}) *b*
proof
assume *red*: *is-red* ((*monic* ‘ *B*) − {*b*}) *b*
have (*monic* ‘ *B*) − {*b*} ⊆ *monic* ‘ (*B* − {*b'*}) **unfolding** *b-def* **by** *auto*
with *red* **have** *is-red* (*monic* ‘ (*B* − {*b'*})) *b* **by** (*rule is-red-subset*)
hence *is-red* (*B* − {*b'*}) *b'* **unfolding** *b-def is-red-monic is-red-image-monic* **.**
with *nred* **show** *False* **..**
qed
qed

lemma *is-minimal-basisI*:
assumes ⋀*p*. *p* ∈ *B* ⟹ *p* ≠ *0* **and** ⋀*p q*. *p* ∈ *B* ⟹ *q* ∈ *B* ⟹ *p* ≠ *q* ⟹ ¬

lt p $adds_t$ *lt q*
shows *is-minimal-basis B*
unfolding *is-minimal-basis-def* **using** *assms* **by** *auto*

lemma *is-minimal-basisD1*:
assumes *is-minimal-basis B* **and** $p \in B$
shows $p \neq 0$
using *assms* **unfolding** *is-minimal-basis-def* **by** *auto*

lemma *is-minimal-basisD2*:
assumes *is-minimal-basis B* **and** $p \in B$ **and** $q \in B$ **and** $p \neq q$
shows ¬ *lt p* $adds_t$ *lt q*
using *assms* **unfolding** *is-minimal-basis-def* **by** *auto*

lemma *is-minimal-basisD3*:
assumes *is-minimal-basis B* **and** $p \in B$ **and** $q \in B$ **and** $p \neq q$
shows ¬ *lt q* $adds_t$ *lt p*
using *assms* **unfolding** *is-minimal-basis-def* **by** *auto*

lemma *is-minimal-basis-subset*:
assumes *is-minimal-basis B* **and** $A \subseteq B$
shows *is-minimal-basis A*
proof (*intro is-minimal-basisI*)
fix *p*
assume $p \in A$
with ‹$A \subseteq B$› **have** $p \in B$ **..**
with ‹*is-minimal-basis B*› **show** $p \neq 0$ **by** (*rule is-minimal-basisD1*)
next
fix *p q*
assume $p \in A$ **and** $q \in A$ **and** $p \neq q$
from ‹$p \in A$› **and** ‹$q \in A$› **have** $p \in B$ **and** $q \in B$ **using** ‹$A \subseteq B$› **by** *auto*
from ‹*is-minimal-basis B*› *this* ‹$p \neq q$› **show** ¬ *lt p* $adds_t$ *lt q* **by** (*rule is-minimal-basisD2*)
qed

lemma *nadds-red*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow$ ¬ *lt q* $adds_t$ *lt p* **and** *red*: *red B p r*
shows $r \neq 0 \wedge lt\ r = lt\ p$
proof −
from *red* **obtain** *q t* **where** $q \in B$ **and** *rs*: *red-single p r q t* **by** (*rule red-setE*)
from *rs* **have** $q \neq 0$ **and** *lookup p* $(t \oplus lt\ q) \neq 0$
and *r-def*: $r = p -$ *monom-mult* (*lookup p* $(t \oplus lt\ q)$ / *lc q*) *t q* **unfolding** *red-single-def* **by** *simp-all*
have $t \oplus lt\ q \preceq_t lt\ p$ **by** (*rule lt-max*, *fact*)
moreover have $t \oplus lt\ q \neq lt\ p$
proof
assume $t \oplus lt\ q = lt\ p$
hence *lt q* $adds_t$ *lt p* **by** (*metis adds-term-triv*)
with *nadds*[*OF* ‹$q \in B$›] **show** *False* **..**

qed
ultimately have $t \oplus lt\ q \prec_t lt\ p$ **by** *simp*
let *?m* = *monom-mult* (*lookup p* ($t \oplus lt\ q$) / *lc q*) *t q*
from ‹*lookup p* ($t \oplus lt\ q$) ≠ *0*› *lc-not-0*[*OF* ‹*q* ≠ *0*›] **have** *c0*: *lookup p* ($t \oplus lt$ *q*) / *lc q* ≠ *0* **by** *simp*
from ‹*q* ≠ *0*› *c0* **have** *?m* ≠ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
have *lt* (− *?m*) = *lt ?m* **by** (*fact lt-uminus*)
also have *lt1*: *lt ?m* = $t \oplus lt\ q$ **by** (*rule lt-monom-mult*, *fact*+)
finally have *lt2*: *lt* (− *?m*) = $t \oplus lt\ q$ **.**

show *?thesis*
proof
  show *r* ≠ *0*
  proof
    assume *r* = *0*
    hence *p* = *?m* **unfolding** *r-def* **by** *simp*
    with *lt1* ‹$t \oplus lt\ q \neq lt\ p$› **show** *False* **by** *simp*
  qed
next
  have *lt* (− *?m* + *p*) = *lt p*
  proof (*rule lt-plus-eqI*)
    show *lt* (− *?m*) $\prec_t$ *lt p* **unfolding** *lt2* **by** *fact*
  qed
  thus *lt r* = *lt p* **unfolding** *r-def* **by** *simp*
qed
qed

lemma *nadds-red-nonzero*:
assumes *nadds*: ⋀*q*. *q* ∈ *B* ⟹ ¬ *lt q* $adds_t$ *lt p* **and** *red B p r*
shows *r* ≠ *0*
using *nadds-red*[*OF assms*] **by** *simp*

lemma *nadds-red-lt*:
assumes *nadds*: ⋀*q*. *q* ∈ *B* ⟹ ¬ *lt q* $adds_t$ *lt p* **and** *red B p r*
shows *lt r* = *lt p*
using *nadds-red*[*OF assms*] **by** *simp*

lemma *nadds-red-rtrancl-lt*:
assumes *nadds*: ⋀*q*. *q* ∈ *B* ⟹ ¬ *lt q* $adds_t$ *lt p* **and** *rtrancl*: (*red B*)** *p r*
shows *lt r* = *lt p*
using *rtrancl*
proof (*induct rule*: *rtranclp-induct*)
case *base*
show *?case* **..**
next
case (*step y z*)
have *lt z* = *lt y*
proof (*rule nadds-red-lt*)
  fix *q*

assume $q \in B$
thus $\neg$ *lt q adds*$_t$ *lt y* **unfolding** ‹*lt y* = *lt p*› **by** (*rule nadds*)
qed *fact*
with ‹*lt y* = *lt p*› **show** *?case* **by** *simp*
qed

lemma *nadds-red-rtrancl-nonzero*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow \neg$ *lt q adds*$_t$ *lt p* **and** $p \neq 0$ **and** *rtrancl*: (*red* $B)^{**}$ *p r*
shows $r \neq 0$
using *rtrancl*
proof (*induct rule*: *rtranclp-induct*)
case *base*
show *?case* **by** *fact*
next
case (*step y z*)
from *nadds* ‹(*red B*)** *p y*› **have** *lt y* = *lt p* **by** (*rule nadds-red-rtrancl-lt*)
show $z \neq 0$
proof (*rule nadds-red-nonzero*)
fix *q*
assume $q \in B$
thus $\neg$ *lt q adds*$_t$ *lt y* **unfolding** ‹*lt y* = *lt p*› **by** (*rule nadds*)
qed *fact*
qed

lemma *minimal-basis-red-rtrancl-nonzero*:
assumes *is-minimal-basis B* **and** $p \in B$ **and** (*red* $(B - \{p\}))^{**}$ *p r*
shows $r \neq 0$
proof (*rule nadds-red-rtrancl-nonzero*)
fix *q*
assume $q \in (B - \{p\})$
hence $q \in B$ **and** $q \neq p$ **by** *auto*
show $\neg$ *lt q adds*$_t$ *lt p* **by** (*rule is-minimal-basisD2*, *fact+*)
next
show $p \neq 0$ **by** (*rule is-minimal-basisD1*, *fact+*)
qed *fact*

lemma *minimal-basis-red-rtrancl-lt*:
assumes *is-minimal-basis B* **and** $p \in B$ **and** (*red* $(B - \{p\}))^{**}$ *p r*
shows *lt r* = *lt p*
proof (*rule nadds-red-rtrancl-lt*)
fix *q*
assume $q \in (B - \{p\})$
hence $q \in B$ **and** $q \neq p$ **by** *auto*
show $\neg$ *lt q adds*$_t$ *lt p* **by** (*rule is-minimal-basisD2*, *fact+*)
qed *fact*

lemma *is-minimal-basis-replace*:
assumes *major*: *is-minimal-basis B* **and** $p \in B$ **and** *red*: (*red* $(B - \{p\}))^{**}$ *p r*

shows *is-minimal-basis* (*insert r* (*B* − {*p*}))
proof (*rule is-minimal-basisI*)
  fix *q*
  assume $q \in$ *insert r* ($B - \{p\}$)
  hence $q = r \vee q \in B \wedge q \neq p$ **by** *simp*
  thus $q \neq 0$
  proof
    assume $q = r$
    from *assms* **show** *?thesis* **unfolding** ‹$q = r$› **by** (*rule minimal-basis-red-rtrancl-nonzero*)
  next
    assume $q \in B \wedge q \neq p$
    hence $q \in B$ **..**
    with *major* **show** *?thesis* **by** (*rule is-minimal-basisD1*)
  qed
next
  fix *a b*
  assume $a \in$ *insert r* ($B - \{p\}$) **and** $b \in$ *insert r* ($B - \{p\}$) **and** $a \neq b$
  from *assms* **have** *ltr*: *lt r* = *lt p* **by** (*rule minimal-basis-red-rtrancl-lt*)
  from ‹$b \in$ *insert r* ($B - \{p\}$)› **have** *b*: $b = r \vee b \in B \wedge b \neq p$ **by** *simp*
  from ‹$a \in$ *insert r* ($B - \{p\}$)› **have** $a = r \vee a \in B \wedge a \neq p$ **by** *simp*
  thus ¬ *lt a* $adds_t$ *lt b*
  proof
    assume $a = r$
    hence *lta*: *lt a* = *lt p* **using** *ltr* **by** *simp*
    from *b* **show** *?thesis*
    proof
      assume $b = r$
      with ‹$a \neq b$› **show** *?thesis* **unfolding** ‹$a = r$› **by** *simp*
    next
      assume $b \in B \wedge b \neq p$
      hence $b \in B$ **and** $p \neq b$ **by** *auto*
      with *major* ‹$p \in B$› **have** ¬ *lt p* $adds_t$ *lt b* **by** (*rule is-minimal-basisD2*)
      thus *?thesis* **unfolding** *lta* **.**
    qed
  next
    assume $a \in B \wedge a \neq p$
    hence $a \in B$ **and** $a \neq p$ **by** *simp-all*
    from *b* **show** *?thesis*
    proof
      assume $b = r$
        from *major* ‹$a \in B$› ‹$p \in B$› ‹$a \neq p$› **have** ¬ *lt a* $adds_t$ *lt p* **by** (*rule*
is-minimal-basisD2)
      thus *?thesis* **unfolding** ‹$b = r$› *ltr* **by** *simp*
    next
      assume $b \in B \wedge b \neq p$
      hence $b \in B$ **..**
     from *major* ‹$a \in B$› ‹$b \in B$› ‹$a \neq b$› **show** *?thesis* **by** (*rule is-minimal-basisD2*)
    qed
  qed

qed

12.3 Computing Minimal Bases

definition *comp-min-basis* :: $('t \Rightarrow_0 'b)$ *list* $\Rightarrow$ $('t \Rightarrow_0 'b::zero)$ *list* **where**
comp-min-basis xs = *filter-min* ($\lambda x\ y.$ *lt x* $adds_t$ *lt y*) (*filter* ($\lambda x.\ x \neq 0$) *xs*)

lemma *comp-min-basis-subset'*: *set* (*comp-min-basis xs*) $\subseteq$ $\{x \in set\ xs.\ x \neq 0\}$
proof −
have *set* (*comp-min-basis xs*) $\subseteq$ *set* (*filter* ($\lambda x.\ x \neq 0$) *xs*)
unfolding *comp-min-basis-def* **by** (*rule filter-min-subset*)
also have ... = $\{x \in set\ xs.\ x \neq 0\}$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *comp-min-basis-subset*: *set* (*comp-min-basis xs*) $\subseteq$ *set xs*
proof −
have *set* (*comp-min-basis xs*) $\subseteq$ $\{x \in set\ xs.\ x \neq 0\}$ **by** (*rule comp-min-basis-subset'*)
also have ... $\subseteq$ *set xs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *comp-min-basis-nonzero*: $p \in$ *set* (*comp-min-basis xs*) $\Longrightarrow$ $p \neq 0$
using *comp-min-basis-subset'* **by** *blast*

lemma *comp-min-basis-adds*:
assumes $p \in$ *set xs* **and** $p \neq 0$
obtains *q* **where** $q \in$ *set* (*comp-min-basis xs*) **and** *lt q* $adds_t$ *lt p*
proof −
let *?rel* = ($\lambda x\ y.$ *lt x* $adds_t$ *lt y*)
have *transp ?rel* **by** (*auto intro*!: *transpI dest*: *adds-term-trans*)
moreover have *reflp ?rel* **by** (*simp add*: *reflp-def adds-term-refl*)
moreover from *assms* **have** $p \in$ *set* (*filter* ($\lambda x.\ x \neq 0$) *xs*) **by** *simp*
ultimately obtain *q* **where** $q \in$ *set* (*comp-min-basis xs*) **and** *lt q* $adds_t$ *lt p*
unfolding *comp-min-basis-def* **by** (*rule filter-min-relE*)
thus *?thesis* **..**
qed

lemma *comp-min-basis-is-red*:
assumes *is-red* (*set xs*) *f*
shows *is-red* (*set* (*comp-min-basis xs*)) *f*
proof −
from *assms* **obtain** *x t* **where** $x \in$ *set xs* **and** $t \in$ *keys f* **and** $x \neq 0$ **and** *lt x* $adds_t$ *t*
by (*rule is-red-addsE*)
from ‹$x \in$ *set xs*› ‹$x \neq 0$› **obtain** *y* **where** *yin*: $y \in$ *set* (*comp-min-basis xs*) **and** *lt y* $adds_t$ *lt x*
by (*rule comp-min-basis-adds*)
show *?thesis*

proof (*rule is-red-addsI*)
  from ‹*lt y adds*$_t$ *lt x*› ‹*lt x adds*$_t$ *t*› **show** *lt y adds*$_t$ *t* **by** (*rule adds-term-trans*)
next
  from *yin* **show** $y \neq 0$ **by** (*rule comp-min-basis-nonzero*)
qed *fact*+
qed

lemma *comp-min-basis-nadds*:
assumes $p \in$ *set* (*comp-min-basis xs*) **and** $q \in$ *set* (*comp-min-basis xs*) **and** $p \neq q$
shows ¬ *lt q adds*$_t$ *lt p*
proof
have *transp* (*λx y. lt x adds*$_t$ *lt y*) **by** (*auto intro*!: *transpI dest*: *adds-term-trans*)
moreover note *assms*(*2*, *1*)
moreover assume *lt q adds*$_t$ *lt p*
ultimately have $q = p$ **unfolding** *comp-min-basis-def* **by** (*rule filter-min-minimal*)
with *assms*(*3*) **show** *False* **by** *simp*
qed

lemma *comp-min-basis-is-minimal-basis*: *is-minimal-basis* (*set* (*comp-min-basis xs*))
by (*rule is-minimal-basisI*, *rule comp-min-basis-nonzero*, *assumption*, *rule comp-min-basis-nadds*, *assumption*+, *simp*)

lemma *comp-min-basis-distinct*: *distinct* (*comp-min-basis xs*)
unfolding *comp-min-basis-def* **by** (*rule filter-min-distinct*) (*simp add*: *reflp-def adds-term-refl*)

end

12.4 Auto-Reduction

context *gd-term*
begin

lemma *is-minimal-basis-trd-is-minimal-basis*:
assumes *is-minimal-basis* (*set* (*x* # *xs*)) **and** $x \notin$ *set xs*
shows *is-minimal-basis* (*set* ((*trd xs x*) # *xs*))
proof −
from *assms*(*1*) **have** *is-minimal-basis* (*insert* (*trd xs x*) (*set* (*x* # *xs*) − {*x*}))
proof (*rule is-minimal-basis-replace*, *simp*)
  from *assms*(*2*) **have** *eq*: *set* (*x* # *xs*) − {*x*} = *set xs* **by** *simp*
    show (*red* (*set* (*x* # *xs*) − {*x*}))** *x* (*trd xs x*) **unfolding** *eq* **by** (*rule trd-red-rtrancl*)
qed
also from *assms*(*2*) **have** ... = *set* ((*trd xs x*) # *xs*) **by** *auto*
finally show *?thesis* **.**
qed

lemma *is-minimal-basis-trd-distinct*:

assumes *min*: *is-minimal-basis* (*set* (*x* # *xs*)) **and** *dist*: *distinct* (*x* # *xs*)
shows *distinct* ((*trd xs x*) # *xs*)
proof −
let *?y* = *trd xs x*
from *min* **have** *lty*: *lt ?y* = *lt x*
proof (*rule minimal-basis-red-rtrancl-lt*, *simp*)
from *dist* **have** $x \notin$ *set xs* **by** *simp*
hence *eq*: *set* (*x* # *xs*) − {*x*} = *set xs* **by** *simp*
show (*red* (*set* (*x* # *xs*) − {*x*}))** *x* (*trd xs x*) **unfolding** *eq* **by** (*rule trd-red-rtrancl*)
qed
have *?y* $\notin$ *set xs*
proof
assume *?y* $\in$ *set xs*
hence *?y* $\in$ *set* (*x* # *xs*) **by** *simp*
with *min* **have** ¬ *lt ?y* $adds_t$ *lt x*
proof (*rule is-minimal-basisD2*, *simp*)
show *?y* $\neq$ *x*
proof
assume *?y* = *x*
from *dist* **have** *x* $\notin$ *set xs* **by** *simp*
with ‹*?y* $\in$ *set xs*› **show** *False* **unfolding** ‹*?y* = *x*› **by** *simp*
qed
qed
thus *False* **unfolding** *lty* **by** (*simp add*: *adds-term-refl*)
qed
moreover from *dist* **have** *distinct xs* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

primrec *comp-red-basis-aux* :: ('*t* $\Rightarrow_0$ '*b*) *list* $\Rightarrow$ ('*t* $\Rightarrow_0$ '*b*) *list* $\Rightarrow$ ('*t* $\Rightarrow_0$ '*b*::*field*) *list* **where**
comp-red-basis-aux-base: *comp-red-basis-aux Nil ys* = *ys*|
comp-red-basis-aux-rec: *comp-red-basis-aux* (*x* # *xs*) *ys* = *comp-red-basis-aux xs* ((*trd* (*xs* @ *ys*) *x*) # *ys*)

lemma *subset-comp-red-basis-aux*: *set ys* $\subseteq$ *set* (*comp-red-basis-aux xs ys*)
proof (*induct xs arbitrary*: *ys*)
case *Nil*
show *?case* **unfolding** *comp-red-basis-aux-base* **..**
next
case (*Cons a xs*)
have *set ys* $\subseteq$ *set* ((*trd* (*xs* @ *ys*) *a*) # *ys*) **by** *auto*
also have ... $\subseteq$ *set* (*comp-red-basis-aux xs* ((*trd* (*xs* @ *ys*) *a*) # *ys*)) **by** (*rule Cons.hyps*)
finally show *?case* **unfolding** *comp-red-basis-aux-rec* **.**
qed

lemma *comp-red-basis-aux-nonzero*:

```
  assumes is-minimal-basis (set (xs @ ys)) and distinct (xs @ ys) and p ∈ set
(comp-red-basis-aux xs ys)
  shows p ≠ 0
  using assms
proof (induct xs arbitrary: ys)
  case Nil
  show ?case
  proof (rule is-minimal-basisD1)
    from Nil(1) show is-minimal-basis (set ys) by simp
  next
    from Nil(3) show p ∈ set ys unfolding comp-red-basis-aux-base .
  qed
next
  case (Cons a xs)
  have eq: (a # xs) @ ys = a # (xs @ ys) by simp
  have a ∈ set (a # xs @ ys) by simp
  from Cons(3) have a ∉ set (xs @ ys) unfolding eq by simp
  let ?ys = trd (xs @ ys) a # ys
  show ?case
  proof (rule Cons.hyps)
    from Cons(3) have a ∉ set (xs @ ys) unfolding eq by simp
     with Cons(2) show is-minimal-basis (set (xs @ ?ys)) unfolding set-reorder
eq
      by (rule is-minimal-basis-trd-is-minimal-basis)
  next
    from Cons(2) Cons(3) show distinct (xs @ ?ys) unfolding distinct-reorder eq
      by (rule is-minimal-basis-trd-distinct)
  next
   from Cons(4) show p ∈ set (comp-red-basis-aux xs ?ys) unfolding comp-red-basis-aux-rec
.
  qed
qed

lemma comp-red-basis-aux-lt:
  assumes is-minimal-basis (set (xs @ ys)) and distinct (xs @ ys)
  shows lt ‘ set (xs @ ys) = lt ‘ set (comp-red-basis-aux xs ys)
  using assms
proof (induct xs arbitrary: ys)
  case Nil
  show ?case unfolding comp-red-basis-aux-base by simp
next
  case (Cons a xs)
  have eq: (a # xs) @ ys = a # (xs @ ys) by simp
  from Cons(3) have a: a ∉ set (xs @ ys) unfolding eq by simp
  let ?b = trd (xs @ ys) a
  let ?ys = ?b # ys
  from Cons(2) have lt ?b = lt a unfolding eq
  proof (rule minimal-basis-red-rtrancl-lt, simp)
    from a have eq2: set (a # xs @ ys) − {a} = set (xs @ ys) by simp
```

show (*red* (*set* (*a* # *xs* @ *ys*) − {*a*}))** *a* ?*b* **unfolding** *eq2* **by** (*rule trd-red-rtrancl*)
qed
hence *lt* ' *set* ((*a* # *xs*) @ *ys*) = *lt* ' *set* ((?*b* # *xs*) @ *ys*) **by** *simp*
also have ... = *lt* ' *set* (*xs* @ (?*b* # *ys*)) **by** *simp*
finally have *eq2*: *lt* ' *set* ((*a* # *xs*) @ *ys*) = *lt* ' *set* (*xs* @ (?*b* # *ys*)) **.**
show ?*case* **unfolding** *comp-red-basis-aux-rec eq2*
proof (*rule Cons.hyps*)
from *Cons*(*3*) **have** *a* ∉ *set* (*xs* @ *ys*) **unfolding** *eq* **by** *simp*
with *Cons*(*2*) **show** *is-minimal-basis* (*set* (*xs* @ ?*ys*)) **unfolding** *set-reorder eq*
by (*rule is-minimal-basis-trd-is-minimal-basis*)
next
from *Cons*(*2*) *Cons*(*3*) **show** *distinct* (*xs* @ ?*ys*) **unfolding** *distinct-reorder eq*
by (*rule is-minimal-basis-trd-distinct*)
qed
qed

lemma *comp-red-basis-aux-pmdl*:
assumes *is-minimal-basis* (*set* (*xs* @ *ys*)) **and** *distinct* (*xs* @ *ys*)
shows *pmdl* (*set* (*comp-red-basis-aux xs ys*)) ⊆ *pmdl* (*set* (*xs* @ *ys*))
using *assms*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
show ?*case* **unfolding** *comp-red-basis-aux-base* **by** *simp*
next
case (*Cons a xs*)
have *eq*: (*a* # *xs*) @ *ys* = *a* # (*xs* @ *ys*) **by** *simp*
from *Cons*(*3*) **have** *a*: *a* ∉ *set* (*xs* @ *ys*) **unfolding** *eq* **by** *simp*
let ?*b* = *trd* (*xs* @ *ys*) *a*
let ?*ys* = ?*b* # *ys*
have *pmdl* (*set* (*comp-red-basis-aux xs* ?*ys*)) ⊆ *pmdl* (*set* (*xs* @ ?*ys*))
proof (*rule Cons.hyps*)
from *Cons*(*3*) **have** *a* ∉ *set* (*xs* @ *ys*) **unfolding** *eq* **by** *simp*
with *Cons*(*2*) **show** *is-minimal-basis* (*set* (*xs* @ ?*ys*)) **unfolding** *set-reorder eq*
by (*rule is-minimal-basis-trd-is-minimal-basis*)
next
from *Cons*(*2*) *Cons*(*3*) **show** *distinct* (*xs* @ ?*ys*) **unfolding** *distinct-reorder eq*
by (*rule is-minimal-basis-trd-distinct*)
qed
also have ... = *pmdl* (*set* (?*b* # *xs* @ *ys*)) **by** *simp*
also from *a* **have** ... = *pmdl* (*insert* ?*b* (*set* (*a* # *xs* @ *ys*) − {*a*})) **by** *auto*
also have ... ⊆ *pmdl* (*set* (*a* # *xs* @ *ys*))
proof (*rule pmdl.replace-span*)
have *a* − (*trd* (*xs* @ *ys*) *a*) ∈ *pmdl* (*set* (*xs* @ *ys*)) **by** (*rule trd-in-pmdl*)
have *a* − (*trd* (*xs* @ *ys*) *a*) ∈ *pmdl* (*set* (*a* # *xs* @ *ys*))
proof
show *pmdl* (*set* (*xs* @ *ys*)) ⊆ *pmdl* (*set* (*a* # *xs* @ *ys*)) **by** (*rule pmdl.span-mono*)

auto
  qed *fact*
   hence − (*a* − (*trd* (*xs* @ *ys*) *a*)) ∈ *pmdl* (*set* (*a* # *xs* @ *ys*)) **by** (*rule pmdl.span-neg*)
  hence (*trd* (*xs* @ *ys*) *a*) − *a* ∈ *pmdl* (*set* (*a* # *xs* @ *ys*)) **by** *simp*
  hence ((*trd* (*xs* @ *ys*) *a*) − *a*) + *a* ∈ *pmdl* (*set* (*a* # *xs* @ *ys*))
  proof (*rule pmdl.span-add*)
   show *a* ∈ *pmdl* (*set* (*a* # *xs* @ *ys*))
   proof
    show *a* ∈ *set* (*a* # *xs* @ *ys*) **by** *simp*
   qed (*rule pmdl.span-superset*)
  qed
  thus *trd* (*xs* @ *ys*) *a* ∈ *pmdl* (*set* (*a* # *xs* @ *ys*)) **by** *simp*
 qed
 also have ... = *pmdl* (*set* ((*a* # *xs*) @ *ys*)) **by** *simp*
 finally show *?case* **unfolding** *comp-red-basis-aux-rec* **.**
qed

lemma *comp-red-basis-aux-irred*:
 assumes *is-minimal-basis* (*set* (*xs* @ *ys*)) **and** *distinct* (*xs* @ *ys*)
  and ⋀*y*. *y* ∈ *set ys* ⟹ ¬ *is-red* (*set* (*xs* @ *ys*) − {*y*}) *y*
  and *p* ∈ *set* (*comp-red-basis-aux xs ys*)
 shows ¬ *is-red* (*set* (*comp-red-basis-aux xs ys*) − {*p*}) *p*
 using *assms*
proof (*induct xs arbitrary*: *ys*)
 case *Nil*
 have ¬ *is-red* (*set* ([] @ *ys*) − {*p*}) *p*
 proof (*rule Nil*(*3*))
  from *Nil*(*4*) **show** *p* ∈ *set ys* **unfolding** *comp-red-basis-aux-base* **.**
 qed
 thus *?case* **unfolding** *comp-red-basis-aux-base* **by** *simp*
next
 case (*Cons a xs*)
 have *eq*: (*a* # *xs*) @ *ys* = *a* # (*xs* @ *ys*) **by** *simp*
 from *Cons*(*3*) **have** *a-notin*: *a* ∉ *set* (*xs* @ *ys*) **unfolding** *eq* **by** *simp*
 from *Cons*(*2*) **have** *is-min*: *is-minimal-basis* (*set* (*a* # *xs* @ *ys*)) **unfolding** *eq*
.
 let *?b* = *trd* (*xs* @ *ys*) *a*
 let *?ys* = *?b* # *ys*
 have *dist*: *distinct* (*?b* # (*xs* @ *ys*))
 proof (*rule is-minimal-basis-trd-distinct*, *fact is-min*)
  from *Cons*(*3*) **show** *distinct* (*a* # *xs* @ *ys*) **unfolding** *eq* **.**
 qed

 show *?case* **unfolding** *comp-red-basis-aux-rec*
 proof (*rule Cons.hyps*)
   from *Cons*(*2*) *a-notin* **show** *is-minimal-basis* (*set* (*xs* @ *?ys*)) **unfolding** *set-reorder eq*
   by (*rule is-minimal-basis-trd-is-minimal-basis*)

```
  next
    from dist show distinct (xs @ ?ys) unfolding distinct-reorder .
  next
    fix y
    assume y ∈ set ?ys
    hence y = ?b ∨ y ∈ set ys by simp
    thus ¬ is-red (set (xs @ ?ys) − {y}) y
    proof
      assume y = ?b
      from dist have ?b ∉ set (xs @ ys) by simp
      hence eq3: set (xs @ ?ys) − {?b} = set (xs @ ys) unfolding set-reorder by
simp
      have ¬ is-red (set (xs @ ys)) ?b by (rule trd-irred)
      thus ?thesis unfolding ‹y = ?b› eq3 .
    next
      assume y ∈ set ys
      hence irred: ¬ is-red (set ((a # xs) @ ys) − {y}) y by (rule Cons(4))
      from ‹y ∈ set ys› a-notin have y ≠ a by auto
      hence eq3: set ((a # xs) @ ys) − {y} = {a} ∪ (set (xs @ ys) − {y}) by auto
      from irred have i1: ¬ is-red {a} y and i2: ¬ is-red (set (xs @ ys) − {y}) y
        unfolding eq3 is-red-union by simp-all
      show ?thesis unfolding set-reorder
      proof (cases y = ?b)
        case True
        from i2 show ¬ is-red (set (?b # xs @ ys) − {y}) y by (simp add: True)
      next
        case False
        hence eq4: set (?b # xs @ ys) − {y} = {?b} ∪ (set (xs @ ys) − {y}) by
auto
        show ¬ is-red (set (?b # xs @ ys) − {y}) y unfolding eq4
        proof
          assume is-red ({?b} ∪ (set (xs @ ys) − {y})) y
          thus False unfolding is-red-union
          proof
            have ltb: lt ?b = lt a
            proof (rule minimal-basis-red-rtrancl-lt, fact is-min)
              show a ∈ set (a # xs @ ys) by simp
            next
              from a-notin have eq: set (a # xs @ ys) − {a} = set (xs @ ys) by
simp
              show (red (set (a # xs @ ys) − {a}))** a ?b unfolding eq by (rule
trd-red-rtrancl)
            qed
            assume is-red {?b} y
            then obtain t where t ∈ keys y and lt ?b adds_t t unfolding is-red-adds-iff
by auto
            with ltb have lt a adds_t t by simp
            have is-red {a} y
              by (rule is-red-addsI, rule, rule is-minimal-basisD1, fact is-min, simp,
```

```
fact+)
          with i1 show False ..
        next
          assume is-red (set (xs @ ys) − {y}) y
          with i2 show False ..
        qed
      qed
    qed
  qed
next
  from Cons(5) show p ∈ set (comp-red-basis-aux xs ?ys) unfolding comp-red-basis-aux-rec
.
qed
qed

lemma comp-red-basis-aux-dgrad-p-set-le:
  assumes dickson-grading d
  shows dgrad-p-set-le d (set (comp-red-basis-aux xs ys)) (set xs ∪ set ys)
proof (induct xs arbitrary: ys)
  case Nil
  show ?case by (simp, rule dgrad-p-set-le-subset, fact subset-refl)
next
  case (Cons x xs)
  let ?h = trd (xs @ ys) x
  have dgrad-p-set-le d (set (comp-red-basis-aux xs (?h # ys))) (set xs ∪ set (?h
# ys))
    by (fact Cons)
  also have ... = insert ?h (set xs ∪ set ys) by simp
  also have dgrad-p-set-le d ... (insert x (set xs ∪ set ys))
  proof (rule dgrad-p-set-leI-insert)
    show dgrad-p-set-le d (set xs ∪ set ys) (insert x (set xs ∪ set ys))
      by (rule dgrad-p-set-le-subset, blast)
  next
    have (red (set (xs @ ys)))** x ?h by (rule trd-red-rtrancl)
    with assms have dgrad-p-set-le d {?h} (insert x (set (xs @ ys)))
      by (rule dgrad-p-set-le-red-rtrancl)
    thus dgrad-p-set-le d {?h} (insert x (set xs ∪ set ys)) by simp
  qed
  finally show ?case by simp
qed

definition comp-red-basis :: ('t ⇒0 'b) list ⇒ ('t ⇒0 'b::field) list
  where comp-red-basis xs = comp-red-basis-aux (comp-min-basis xs) []

lemma comp-red-basis-nonzero:
  assumes p ∈ set (comp-red-basis xs)
  shows p ≠ 0
proof −
 have is-minimal-basis (set ((comp-min-basis xs) @ [])) by (simp add: comp-min-basis-is-minimal-basis)
```

moreover have *distinct* ((*comp-min-basis xs*) @ []) **by** (*simp add*: *comp-min-basis-distinct*)
moreover from *assms* **have** *p* ∈ *set* (*comp-red-basis-aux* (*comp-min-basis xs*) []) **unfolding** *comp-red-basis-def* **.**
ultimately show *?thesis* **by** (*rule comp-red-basis-aux-nonzero*)
qed

lemma *pmdl-comp-red-basis-subset*: *pmdl* (*set* (*comp-red-basis xs*)) ⊆ *pmdl* (*set xs*)
proof
fix *f*
assume *fin*: *f* ∈ *pmdl* (*set* (*comp-red-basis xs*))
have *f* ∈ *pmdl* (*set* (*comp-min-basis xs*))
proof
from *fin* **show** *f* ∈ *pmdl* (*set* (*comp-red-basis-aux* (*comp-min-basis xs*) []))
unfolding *comp-red-basis-def* **.**
next
have *pmdl* (*set* (*comp-red-basis-aux* (*comp-min-basis xs*) [])) ⊆ *pmdl* (*set* ((*comp-min-basis xs*) @ []))
by (*rule comp-red-basis-aux-pmdl*, *simp-all*, *rule comp-min-basis-is-minimal-basis*, *rule comp-min-basis-distinct*)
thus *pmdl* (*set* (*comp-red-basis-aux* (*comp-min-basis xs*) [])) ⊆ *pmdl* (*set* (*comp-min-basis xs*))
by *simp*
qed
also from *comp-min-basis-subset* **have** ... ⊆ *pmdl* (*set xs*) **by** (*rule pmdl.span-mono*)
finally show *f* ∈ *pmdl* (*set xs*) **.**
qed

lemma *comp-red-basis-adds*:
assumes *p* ∈ *set xs* **and** *p* ≠ *0*
obtains *q* **where** *q* ∈ *set* (*comp-red-basis xs*) **and** *lt q* $adds_t$ *lt p*
proof −
from *assms* **obtain** *q1* **where** *q1* ∈ *set* (*comp-min-basis xs*) **and** *lt q1* $adds_t$ *lt p*
by (*rule comp-min-basis-adds*)
from ‹*q1* ∈ *set* (*comp-min-basis xs*)› **have** *lt q1* ∈ *lt* ‘ *set* (*comp-min-basis xs*) **by** *simp*
also have ... = *lt* ‘ *set* ((*comp-min-basis xs*) @ []) **by** *simp*
also have ... = *lt* ‘ *set* (*comp-red-basis-aux* (*comp-min-basis xs*) [])
by (*rule comp-red-basis-aux-lt*, *simp-all*, *rule comp-min-basis-is-minimal-basis*, *rule comp-min-basis-distinct*)
finally obtain *q* **where** *q* ∈ *set* (*comp-red-basis-aux* (*comp-min-basis xs*) []) **and** *lt q* = *lt q1*
by *auto*
show *?thesis*
proof
show *q* ∈ *set* (*comp-red-basis xs*) **unfolding** *comp-red-basis-def* **by** *fact*
next
from ‹*lt q1* $adds_t$ *lt p*› **show** *lt q* $adds_t$ *lt p* **unfolding** ‹*lt q* = *lt q1*› **.**
qed

qed

lemma *comp-red-basis-lt*:
  assumes $p \in$ *set* (*comp-red-basis xs*)
  obtains *q* **where** $q \in$ *set xs* **and** $q \neq 0$ **and** *lt q* = *lt p*
proof −
  have *eq*: *lt* ' *set* ((*comp-min-basis xs*) @ []) = *lt* ' *set* (*comp-red-basis-aux* (*comp-min-basis xs*) [])
    by (*rule comp-red-basis-aux-lt*, *simp-all*, *rule comp-min-basis-is-minimal-basis*, *rule comp-min-basis-distinct*)
  from *assms* **have** *lt* $p \in$ *lt* ' *set* (*comp-red-basis xs*) **by** *simp*
  also have ... = *lt* ' *set* (*comp-red-basis-aux* (*comp-min-basis xs*) []) **unfolding** *comp-red-basis-def* **..**
  also have ... = *lt* ' *set* (*comp-min-basis xs*) **unfolding** *eq*[*symmetric*] **by** *simp*
  finally obtain *q* **where** $q \in$ *set* (*comp-min-basis xs*) **and** *lt q* = *lt p* **by** *auto*
  show *?thesis*
  proof
    show $q \in$ *set xs* **by** (*rule*, *fact*, *rule comp-min-basis-subset*)
  next
    show $q \neq 0$ **by** (*rule comp-min-basis-nonzero*, *fact*)
  qed *fact*
qed

lemma *comp-red-basis-is-red*: *is-red* (*set* (*comp-red-basis xs*)) *f* ⟷ *is-red* (*set xs*) *f*
proof
  assume *is-red* (*set* (*comp-red-basis xs*)) *f*
  then obtain *x t* **where** $x \in$ *set* (*comp-red-basis xs*) **and** $t \in$ *keys f* **and** $x \neq 0$ **and** *lt x* $adds_t$ *t*
    by (*rule is-red-addsE*)
  from ‹$x \in$ *set* (*comp-red-basis xs*)› **obtain** *y* **where** *yin*: $y \in$ *set xs* **and** $y \neq 0$ **and** *lt y* = *lt x*
    by (*rule comp-red-basis-lt*)
  show *is-red* (*set xs*) *f*
  proof (*rule is-red-addsI*)
    from ‹*lt x* $adds_t$ *t*› **show** *lt y* $adds_t$ *t* **unfolding** ‹*lt y* = *lt x*› **.**
  qed *fact+*
next
  assume *is-red* (*set xs*) *f*
  then obtain *x t* **where** $x \in$ *set xs* **and** $t \in$ *keys f* **and** $x \neq 0$ **and** *lt x* $adds_t$ *t*
    by (*rule is-red-addsE*)
  from ‹$x \in$ *set xs*› ‹$x \neq 0$› **obtain** *y* **where** *yin*: $y \in$ *set* (*comp-red-basis xs*) **and** *lt y* $adds_t$ *lt x*
    by (*rule comp-red-basis-adds*)
  show *is-red* (*set* (*comp-red-basis xs*)) *f*
  proof (*rule is-red-addsI*)
    from ‹*lt y* $adds_t$ *lt x*› ‹*lt x* $adds_t$ *t*› **show** *lt y* $adds_t$ *t* **by** (*rule adds-term-trans*)
  next
    from *yin* **show** $y \neq 0$ **by** (*rule comp-red-basis-nonzero*)

qed *fact+*
qed

lemma *comp-red-basis-is-auto-reduced*: *is-auto-reduced* (*set* (*comp-red-basis xs*))
unfolding *is-auto-reduced-def remove-def*
proof (*intro ballI*)
fix *x*
assume *xin*: *x* ∈ *set* (*comp-red-basis xs*)
show ¬ *is-red* (*set* (*comp-red-basis xs*) − {*x*}) *x* **unfolding** *comp-red-basis-def*
proof (*rule comp-red-basis-aux-irred*, *simp-all*, *rule comp-min-basis-is-minimal-basis*, *rule comp-min-basis-distinct*)
from *xin* **show** *x* ∈ *set* (*comp-red-basis-aux* (*comp-min-basis xs*) []) **unfolding** *comp-red-basis-def* **.**
qed
qed

lemma *comp-red-basis-dgrad-p-set-le*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* (*set* (*comp-red-basis xs*)) (*set xs*)
proof −
have *dgrad-p-set-le d* (*set* (*comp-red-basis xs*)) (*set* (*comp-min-basis xs*) ∪ *set* [])
unfolding *comp-red-basis-def* **using** *assms* **by** (*rule comp-red-basis-aux-dgrad-p-set-le*)
also have ... = *set* (*comp-min-basis xs*) **by** *simp*
also from *comp-min-basis-subset* **have** *dgrad-p-set-le d* ... (*set xs*)
by (*rule dgrad-p-set-le-subset*)
finally show *?thesis* **.**
qed

12.5 Auto-Reduction and Monicity

definition *comp-red-monic-basis* :: ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t \Rightarrow_0 'b$::*field*) *list* **where**
comp-red-monic-basis xs = *map monic* (*comp-red-basis xs*)

lemma *set-comp-red-monic-basis*: *set* (*comp-red-monic-basis xs*) = *monic* ‘ (*set* (*comp-red-basis xs*))
by (*simp add*: *comp-red-monic-basis-def*)

lemma *comp-red-monic-basis-nonzero*:
assumes *p* ∈ *set* (*comp-red-monic-basis xs*)
shows *p* ≠ *0*
proof −
from *assms* **obtain** *p'* **where** *p-def*: *p* = *monic p'* **and** *p'*: *p'* ∈ *set* (*comp-red-basis xs*)
unfolding *set-comp-red-monic-basis* **..**
from *p'* **have** *p'* ≠ *0* **by** (*rule comp-red-basis-nonzero*)
thus *?thesis* **unfolding** *p-def monic-0-iff* **.**
qed

lemma *comp-red-monic-basis-is-monic-set*: *is-monic-set* (*set* (*comp-red-monic-basis*

xs))
unfolding *set-comp-red-monic-basis* **by** (*rule image-monic-is-monic-set*)

lemma *pmdl-comp-red-monic-basis-subset*: *pmdl* (*set* (*comp-red-monic-basis xs*)) ⊆ *pmdl* (*set xs*)
unfolding *set-comp-red-monic-basis pmdl-image-monic* **by** (*fact pmdl-comp-red-basis-subset*)

lemma *comp-red-monic-basis-is-auto-reduced*: *is-auto-reduced* (*set* (*comp-red-monic-basis xs*))
unfolding *set-comp-red-monic-basis* **by** (*rule image-monic-is-auto-reduced*, *rule comp-red-basis-is-auto-reduced*)

lemma *comp-red-monic-basis-dgrad-p-set-le*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* (*set* (*comp-red-monic-basis xs*)) (*set xs*)
proof −
have *dgrad-p-set-le d* (*monic* ‘ (*set* (*comp-red-basis xs*))) (*set* (*comp-red-basis xs*))
by (*simp add*: *dgrad-p-set-le-def*, *fact dgrad-set-le-refl*)
also from *assms* **have** *dgrad-p-set-le d* ... (*set xs*) **by** (*rule comp-red-basis-dgrad-p-set-le*)
finally show *?thesis* **by** (*simp add*: *set-comp-red-monic-basis*)
qed

end

end

13 Reduced Gröbner Bases

theory *Reduced-GB*
imports *Groebner-Bases Auto-Reduction*
begin

lemma (**in** *gd-term*) *GB-image-monic*: *is-Groebner-basis* (*monic* ‘ *G*) ⟷ *is-Groebner-basis G*
by (*simp add*: *GB-alt-1*)

13.1 Definition and Uniqueness of Reduced Gröbner Bases

context *ordered-term*
begin

definition *is-reduced-GB* :: ($'t \Rightarrow_0$ *'b::field*) *set* ⇒ *bool* **where**
is-reduced-GB B ≡ *is-Groebner-basis B* ∧ *is-auto-reduced B* ∧ *is-monic-set B* ∧ *0* ∉ *B*

lemma *reduced-GB-D1*:
assumes *is-reduced-GB G*
shows *is-Groebner-basis G*
using *assms* **unfolding** *is-reduced-GB-def* **by** *simp*

lemma *reduced-GB-D2*:
assumes *is-reduced-GB G*
shows *is-auto-reduced G*
using *assms* **unfolding** *is-reduced-GB-def* **by** *simp*

lemma *reduced-GB-D3*:
assumes *is-reduced-GB G*
shows *is-monic-set G*
using *assms* **unfolding** *is-reduced-GB-def* **by** *simp*

lemma *reduced-GB-D4*:
assumes *is-reduced-GB G* **and** $g \in G$
shows $g \neq 0$
using *assms* **unfolding** *is-reduced-GB-def* **by** *auto*

lemma *reduced-GB-lc*:
assumes *major*: *is-reduced-GB G* **and** $g \in G$
shows *lc* $g = 1$
by (*rule is-monic-setD*, *rule reduced-GB-D3*, *fact major*, *fact* ‹$g \in G$›, *rule reduced-GB-D4*, *fact major*, *fact* ‹$g \in G$›)

end

context *gd-term*
begin

lemma *is-reduced-GB-subsetI*:
assumes *Ared*: *is-reduced-GB A* **and** *BGB*: *is-Groebner-basis B* **and** *Bmon*: *is-monic-set B*
and ∗: $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a - b \neq 0 \Longrightarrow lt\ (a - b) \in keys\ b \Longrightarrow lt\ (a - b) \prec_t lt\ b \Longrightarrow False$
and *id-eq*: *pmdl* $A =$ *pmdl* B
shows $A \subseteq B$
proof
fix *a*
assume $a \in A$

have $a \neq 0$ **by** (*rule reduced-GB-D4*, *fact Ared*, *fact* ‹$a \in A$›)
have *lca*: *lc* $a = 1$ **by** (*rule reduced-GB-lc*, *fact Ared*, *fact* ‹$a \in A$›)
have *AGB*: *is-Groebner-basis A* **by** (*rule reduced-GB-D1*, *fact Ared*)

from ‹$a \in A$› **have** $a \in$ *pmdl A* **by** (*rule pmdl.span-base*)
also have ... = *pmdl B* **using** *id-eq* **by** *simp*
finally have $a \in$ *pmdl B* **.**

from *BGB this* ‹$a \neq 0$› **obtain** *b* **where** $b \in B$ **and** $b \neq 0$ **and** *baddsa*: $lt\ b\ adds_t\ lt\ a$
by (*rule GB-adds-lt*)

from *Bmon this*(*1*) *this*(*2*) **have** *lcb*: *lc b* = *1* **by** (*rule is-monic-setD*)
from ‹*b* ∈ *B*› **have** *b* ∈ *pmdl B* **by** (*rule pmdl.span-base*)
also have ... = *pmdl A* **using** *id-eq* **by** *simp*
finally have *b* ∈ *pmdl A* **.**

have *lt-eq*: *lt b* = *lt a*
proof (*rule ccontr*)
  assume *lt b* ≠ *lt a*
  from *AGB* ‹*b* ∈ *pmdl A*› ‹*b* ≠ *0*› **obtain** a'
    where a' ∈ *A* **and** a' ≠ *0* **and** a'*addsb*: *lt* a' $adds_t$ *lt b* **by** (*rule GB-adds-lt*)
  have a'*addsa*: *lt* a' $adds_t$ *lt a* **by** (*rule adds-term-trans*, *fact* a'*addsb*, *fact baddsa*)
  have *lt* a' ≠ *lt a*
  proof
    assume *lt* a' = *lt a*
    hence *aaddsa*$'$: *lt a* $adds_t$ *lt* a' **by** (*simp add*: *adds-term-refl*)
    have *lt a* $adds_t$ *lt b* **by** (*rule adds-term-trans*, *fact aaddsa*$'$, *fact* a'*addsb*)
    have *lt a* = *lt b* **by** (*rule adds-term-antisym*, *fact*+)
    with ‹*lt b* ≠ *lt a*› **show** *False* **by** *simp*
  qed
  hence a' ≠ *a* **by** *auto*
  with ‹a' ∈ *A*› **have** a' ∈ *A* − {*a*} **by** *blast*
  have *is-red*: *is-red* (*A* − {*a*}) *a* **by** (*intro is-red-addsI*, *fact*, *fact*, *rule lt-in-keys*, *fact*+)
  have ¬ *is-red* (*A* − {*a*}) *a* **by** (*rule is-auto-reducedD*, *rule reduced-GB-D2*, *fact Ared*, *fact*+)
  from *this is-red* **show** *False* **..**
qed

have *a* − *b* = *0*
proof (*rule ccontr*)
  let *?c* = *a* − *b*
  assume *?c* ≠ *0*
  have *?c* ∈ *pmdl A* **by** (*rule pmdl.span-diff*, *fact*+)
  also have ... = *pmdl B* **using** *id-eq* **by** *simp*
  finally have *?c* ∈ *pmdl B* **.**

  from ‹*b* ≠ *0*› **have** − *b* ≠ *0* **by** *simp*
  have *lt* (−*b*) = *lt a* **unfolding** *lt-uminus* **by** *fact*
  have *lc* (−*b*) = − *lc a* **unfolding** *lc-uminus lca lcb* **..**
  from ‹*?c* ≠ *0*› **have** *a* + (−*b*) ≠ *0* **by** *simp*

  have *lt ?c* ∈ *keys ?c* **by** (*rule lt-in-keys*, *fact*)
  have *keys ?c* ⊆ (*keys a* ∪ *keys b*) **by** (*fact keys-minus*)
  with ‹*lt ?c* ∈ *keys ?c*› **have** *lt ?c* ∈ *keys a* ∨ *lt ?c* ∈ *keys b* **by** *auto*
  thus *False*
  proof
    assume *lt ?c* ∈ *keys a*

    from *AGB* ‹*?c* ∈ *pmdl A*› ‹*?c* ≠ *0*› **obtain** a'

where $a' \in A$ **and** $a' \neq 0$ **and** $a'addsc$: $lt\ a'\ adds_t\ lt\ ?c$ **by** (*rule GB-adds-lt*)

from $a'addsc$ **have** $lt\ a' \preceq_t lt\ ?c$ **by** (*rule ord-adds-term*)
also have ... $= lt\ (a + (-\ b))$ **by** *simp*
also have ... $\prec_t lt\ a$ **by** (*rule lt-plus-lessI*, *fact+*)
finally have $lt\ a' \prec_t lt\ a$ **.**
hence $lt\ a' \neq lt\ a$ **by** *simp*
hence $a' \neq a$ **by** *auto*
with ‹$a' \in A$› **have** $a' \in A - \{a\}$ **by** *blast*

have *is-red*: *is-red* $(A - \{a\})\ a$ **by** (*intro is-red-addsI*, *fact*, *fact*, *fact+*)
have ¬ *is-red* $(A - \{a\})\ a$ **by** (*rule is-auto-reducedD*, *rule reduced-GB-D2*, *fact Ared*, *fact+*)
from *this is-red* **show** *False* **..**
next
assume $lt\ ?c \in keys\ b$

with ‹$a \in A$› ‹$b \in B$› ‹$a \neq 0$› ‹$b \neq 0$› ‹$?c \neq 0$› **show** *False*
proof (*rule* ∗)
have $lt\ ?c = lt\ ((-\ b) + a)$ **by** *simp*
also have ... $\prec_t lt\ (-b)$
proof (*rule lt-plus-lessI*)
from ‹$?c \neq 0$› **show** $-b + a \neq 0$ **by** *simp*
next
from ‹$lt\ (-b) = lt\ a$› **show** $lt\ a = lt\ (-b)$ **by** *simp*
next
from ‹$lc\ (-b) = -\ lc\ a$› **show** $lc\ a = -\ lc\ (-b)$ **by** *simp*
qed
finally show $lt\ ?c \prec_t lt\ b$ **unfolding** *lt-uminus* **.**
qed
qed
qed

hence $a = b$ **by** *simp*
with ‹$b \in B$› **show** $a \in B$ **by** *simp*
qed

lemma *is-reduced-GB-unique′*:
assumes *Ared*: *is-reduced-GB A* **and** *Bred*: *is-reduced-GB B* **and** *id-eq*: *pmdl A* = *pmdl B*
shows $A \subseteq B$
proof −
from *Bred* **have** *BGB*: *is-Groebner-basis B* **by** (*rule reduced-GB-D1*)
with *assms*(*1*) **show** *?thesis*
proof (*rule is-reduced-GB-subsetI*)
from *Bred* **show** *is-monic-set B* **by** (*rule reduced-GB-D3*)
next
fix $a\ b :: 't \Rightarrow_0 'b$
let $?c = a - b$

assume $a \in A$ **and** $b \in B$ **and** $a \neq 0$ **and** $b \neq 0$ **and** $?c \neq 0$ **and** $lt\ ?c \in keys\ b$ **and** $lt\ ?c \prec_t lt\ b$

from ‹$a \in A$› **have** $a \in pmdl\ B$ **by** (*simp only*: *id-eq*[*symmetric*], *rule pmdl.span-base*)
moreover from ‹$b \in B$› **have** $b \in pmdl\ B$ **by** (*rule pmdl.span-base*)
ultimately have $?c \in pmdl\ B$ **by** (*rule pmdl.span-diff*)
from *BGB this* ‹$?c \neq 0$› **obtain** b'
where $b' \in B$ **and** $b' \neq 0$ **and** $b'addsc$: $lt\ b'\ adds_t\ lt\ ?c$ **by** (*rule GB-adds-lt*)

from $b'addsc$ **have** $lt\ b' \preceq_t lt\ ?c$ **by** (*rule ord-adds-term*)
also have ... $\prec_t lt\ b$ **by** *fact*
finally have $lt\ b' \prec_t lt\ b$ **unfolding** *lt-uminus* **.**
hence $lt\ b' \neq lt\ b$ **by** *simp*
hence $b' \neq b$ **by** *auto*
with ‹$b' \in B$› **have** $b' \in B - \{b\}$ **by** *blast*

have *is-red*: *is-red* $(B - \{b\})\ b$ **by** (*intro is-red-addsI*, *fact*, *fact*, *fact+*)
have ¬ *is-red* $(B - \{b\})\ b$ **by** (*rule is-auto-reducedD*, *rule reduced-GB-D2*, *fact Bred*, *fact+*)
from *this is-red* **show** *False* **..**
qed *fact*
qed

theorem *is-reduced-GB-unique*:
assumes *Ared*: *is-reduced-GB A* **and** *Bred*: *is-reduced-GB B* **and** *id-eq*: *pmdl A* = *pmdl B*
shows $A = B$
proof
from *assms* **show** $A \subseteq B$ **by** (*rule is-reduced-GB-unique'*)
next
from *Bred Ared id-eq*[*symmetric*] **show** $B \subseteq A$ **by** (*rule is-reduced-GB-unique'*)
qed

13.2 Computing Reduced Gröbner Bases by Auto-Reduction

13.2.1 Minimal Bases

lemma *minimal-basis-is-reduced-GB*:
assumes *is-minimal-basis B* **and** *is-monic-set B* **and** *is-reduced-GB G* **and** $G \subseteq B$
and *pmdl B* = *pmdl G*
shows $B = G$
using - *assms*(*3*) *assms*(*5*)
proof (*rule is-reduced-GB-unique*)
from *assms*(*3*) **have** *is-Groebner-basis G* **by** (*rule reduced-GB-D1*)
show *is-reduced-GB B* **unfolding** *is-reduced-GB-def*
proof (*intro conjI*)
show $0 \notin B$
proof
assume $0 \in B$

with *assms*(*1*) **have** $0 \neq (0::'t \Rightarrow_0 'b)$ **by** (*rule is-minimal-basisD1*)
thus *False* **by** *simp*
qed
next
from ‹*is-Groebner-basis G*› *assms*(*4*) *assms*(*5*) **show** *is-Groebner-basis B* **by** (*rule GB-subset*)
next
show *is-auto-reduced B* **unfolding** *is-auto-reduced-def*
proof (*intro ballI notI*)
fix *b*
assume $b \in B$
with *assms*(*1*) **have** $b \neq 0$ **by** (*rule is-minimal-basisD1*)
assume *is-red* $(B - \{b\})$ *b*
then obtain *f* **where** $f \in B - \{b\}$ **and** *is-red* $\{f\}$ *b* **by** (*rule is-red-singletonI*)
from *this*(*1*) **have** $f \in B$ **and** $f \neq b$ **by** *simp-all*

from *assms*(*1*) ‹$f \in B$› **have** $f \neq 0$ **by** (*rule is-minimal-basisD1*)
from ‹$f \in B$› **have** $f \in$ *pmdl B* **by** (*rule pmdl.span-base*)
hence $f \in$ *pmdl G* **by** (*simp only*: *assms*(*5*))
from ‹*is-Groebner-basis G*› *this* ‹$f \neq 0$› **obtain** *g* **where** $g \in G$ **and** $g \neq 0$ **and** *lt g* $adds_t$ *lt f*
by (*rule GB-adds-lt*)
from ‹$g \in G$› ‹$G \subseteq B$› **have** $g \in B$ **..**
have $g = f$
proof (*rule ccontr*)
assume $g \neq f$
with *assms*(*1*) ‹$g \in B$› ‹$f \in B$› **have** ¬ *lt g* $adds_t$ *lt f* **by** (*rule is-minimal-basisD2*)
from *this* ‹*lt g* $adds_t$ *lt f*› **show** *False* **..**
qed
with ‹$g \in G$› **have** $f \in G$ **by** *simp*
with ‹$f \in B - \{b\}$› ‹*is-red* $\{f\}$ *b*› **have** *red*: *is-red* $(G - \{b\})$ *b*
by (*meson Diff-iff is-red-singletonD*)

from ‹$b \in B$› **have** $b \in$ *pmdl B* **by** (*rule pmdl.span-base*)
hence $b \in$ *pmdl G* **by** (*simp only*: *assms*(*5*))
from ‹*is-Groebner-basis G*› *this* ‹$b \neq 0$› **obtain** g' **where** $g' \in G$ **and** $g' \neq 0$ **and** *lt* g' $adds_t$ *lt b*
by (*rule GB-adds-lt*)
from ‹$g' \in G$› ‹$G \subseteq B$› **have** $g' \in B$ **..**
have $g' = b$
proof (*rule ccontr*)
assume $g' \neq b$
with *assms*(*1*) ‹$g' \in B$› ‹$b \in B$› **have** ¬ *lt* g' $adds_t$ *lt b* **by** (*rule is-minimal-basisD2*)
from *this* ‹*lt* g' $adds_t$ *lt b*› **show** *False* **..**
qed
with ‹$g' \in G$› **have** $b \in G$ **by** *simp*

from *assms*(*3*) **have** *is-auto-reduced G* **by** (*rule reduced-GB-D2*)

from *this* ‹$b \in G$› **have** ¬ *is-red* $(G - \{b\})$ *b* **by** (*rule is-auto-reducedD*)
from *this red* **show** *False* **..**
qed
qed *fact*
qed

13.2.2 Computing Minimal Bases

lemma *comp-min-basis-pmdl*:
assumes *is-Groebner-basis* (*set xs*)
shows *pmdl* (*set* (*comp-min-basis xs*)) = *pmdl* (*set xs*) (**is** *pmdl* (*set ?ys*) = -)
using *finite-set*
proof (*rule pmdl-eqI-adds-lt-finite*)
from *comp-min-basis-subset* **show** $*$: *pmdl* (*set ?ys*) $\subseteq$ *pmdl* (*set xs*) **by** (*rule pmdl.span-mono*)
next
fix *f*
assume $f \in$ *pmdl* (*set xs*) **and** $f \neq 0$
with *assms* **obtain** *g* **where** $g \in$ *set xs* **and** $g \neq 0$ **and** *1*: *lt g* $adds_t$ *lt f* **by** (*rule GB-adds-lt*)
from *this*(*1*, *2*) **obtain** g' **where** $g' \in$ *set ?ys* **and** *2*: *lt* g' $adds_t$ *lt g*
by (*rule comp-min-basis-adds*)
note *this*(*1*)
moreover from *this* **have** $g' \neq 0$ **by** (*rule comp-min-basis-nonzero*)
moreover from *2 1* **have** *lt* g' $adds_t$ *lt f* **by** (*rule adds-term-trans*)
ultimately show $\exists\, g \in$ *set ?ys*. $g \neq 0 \wedge$ *lt g* $adds_t$ *lt f* **by** *blast*
qed

lemma *comp-min-basis-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-Groebner-basis* (*set* (*comp-min-basis xs*)) (**is** *is-Groebner-basis* (*set ?ys*))
unfolding *GB-alt-2-finite*[*OF finite-set*]
proof (*intro ballI impI*)
fix *f*
assume $f \in$ *pmdl* (*set ?ys*)
also from *assms* **have** ... = *pmdl* (*set xs*) **by** (*rule comp-min-basis-pmdl*)
finally have $f \in$ *pmdl* (*set xs*) **.**
moreover assume $f \neq 0$
ultimately have *is-red* (*set xs*) *f* **using** *assms* **unfolding** *GB-alt-2-finite*[*OF finite-set*] **by** *blast*
thus *is-red* (*set ?ys*) *f* **by** (*rule comp-min-basis-is-red*)
qed

13.2.3 Computing Reduced Bases

lemma *comp-red-basis-pmdl*:
assumes *is-Groebner-basis* (*set xs*)
shows *pmdl* (*set* (*comp-red-basis xs*)) = *pmdl* (*set xs*)
proof (*rule*, *fact pmdl-comp-red-basis-subset*, *rule*)
fix *f*

assume *f* ∈ *pmdl* (*set xs*)
show *f* ∈ *pmdl* (*set* (*comp-red-basis xs*))
proof (*cases f* = *0*)
case *True*
show *?thesis* **unfolding** *True* **by** (*rule pmdl.span-zero*)
next
case *False*
let *?xs* = *comp-red-basis xs*
have (*red* (*set ?xs*))** *f 0*
proof (*rule is-red-implies-0-red-finite*, *fact finite-set*, *fact pmdl-comp-red-basis-subset*)
fix *q*
assume *q* ≠ *0* **and** *q* ∈ *pmdl* (*set xs*)
with *assms* **have** *is-red* (*set xs*) *q* **by** (*rule GB-imp-reducibility*)
thus *is-red* (*set* (*comp-red-basis xs*)) *q* **unfolding** *comp-red-basis-is-red* **.**
qed *fact*
thus *?thesis* **by** (*rule red-rtranclp-0-in-pmdl*)
qed
qed

lemma *comp-red-basis-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-Groebner-basis* (*set* (*comp-red-basis xs*))
unfolding *GB-alt-2-finite*[*OF finite-set*]
proof (*intro ballI impI*)
fix *f*
assume *fin*: *f* ∈ *pmdl* (*set* (*comp-red-basis xs*))
hence *f* ∈ *pmdl* (*set xs*) **unfolding** *comp-red-basis-pmdl*[*OF assms*] **.**
assume *f* ≠ *0*
from *assms* ‹*f* ≠ *0*› ‹*f* ∈ *pmdl* (*set xs*)› **show** *is-red* (*set* (*comp-red-basis xs*)) *f*
by (*simp add*: *comp-red-basis-is-red GB-alt-2-finite*)
qed

13.2.4 Computing Reduced Gröbner Bases

lemma *comp-red-monic-basis-pmdl*:
assumes *is-Groebner-basis* (*set xs*)
shows *pmdl* (*set* (*comp-red-monic-basis xs*)) = *pmdl* (*set xs*)
unfolding *set-comp-red-monic-basis pmdl-image-monic comp-red-basis-pmdl*[*OF assms*] **..**

lemma *comp-red-monic-basis-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-Groebner-basis* (*set* (*comp-red-monic-basis xs*))
unfolding *set-comp-red-monic-basis GB-image-monic* **using** *assms* **by** (*rule comp-red-basis-GB*)

lemma *comp-red-monic-basis-is-reduced-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-reduced-GB* (*set* (*comp-red-monic-basis xs*))
unfolding *is-reduced-GB-def*

proof (*intro conjI*, *rule comp-red-monic-basis-GB*, *fact assms*,
rule comp-red-monic-basis-is-auto-reduced, *rule comp-red-monic-basis-is-monic-set*, *intro notI*)
assume *0* ∈ *set* (*comp-red-monic-basis xs*)
hence *0* ≠ (*0*::*'t* $\Rightarrow_0$ *'b*) **by** (*rule comp-red-monic-basis-nonzero*)
thus *False* **by** *simp*
qed

lemma *ex-finite-reduced-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
obtains *G* **where** *G* ⊆ *dgrad-p-set d m* **and** *finite G* **and** *is-reduced-GB G* **and** *pmdl G* = *pmdl F*
proof −
from *assms* **obtain** *G0* **where** *G0-sub*: *G0* ⊆ *dgrad-p-set d m* **and** *fin*: *finite G0*
and *gb*: *is-Groebner-basis G0* **and** *pid*: *pmdl G0* = *pmdl F*
by (*rule ex-finite-GB-dgrad-p-set*)
from *fin* **obtain** *xs* **where** *set*: *G0* = *set xs* **using** *finite-list* **by** *blast*
let *?G* = *set* (*comp-red-monic-basis xs*)
show *?thesis*
proof
from *assms*(*1*) **have** *dgrad-p-set-le d* (*set* (*comp-red-monic-basis xs*)) *G0* **unfolding** *set*
by (*rule comp-red-monic-basis-dgrad-p-set-le*)
from *this G0-sub* **show** *set* (*comp-red-monic-basis xs*) ⊆ *dgrad-p-set d m*
by (*rule dgrad-p-set-le-dgrad-p-set*)
next
from *gb* **show** *rgb*: *is-reduced-GB ?G* **unfolding** *set*
by (*rule comp-red-monic-basis-is-reduced-GB*)
next
from *gb* **show** *pmdl ?G* = *pmdl F* **unfolding** *set pid*[*symmetric*]
by (*rule comp-red-monic-basis-pmdl*)
qed (*fact finite-set*)
qed

theorem *ex-unique-reduced-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows ∃!*G*. *G* ⊆ *dgrad-p-set d m* ∧ *finite G* ∧ *is-reduced-GB G* ∧ *pmdl G* = *pmdl F*
proof −
from *assms* **obtain** *G* **where** *G* ⊆ *dgrad-p-set d m* **and** *finite G*
and *is-reduced-GB G* **and** *G*: *pmdl G* = *pmdl F* **by** (*rule ex-finite-reduced-GB-dgrad-p-set*)
hence *G* ⊆ *dgrad-p-set d m* ∧ *finite G* ∧ *is-reduced-GB G* ∧ *pmdl G* = *pmdl F* **by** *simp*
thus *?thesis*
proof (*rule ex1I*)
fix *G'*

assume $G' \subseteq$ *dgrad-p-set d m* $\wedge$ *finite* $G' \wedge$ *is-reduced-GB* $G' \wedge$ *pmdl* $G' =$ *pmdl F*
hence *is-reduced-GB* G' **and** G': *pmdl* $G' =$ *pmdl F* **by** *simp-all*
note *this*(*1*) ‹*is-reduced-GB G*›
moreover have *pmdl* $G' =$ *pmdl G* **by** (*simp only*: *G* G')
ultimately show $G' = G$ **by** (*rule is-reduced-GB-unique*)
qed
qed

corollary *ex-unique-reduced-GB-dgrad-p-set*$'$:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows $\exists!G.$ *finite G* $\wedge$ *is-reduced-GB G* $\wedge$ *pmdl G* = *pmdl F*
proof −
from *assms* **obtain** *G* **where** $G \subseteq$ *dgrad-p-set d m* **and** *finite G*
and *is-reduced-GB G* **and** *G*: *pmdl G* = *pmdl F* **by** (*rule ex-finite-reduced-GB-dgrad-p-set*)
hence *finite G* $\wedge$ *is-reduced-GB G* $\wedge$ *pmdl G* = *pmdl F* **by** *simp*
thus *?thesis*
proof (*rule ex1I*)
fix G'
assume *finite* $G' \wedge$ *is-reduced-GB* $G' \wedge$ *pmdl* $G' =$ *pmdl F*
hence *is-reduced-GB* G' **and** G': *pmdl* $G' =$ *pmdl F* **by** *simp-all*
note *this*(*1*) ‹*is-reduced-GB G*›
moreover have *pmdl* $G' =$ *pmdl G* **by** (*simp only*: *G* G')
ultimately show $G' = G$ **by** (*rule is-reduced-GB-unique*)
qed
qed

definition *reduced-GB* :: $('t \Rightarrow_0 'b)$ *set* $\Rightarrow$ $('t \Rightarrow_0 'b$::*field*) *set*
where *reduced-GB B* = (*THE G. finite G* $\wedge$ *is-reduced-GB G* $\wedge$ *pmdl G* = *pmdl B*)

reduced-GB returns the unique reduced Gröbner basis of the given set, provided its Dickson grading is bounded. Combining *comp-red-monic-basis* with any function for computing Gröbner bases, e. g. *gb* from theory "Buchberger", makes *reduced-GB* computable.

lemma *finite-reduced-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *finite* (*reduced-GB F*)
unfolding *reduced-GB-def*
by (*rule the1I2*, *rule ex-unique-reduced-GB-dgrad-p-set*$'$, *fact*, *fact*, *fact*, *elim conjE*)

lemma *reduced-GB-is-reduced-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *is-reduced-GB* (*reduced-GB F*)
unfolding *reduced-GB-def*

by (*rule the1I2*, *rule ex-unique-reduced-GB-dgrad-p-set'*, *fact*, *fact*, *fact*, *elim conjE*)

lemma *reduced-GB-is-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *is-Groebner-basis* (*reduced-GB F*)
proof −
from *assms* **have** *is-reduced-GB* (*reduced-GB F*) **by** (*rule reduced-GB-is-reduced-GB-dgrad-p-set*)
thus *?thesis* **unfolding** *is-reduced-GB-def* **..**
qed

lemma *reduced-GB-is-auto-reduced-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *is-auto-reduced* (*reduced-GB F*)
proof −
from *assms* **have** *is-reduced-GB* (*reduced-GB F*) **by** (*rule reduced-GB-is-reduced-GB-dgrad-p-set*)
thus *?thesis* **unfolding** *is-reduced-GB-def* **by** *simp*
qed

lemma *reduced-GB-is-monic-set-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *is-monic-set* (*reduced-GB F*)
proof −
from *assms* **have** *is-reduced-GB* (*reduced-GB F*) **by** (*rule reduced-GB-is-reduced-GB-dgrad-p-set*)
thus *?thesis* **unfolding** *is-reduced-GB-def* **by** *simp*
qed

lemma *reduced-GB-nonzero-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *0* ∉ *reduced-GB F*
proof −
from *assms* **have** *is-reduced-GB* (*reduced-GB F*) **by** (*rule reduced-GB-is-reduced-GB-dgrad-p-set*)
thus *?thesis* **unfolding** *is-reduced-GB-def* **by** *simp*
qed

lemma *reduced-GB-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆ *dgrad-p-set d m*
shows *pmdl* (*reduced-GB F*) = *pmdl F*
unfolding *reduced-GB-def*
by (*rule the1I2*, *rule ex-unique-reduced-GB-dgrad-p-set'*, *fact*, *fact*, *fact*, *elim conjE*)

lemma *reduced-GB-unique-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term ‘ Keys F*) **and** *F* ⊆

dgrad-p-set d m
  and *is-reduced-GB G* **and** *pmdl G = pmdl F*
 shows *reduced-GB F = G*
 by (*rule is-reduced-GB-unique*, *rule reduced-GB-is-reduced-GB-dgrad-p-set*, *fact+*,
   simp only: *reduced-GB-pmdl-dgrad-p-set*[*OF assms*(*1*, *2*, *3*)] *assms*(*5*))

lemma *reduced-GB-dgrad-p-set*:
 assumes *dickson-grading d* **and** *finite* (*component-of-term ' Keys F*) **and** *F* ⊆
dgrad-p-set d m
 shows *reduced-GB F* ⊆ *dgrad-p-set d m*
proof −
 from *assms* **obtain** *G* **where** *G*: *G* ⊆ *dgrad-p-set d m* **and** *is-reduced-GB G*
and *pmdl G = pmdl F*
  by (*rule ex-finite-reduced-GB-dgrad-p-set*)
 from *assms this*(*2*, *3*) **have** *reduced-GB F = G* **by** (*rule reduced-GB-unique-dgrad-p-set*)
 with *G* **show** *?thesis* **by** *simp*
qed

lemma *reduced-GB-unique*:
 assumes *finite G* **and** *is-reduced-GB G* **and** *pmdl G = pmdl F*
 shows *reduced-GB F = G*
proof −
 from *assms* **have** *finite G* ∧ *is-reduced-GB G* ∧ *pmdl G = pmdl F* **by** *simp*
 thus *?thesis* **unfolding** *reduced-GB-def*
 proof (*rule the-equality*)
  fix *G'*
  assume *finite G'* ∧ *is-reduced-GB G'* ∧ *pmdl G' = pmdl F*
  hence *is-reduced-GB G'* **and** *eq*: *pmdl G' = pmdl F* **by** *simp-all*
  note *this*(*1*)
  moreover note *assms*(*2*)
  moreover have *pmdl G' = pmdl G* **by** (*simp only*: *assms*(*3*) *eq*)
  ultimately show *G' = G* **by** (*rule is-reduced-GB-unique*)
 qed
qed

lemma *is-reduced-GB-empty*: *is-reduced-GB* {}
 by (*simp add*: *is-reduced-GB-def is-Groebner-basis-empty is-monic-set-def is-auto-reduced-def*)

lemma *is-reduced-GB-singleton*: *is-reduced-GB* {*f*} ⟷ *lc f = 1*
proof
 assume *is-reduced-GB* {*f*}
 hence *is-monic-set* {*f*} **and** *f* ≠ *0* **by** (*rule reduced-GB-D3*, *rule reduced-GB-D4*)
simp
 from *this*(*1*) - *this*(*2*) **show** *lc f = 1* **by** (*rule is-monic-setD*) *simp*
next
 assume *lc f = 1*
 moreover from *this* **have** *f* ≠ *0* **by** *auto*
 ultimately show *is-reduced-GB* {*f*}
  by (*simp add*: *is-reduced-GB-def is-Groebner-basis-singleton is-monic-set-def*

is-auto-reduced-def
not-is-red-empty)
qed

lemma *reduced-GB-empty*: *reduced-GB* {} = {}
using *finite.emptyI is-reduced-GB-empty refl* **by** (*rule reduced-GB-unique*)

lemma *reduced-GB-singleton*: *reduced-GB* {*f*} = (*if f* = *0 then* {} *else* {*monic f*})
proof (*cases f* = *0*)
case *True*
from *finite.emptyI is-reduced-GB-empty* **have** *reduced-GB* {*f*} = {}
by (*rule reduced-GB-unique*) (*simp add*: *True flip*: *pmdl.span-Diff-zero*[*of* {*0*}])
with *True* **show** *?thesis* **by** *simp*
next
case *False*
have *reduced-GB* {*f*} = {*monic f*}
proof (*rule reduced-GB-unique*)
from *False* **have** *lc f* ≠ *0* **by** (*rule lc-not-0*)
thus *is-reduced-GB* {*monic f*} **by** (*simp add*: *is-reduced-GB-singleton monic-def*)
next
have *pmdl* {*monic f*} = *pmdl* (*monic* ‘ {*f*}) **by** *simp*
also have ... = *pmdl* {*f*} **by** (*fact pmdl-image-monic*)
finally show *pmdl* {*monic f*} = *pmdl* {*f*} **.**
qed *simp*
with *False* **show** *?thesis* **by** *simp*
qed

lemma *ex-unique-reduced-GB-finite*: *finite F* ⟹ (∃!*G*. *finite G* ∧ *is-reduced-GB G* ∧ *pmdl G* = *pmdl F*)
by (*rule ex-unique-reduced-GB-dgrad-p-set′*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *finite-reduced-GB-finite*: *finite F* ⟹ *finite* (*reduced-GB F*)
by (*rule finite-reduced-GB-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-is-reduced-GB-finite*: *finite F* ⟹ *is-reduced-GB* (*reduced-GB F*)
by (*rule reduced-GB-is-reduced-GB-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-is-GB-finite*: *finite F* ⟹ *is-Groebner-basis* (*reduced-GB F*)
by (*rule reduced-GB-is-GB-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-is-auto-reduced-finite*: *finite F* ⟹ *is-auto-reduced* (*reduced-GB F*)
by (*rule reduced-GB-is-auto-reduced-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-is-monic-set-finite*: *finite F* ⟹ *is-monic-set* (*reduced-GB F*)
by (*rule reduced-GB-is-monic-set-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-nonzero-finite*: *finite F* ⟹ *0* ∉ *reduced-GB F*
by (*rule reduced-GB-nonzero-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-pmdl-finite*: *finite F* ⟹ *pmdl* (*reduced-GB F*) = *pmdl F*
by (*rule reduced-GB-pmdl-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

lemma *reduced-GB-unique-finite*: *finite F* ⟹ *is-reduced-GB G* ⟹ *pmdl G* = *pmdl F* ⟹ *reduced-GB F* = *G*
by (*rule reduced-GB-unique-dgrad-p-set*, *rule dickson-grading-dgrad-dummy*,
erule finite-imp-finite-component-Keys, *erule dgrad-p-set-exhaust-expl*)

end

13.2.5 Properties of the Reduced Gröbner Basis of an Ideal

context *gd-powerprod*
begin

lemma *ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set*:
assumes *dickson-grading d* **and** *F* ⊆ *punit.dgrad-p-set d m*
shows *ideal F* = *UNIV* ⟷ *punit.reduced-GB F* = {*1*}
proof −
have *fin*: *finite* (*local.punit.component-of-term* ‘ *Keys F*) **by** *simp*
show *?thesis*
proof
assume *ideal F* = *UNIV*
from *assms*(*1*) *fin assms*(*2*) **show** *punit.reduced-GB F* = {*1*}
proof (*rule punit.reduced-GB-unique-dgrad-p-set*)
show *punit.is-reduced-GB* {*1*} **unfolding** *punit.is-reduced-GB-def*
proof (*intro conjI*, *fact punit.is-Groebner-basis-singleton*)
show *punit.is-auto-reduced* {*1*} **unfolding** *punit.is-auto-reduced-def*
by (*rule ballI*, *simp add*: *remove-def punit.not-is-red-empty*)
next
show *punit.is-monic-set* {*1*}
by (*rule punit.is-monic-setI*, *simp del*: *single-one add*: *single-one*[*symmetric*])
qed *simp*
next
have *punit.pmdl* {*1*} = *ideal* {*1*} **by** *simp*
also have ... = *ideal F*
proof (*simp only*: ‹*ideal F* = *UNIV*› *ideal-eq-UNIV-iff-contains-one*)
have *1* ∈ {*1*} **..**
with *module-times* **show** *1* ∈ *ideal* {*1*} **by** (*rule module.span-base*)

qed
also have ... = *punit.pmdl F* **by** *simp*
finally show *punit.pmdl* {*1*} = *punit.pmdl F* **.**
qed
next
assume *punit.reduced-GB F* = {*1*}
hence *1* ∈ *punit.reduced-GB F* **by** *simp*
hence *1* ∈ *punit.pmdl* (*punit.reduced-GB F*) **by** (*rule punit.pmdl.span-base*)
also from *assms*(*1*) *fin assms*(*2*) **have** ... = *punit.pmdl F* **by** (*rule punit.reduced-GB-pmdl-dgrad-p-set*)
finally show *ideal F* = *UNIV* **by** (*simp add*: *ideal-eq-UNIV-iff-contains-one*)
qed
qed

lemmas *ideal-eq-UNIV-iff-reduced-GB-eq-one-finite* =
ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set[*OF dickson-grading-dgrad-dummy punit.dgrad-p-set-exhaust-expl*]

end

13.2.6 Context *od-term*

context *od-term*
begin

lemmas *ex-unique-reduced-GB* =
ex-unique-reduced-GB-dgrad-p-set'[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *finite-reduced-GB* =
finite-reduced-GB-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-is-reduced-GB* =
reduced-GB-is-reduced-GB-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-is-GB* =
reduced-GB-is-GB-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-is-auto-reduced* =
reduced-GB-is-auto-reduced-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-is-monic-set* =
reduced-GB-is-monic-set-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-nonzero* =
reduced-GB-nonzero-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-pmdl* =
reduced-GB-pmdl-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]
lemmas *reduced-GB-unique* =
reduced-GB-unique-dgrad-p-set[*OF dickson-grading-zero - subset-dgrad-p-set-zero*]

end

end

14 Sample Computations of Reduced Gröbner Bases

theory *Reduced-GB-Examples*
imports *Buchberger Reduced-GB Polynomials.MPoly-Type-Class-OAlist Code-Target-Rat*
begin

context *gd-term*
begin

definition *rgb* :: $('t \Rightarrow_0 'b)$ *list* $\Rightarrow$ $('t \Rightarrow_0 'b::field)$ *list*
where *rgb bs* = *comp-red-monic-basis* (*map fst* (*gb* (*map* (λ*b*. (*b*, ())) *bs*) ()))

definition *rgb-punit* :: $('a \Rightarrow_0 'b)$ *list* $\Rightarrow$ $('a \Rightarrow_0 'b::field)$ *list*
where *rgb-punit bs* = *punit.comp-red-monic-basis* (*map fst* (*gb-punit* (*map* (λ*b*. (*b*, ())) *bs*) ()))

lemma *compute-trd-aux* [*code*]:
trd-aux fs p r =
(*if is-zero p then*
r
else
case find-adds fs (*lt p*) *of*
None ⇒ *trd-aux fs* (*tail p*) (*plus-monomial-less r* (*lc p*) (*lt p*))
| *Some f* ⇒ *trd-aux fs* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *r*
)
by (*simp only*: *trd-aux.simps*[*of fs p r*] *plus-monomial-less-def is-zero-def*)

end

We only consider scalar polynomials here, but vector-polynomials could be handled, too.

global-interpretation *punit′*: *gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit cmp-term*
rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = (λ*x*. *x*)
and *punit.component-of-term* = (λ-. ())
and *punit.monom-mult* = *monom-mult-punit*
and *punit.mult-scalar* = *mult-scalar-punit*
and *punit′.punit.min-term* = *min-term-punit*
and *punit′.punit.lt* = *lt-punit cmp-term*
and *punit′.punit.lc* = *lc-punit cmp-term*
and *punit′.punit.tail* = *tail-punit cmp-term*
and *punit′.punit.ord-p* = *ord-p-punit cmp-term*
and *punit′.punit.ord-strict-p* = *ord-strict-p-punit cmp-term*
for *cmp-term* :: (*'a*::*nat*, *'b*::{*nat*,*add-wellorder*}) *pp nat-term-order*

defines *find-adds-punit* = *punit′.punit.find-adds*
and *trd-aux-punit* = *punit′.punit.trd-aux*

and *trd-punit* = *punit′.punit.trd*
and *spoly-punit* = *punit′.punit.spoly*
and *count-const-lt-components-punit* = *punit′.punit.count-const-lt-components*
and *count-rem-components-punit* = *punit′.punit.count-rem-components*
and *const-lt-component-punit* = *punit′.punit.const-lt-component*
and *full-gb-punit* = *punit′.punit.full-gb*
and *add-pairs-single-sorted-punit* = *punit′.punit.add-pairs-single-sorted*
and *add-pairs-punit* = *punit′.punit.add-pairs*
and *canon-pair-order-aux-punit* = *punit′.punit.canon-pair-order-aux*
and *canon-basis-order-punit* = *punit′.punit.canon-basis-order*
and *new-pairs-sorted-punit* = *punit′.punit.new-pairs-sorted*
and *product-crit-punit* = *punit′.punit.product-crit*
and *chain-ncrit-punit* = *punit′.punit.chain-ncrit*
and *chain-ocrit-punit* = *punit′.punit.chain-ocrit*
and *apply-icrit-punit* = *punit′.punit.apply-icrit*
and *apply-ncrit-punit* = *punit′.punit.apply-ncrit*
and *apply-ocrit-punit* = *punit′.punit.apply-ocrit*
and *trdsp-punit* = *punit′.punit.trdsp*
and *gb-sel-punit* = *punit′.punit.gb-sel*
and *gb-red-aux-punit* = *punit′.punit.gb-red-aux*
and *gb-red-punit* = *punit′.punit.gb-red*
and *gb-aux-punit* = *punit′.punit.gb-aux-punit*
and *gb-punit* = *punit′.punit.gb-punit* — Faster, because incorporates product criterion.
and *comp-min-basis-punit* = *punit′.punit.comp-min-basis*
and *comp-red-basis-aux-punit* = *punit′.punit.comp-red-basis-aux*
and *comp-red-basis-punit* = *punit′.punit.comp-red-basis*
and *monic-punit* = *punit′.punit.monic*
and *comp-red-monic-basis-punit* = *punit′.punit.comp-red-monic-basis*
and *rgb-punit* = *punit′.punit.rgb-punit*
subgoal by (*fact gd-powerprod-ord-pp-punit*)
subgoal by (*fact punit-adds-term*)
subgoal by (*simp add*: *id-def*)
subgoal by (*fact punit-component-of-term*)
subgoal by (*simp only*: *monom-mult-punit-def*)
subgoal by (*simp only*: *mult-scalar-punit-def*)
subgoal using *min-term-punit-def* **by** *fastforce*
subgoal by (*simp only*: *lt-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *lc-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *tail-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *ord-p-punit-def ord-pp-strict-punit-alt*)
subgoal by (*simp only*: *ord-strict-p-punit-def ord-pp-strict-punit-alt*)
done

lemma *compute-spoly-punit* [*code*]:
spoly-punit to p q = (*let t1* = *lt-punit to p*; *t2* = *lt-punit to q*; *l* = *lcs t1 t2 in* (*monom-mult-punit* (*1* / *lc-punit to p*) (*l* − *t1*) *p*) − (*monom-mult-punit* (*1* / *lc-punit to q*) (*l* − *t2*) *q*))
by (*simp add*: *punit′.punit.spoly-def Let-def punit′.punit.lc-def*)

lemma *compute-trd-punit* [*code*]: *trd-punit to fs p* = *trd-aux-punit to fs p* (*change-ord to 0*)
  by (*simp only*: *punit'.punit.trd-def change-ord-def*)

experiment begin interpretation *trivariate*$_0$*-rat* **.**

lemma
  rgb-punit DRLEX
    [
     $X \mathbin{\hat{}} 3 - X * Y * Z^2$,
     $Y^2 * Z - 1$
    ] =
    [
     $X \mathbin{\hat{}} 3 * Y - X * Z$,
     $-(X \mathbin{\hat{}} 3) + X * Y * Z^2$,
     $Y^2 * Z - 1$,
     $-(X * Z \mathbin{\hat{}} 3) + X \mathbin{\hat{}} 5$
    ]
  by *eval*

lemma
  rgb-punit DRLEX
    [
     $X^2 + Y^2 + Z^2 - 1$,
     $X * Y - Z - 1$,
     $Y^2 + X$,
     $Z^2 + X$
    ] =
    [
     1
    ]
  by *eval*

Note: The above computations have been cross-checked with Mathematica 11.1.

end

end

15 Macaulay Matrices

theory *Macaulay-Matrix*
 imports *More-MPoly-Type-Class Jordan-Normal-Form.Gauss-Jordan-Elimination*
begin

We build upon vectors and matrices represented by dimension and characteristic function, because later on we need to quantify the dimensions of certain matrices existentially. This is not possible (at least not easily possible) with a type-based approach, as in HOL-Multivariate Analysis.

15.1 More about Vectors

lemma *vec-of-list-alt*: *vec-of-list xs* = *vec* (*length xs*) (*nth xs*)
by (*transfer*, *rule refl*)

lemma *vec-cong*:
assumes $n = m$ **and** $\bigwedge i.\ i < m \Longrightarrow f\ i = g\ i$
shows *vec n f* = *vec m g*
using *assms* **by** *auto*

lemma *scalar-prod-comm*:
assumes *dim-vec v* = *dim-vec w*
shows $v \cdot w = w \cdot$ (*v*::'*a*::*comm-semiring-0 vec*)
by (*simp add*: *scalar-prod-def assms*, *rule sum.cong*, *rule refl*, *simp only*: *ac-simps*)

lemma *vec-scalar-mult-fun*: *vec n* ($\lambda x.\ c * f\ x$) = $c \cdot_v$ *vec n f*
by (*simp add*: *smult-vec-def*, *rule vec-cong*, *rule refl*, *simp*)

definition *mult-vec-mat* :: '*a vec* ⇒ '*a* :: *semiring-0 mat* ⇒ '*a vec* (**infixl** ‹$_v$∗› *70*)
where $v\ {}_v{*}\ A \equiv$ *vec* (*dim-col A*) ($\lambda j.\ v \cdot$ *col A j*)

definition *resize-vec* :: *nat* ⇒ '*a vec* ⇒ '*a vec*
where *resize-vec n v* = *vec n* (*vec-index v*)

lemma *dim-resize-vec*[*simp*]: *dim-vec* (*resize-vec n v*) = *n*
by (*simp add*: *resize-vec-def*)

lemma *resize-vec-carrier*: *resize-vec n v* ∈ *carrier-vec n*
by (*simp add*: *carrier-dim-vec*)

lemma *resize-vec-dim*[*simp*]: *resize-vec* (*dim-vec v*) *v* = *v*
by (*simp add*: *resize-vec-def eq-vecI*)

lemma *resize-vec-index*:
assumes $i < n$
shows *resize-vec n v* \$ *i* = *v* \$ *i*
using *assms* **by** (*simp add*: *resize-vec-def*)

lemma *mult-mat-vec-resize*:
$v\ {}_v{*}\ A$ = (*resize-vec* (*dim-row A*) *v*) $_v{*}\ A$
by (*simp add*: *mult-vec-mat-def scalar-prod-def*, *rule arg-cong2*[*of - - - - vec*], *rule*, *rule*,
rule sum.cong, *rule*, *simp add*: *resize-vec-index*)

lemma *assoc-mult-vec-mat*:
assumes *v* ∈ *carrier-vec n1* **and** *A* ∈ *carrier-mat n1 n2* **and** *B* ∈ *carrier-mat*

n2 n3
shows $v\ {}_v\!*\ (A * B) = (v\ {}_v\!*\ A)\ {}_v\!*\ B$
using *assms* **by** (*intro eq-vecI*, *auto simp add*: *mult-vec-mat-def mult-mat-vec-def assoc-scalar-prod*)

lemma *mult-vec-mat-transpose*:
assumes *dim-vec v* = *dim-row A*
shows $v\ {}_v\!*\ A$ = (*transpose-mat A*) $*_v$ (*v*::'*a*::*comm-semiring-0 vec*)
proof (*simp add*: *mult-vec-mat-def mult-mat-vec-def*, *rule vec-cong*, *rule refl*, *simp*)
fix *j*
show *v* · *col A j* = *col A j* · *v* **by** (*rule scalar-prod-comm*, *simp add*: *assms*)
qed

15.2 More about Matrices

definition *nzrows* :: '*a*::*zero mat* ⇒ '*a vec list*
where *nzrows A* = *filter* (λ*r*. *r* ≠ 0_v (*dim-col A*)) (*rows A*)

definition *row-space* :: '*a mat* ⇒ '*a*::*semiring-0 vec set*
where *row-space A* = (λ*v*. *mult-vec-mat v A*) ' (*carrier-vec* (*dim-row A*))

definition *row-echelon* :: '*a mat* ⇒ '*a*::*field mat*
where *row-echelon A* = *fst* (*gauss-jordan A* (1_m (*dim-row A*)))

15.2.1 *nzrows*

lemma *length-nzrows*: *length* (*nzrows A*) ≤ *dim-row A*
by (*simp add*: *nzrows-def length-rows*[*symmetric*] *del*: *length-rows*)

lemma *set-nzrows*: *set* (*nzrows A*) = *set* (*rows A*) − {0_v (*dim-col A*)}
by (*auto simp add*: *nzrows-def*)

lemma *nzrows-nth-not-zero*:
assumes *i* < *length* (*nzrows A*)
shows *nzrows A* ! *i* ≠ 0_v (*dim-col A*)
using *assms* **unfolding** *nzrows-def* **using** *nth-mem* **by** *force*

15.2.2 *row-space*

lemma *row-spaceI*:
assumes $x = v\ {}_v\!*\ A$
shows *x* ∈ *row-space A*
unfolding *row-space-def assms* **by** (*rule*, *fact mult-mat-vec-resize*, *fact resize-vec-carrier*)

lemma *row-spaceE*:
assumes *x* ∈ *row-space A*
obtains *v* **where** *v* ∈ *carrier-vec* (*dim-row A*) **and** $x = v\ {}_v\!*\ A$
using *assms* **unfolding** *row-space-def* **by** *auto*

lemma *row-space-alt*: *row-space A* = *range* (λ*v*. *mult-vec-mat v A*)

proof
  show *row-space A* ⊆ *range* (λ*v*. *v* $_v$∗ *A*) **unfolding** *row-space-def* **by** *auto*
next
  show *range* (λ*v*. *v* $_v$∗ *A*) ⊆ *row-space A*
  proof
    fix *x*
    assume *x* ∈ *range* (λ*v*. *v* $_v$∗ *A*)
    then obtain *v* **where** *x* = *v* $_v$∗ *A* **..**
    thus *x* ∈ *row-space A* **by** (*rule row-spaceI*)
  qed
qed

lemma *row-space-mult*:
  assumes *A* ∈ *carrier-mat nr nc* **and** *B* ∈ *carrier-mat nr nr*
  shows *row-space* (*B* ∗ *A*) ⊆ *row-space A*
proof
  from *assms*(*2*) *assms*(*1*) **have** *B* ∗ *A* ∈ *carrier-mat nr nc* **by** (*rule mult-carrier-mat*)
  hence *nr* = *dim-row* (*B* ∗ *A*) **by** *blast*
  fix *x*
  assume *x* ∈ *row-space* (*B* ∗ *A*)
  then obtain *v* **where** *v* ∈ *carrier-vec nr* **and** *x*: *x* = *v* $_v$∗ (*B* ∗ *A*)
    unfolding ‹*nr* = *dim-row* (*B* ∗ *A*)› **by** (*rule row-spaceE*)
  from *this*(*1*) *assms*(*2*) *assms*(*1*) **have** *x* = (*v* $_v$∗ *B*) $_v$∗ *A* **unfolding** *x* **by** (*rule assoc-mult-vec-mat*)
  thus *x* ∈ *row-space A* **by** (*rule row-spaceI*)
qed

lemma *row-space-mult-unit*:
  assumes *P* ∈ *Units* (*ring-mat TYPE*(′*a*::*semiring-1*) (*dim-row A*) *b*)
  shows *row-space* (*P* ∗ *A*) = *row-space A*
proof −
  have *A*: *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *simp*
  from *assms* **have** *P*: *P* ∈ *carrier* (*ring-mat TYPE*(′*a*) (*dim-row A*) *b*) **and**
    ∗: ∃ *Q*∈(*carrier* (*ring-mat TYPE*(′*a*) (*dim-row A*) *b*)).
        $Q \otimes_{ring\text{-}mat\ TYPE('a)\ (dim\text{-}row\ A)\ b} P = \mathbf{1}_{ring\text{-}mat\ TYPE('a)\ (dim\text{-}row\ A)\ b}$
    unfolding *Units-def* **by** *auto*
  from *P* **have** *P-in*: *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **by** (*simp add*: *ring-mat-def*)
  from ∗ **obtain** *Q* **where** *Q* ∈ *carrier* (*ring-mat TYPE*(′*a*) (*dim-row A*) *b*)
    and $Q \otimes_{ring\text{-}mat\ TYPE('a)\ (dim\text{-}row\ A)\ b} P = \mathbf{1}_{ring\text{-}mat\ TYPE('a)\ (dim\text{-}row\ A)\ b}$ **..**
  hence *Q-in*: *Q* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **and** *QP*: *Q* ∗ *P* = 1_m (*dim-row A*)
    by (*simp-all add*: *ring-mat-def*)
  show *?thesis*
  proof
    from *A P-in* **show** *row-space* (*P* ∗ *A*) ⊆ *row-space A* **by** (*rule row-space-mult*)
  next
    from *A P-in Q-in* **have** *Q* ∗ (*P* ∗ *A*) = (*Q* ∗ *P*) ∗ *A* **by** (*simp only*: *as-*

soc-mult-mat)
    also from *A* **have** ... = *A* **by** (*simp add*: *QP*)
    finally have *eq*: *row-space A* = *row-space* (*Q* ∗ (*P* ∗ *A*)) **by** *simp*
    show *row-space A* ⊆ *row-space* (*P* ∗ *A*) **unfolding** *eq* **by** (*rule row-space-mult*, *rule mult-carrier-mat*, *fact*+)
  qed
qed

15.2.3 *row-echelon*

lemma *row-eq-zero-iff-pivot-fun*:
  assumes *pivot-fun A f* (*dim-col A*) **and** *i* < *dim-row* (*A*::'*a*::*zero-neq-one mat*)
  shows (*row A i* = 0_v (*dim-col A*)) ⟷ (*f i* = *dim-col A*)
proof −
  have ∗: *dim-row A* = *dim-row A* **..**
  show *?thesis*
  proof
    assume *a*: *row A i* = 0_v (*dim-col A*)
    show *f i* = *dim-col A*
    proof (*rule ccontr*)
      assume *f i* ≠ *dim-col A*
      with *pivot-funD*(*1*)[*OF* ∗ *assms*] **have** ∗∗: *f i* < *dim-col A* **by** *simp*
      with ∗ *assms* **have** *A* $$ (*i*, *f i*) = *1* **by** (*rule pivot-funD*)
      with ∗∗ *assms*(*2*) **have** *row A i* $ (*f i*) = *1* **by** *simp*
      hence (*1*::'*a*) = (0_v (*dim-col A*)) $ (*f i*) **by** (*simp only*: *a*)
      also have ... = (*0*::'*a*) **using** ∗∗ **by** *simp*
      finally show *False* **by** *simp*
    qed
  next
    assume *a*: *f i* = *dim-col A*
    show *row A i* = 0_v (*dim-col A*)
    proof (*rule*, *simp-all add*: *assms*(*2*))
      fix *j*
      assume *j* < *dim-col A*
      hence *j* < *f i* **by** (*simp only*: *a*)
      with ∗ *assms* **show** *A* $$ (*i*, *j*) = *0* **by** (*rule pivot-funD*)
    qed
  qed
qed

lemma *row-not-zero-iff-pivot-fun*:
  assumes *pivot-fun A f* (*dim-col A*) **and** *i* < *dim-row* (*A*::'*a*::*zero-neq-one mat*)
  shows (*row A i* ≠ 0_v (*dim-col A*)) ⟷ (*f i* < *dim-col A*)
proof (*simp only*: *row-eq-zero-iff-pivot-fun*[*OF assms*])
  have *f i* ≤ *dim-col A* **by** (*rule pivot-funD*[**where** *?f* = *f*], *rule refl*, *fact*+)
  thus (*f i* ≠ *dim-col A*) = (*f i* < *dim-col A*) **by** *auto*
qed

lemma *pivot-fun-stabilizes*:

assumes *pivot-fun A f nc* **and** *i1* ≤ *i2* **and** *i2* < *dim-row A* **and** *nc* ≤ *f i1*
shows *f i2* = *nc*
proof −
from *assms*(*2*) **have** *i2* = *i1* + (*i2* − *i1*) **by** *simp*
then obtain *k* **where** *i2* = *i1* + *k* **..**
from *assms*(*3*) *assms*(*4*) **show** *?thesis* **unfolding** ‹*i2* = *i1* + *k*›
proof (*induct k arbitrary*: *i1*)
case *0*
from *this*(*1*) **have** *i1* < *dim-row A* **by** *simp*
from - *assms*(*1*) *this* **have** *f i1* ≤ *nc* **by** (*rule pivot-funD*, *intro refl*)
with ‹*nc* ≤ *f i1*› **show** *?case* **by** *simp*
next
case (*Suc k*)
from *Suc*(*2*) **have** *Suc* (*i1* + *k*) < *dim-row A* **by** *simp*
hence *Suc i1* + *k* < *dim-row A* **by** *simp*
hence *Suc i1* < *dim-row A* **by** *simp*
hence *i1* < *dim-row A* **by** *simp*
have *nc* ≤ *f* (*Suc i1*)
proof −
have *f i1* < *f* (*Suc i1*) ∨ *f* (*Suc i1*) = *nc* **by** (*rule pivot-funD*, *rule refl*, *fact*+)
with *Suc*(*3*) **show** *?thesis* **by** *auto*
qed
with ‹*Suc i1* + *k* < *dim-row A*› **have** *f* (*Suc i1* + *k*) = *nc* **by** (*rule Suc*(*1*))
thus *?case* **by** *simp*
qed
qed

lemma *pivot-fun-mono-strict*:
assumes *pivot-fun A f nc* **and** *i1* < *i2* **and** *i2* < *dim-row A* **and** *f i1* < *nc*
shows *f i1* < *f i2*
proof −
from *assms*(*2*) **have** *i2* − *i1* ≠ *0* **and** *i2* = *i1* + (*i2* − *i1*) **by** *simp-all*
then obtain *k* **where** *k* ≠ *0* **and** *i2* = *i1* + *k* **..**
from *this*(*1*) *assms*(*3*) *assms*(*4*) **show** *?thesis* **unfolding** ‹*i2* = *i1* + *k*›
proof (*induct k arbitrary*: *i1*)
case *0*
thus *?case* **by** *simp*
next
case (*Suc k*)
from *Suc*(*3*) **have** *Suc* (*i1* + *k*) < *dim-row A* **by** *simp*
hence *Suc i1* + *k* < *dim-row A* **by** *simp*
hence *Suc i1* < *dim-row A* **by** *simp*
hence *i1* < *dim-row A* **by** *simp*
have ∗: *f i1* < *f* (*Suc i1*)
proof −
have *f i1* < *f* (*Suc i1*) ∨ *f* (*Suc i1*) = *nc* **by** (*rule pivot-funD*, *rule refl*, *fact*+)
with *Suc*(*4*) **show** *?thesis* **by** *auto*

```
    qed
    show ?case
    proof (simp, cases k = 0)
      case True
      show f i1 < f (Suc (i1 + k)) by (simp add: True *)
    next
      case False
      have f (Suc i1) ≤ f (Suc i1 + k)
      proof (cases f (Suc i1) < nc)
        case True
        from False ‹Suc i1 + k < dim-row A› True have f (Suc i1) < f (Suc i1
+ k) by (rule Suc(1))
        thus ?thesis by simp
      next
        case False
        hence nc ≤ f (Suc i1) by simp
        from assms(1) - ‹Suc i1 + k < dim-row A› this have f (Suc i1 + k) = nc
          by (rule pivot-fun-stabilizes[where ?f=f], simp)
        moreover have f (Suc i1) = nc by (rule pivot-fun-stabilizes[where ?f=f],
fact, rule le-refl, fact+)
        ultimately show ?thesis by simp
      qed
      also have ... = f (i1 + Suc k) by simp
      finally have f (Suc i1) ≤ f (i1 + Suc k) .
      with * show f i1 < f (Suc (i1 + k)) by simp
    qed
  qed
qed

lemma pivot-fun-mono:
  assumes pivot-fun A f nc and i1 ≤ i2 and i2 < dim-row A
  shows f i1 ≤ f i2
proof −
  from assms(2) have i1 < i2 ∨ i1 = i2 by auto
  thus ?thesis
  proof
    assume i1 < i2
    show ?thesis
    proof (cases f i1 < nc)
      case True
      from assms(1) ‹i1 < i2› assms(3) this have f i1 < f i2 by (rule pivot-fun-mono-strict)
      thus ?thesis by simp
    next
      case False
      hence nc ≤ f i1 by simp
      from assms(1) - - this have f i1 = nc
      proof (rule pivot-fun-stabilizes[where ?f=f], simp)
        from assms(2) assms(3) show i1 < dim-row A by (rule le-less-trans)
      qed
```

moreover have *f i2* = *nc* **by** (*rule pivot-fun-stabilizes*[**where** *?f*=*f*], *fact*+)
ultimately show *?thesis* **by** *simp*
qed
next
assume *i1* = *i2*
thus *?thesis* **by** *simp*
qed
qed

lemma *row-echelon-carrier*:
assumes *A* ∈ *carrier-mat nr nc*
shows *row-echelon A* ∈ *carrier-mat nr nc*
proof −
from *assms* **have** *dim-row A* = *nr* **by** *simp*
let *?B* = 1_m (*dim-row A*)
note *assms*
moreover have *?B* ∈ *carrier-mat nr nr* **by** (*simp add*: ‹*dim-row A* = *nr*›)
moreover from *surj-pair* **obtain** *A′ B′* **where** ∗: *gauss-jordan A ?B* = (*A′*, *B′*)
by *metis*
ultimately have *A′* ∈ *carrier-mat nr nc* **by** (*rule gauss-jordan-carrier*)
thus *?thesis* **by** (*simp add*: *row-echelon-def* ∗)
qed

lemma *dim-row-echelon*[*simp*]:
shows *dim-row* (*row-echelon A*) = *dim-row A* **and** *dim-col* (*row-echelon A*) =
dim-col A
proof −
have *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *simp*
hence *row-echelon A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** (*rule row-echelon-carrier*)
thus *dim-row* (*row-echelon A*) = *dim-row A* **and** *dim-col* (*row-echelon A*) =
dim-col A **by** *simp-all*
qed

lemma *row-echelon-transform*:
obtains *P* **where** *P* ∈ *Units* (*ring-mat TYPE*(*′a*::*field*) (*dim-row A*) *b*) **and**
row-echelon A = *P* ∗ *A*
proof −
let *?B* = 1_m (*dim-row A*)
have *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *simp*
moreover have *?B* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **by** *simp*
moreover from *surj-pair* **obtain** *A′ B′* **where** ∗: *gauss-jordan A ?B* = (*A′*, *B′*)
by *metis*
ultimately have ∃ *P*∈*Units* (*ring-mat TYPE*(*′a*) (*dim-row A*) *b*). *A′* = *P* ∗ *A*
∧ *B′* = *P* ∗ *?B*
by (*rule gauss-jordan-transform*)
then obtain *P* **where** *P* ∈ *Units* (*ring-mat TYPE*(*′a*) (*dim-row A*) *b*) **and** ∗∗:
A′ = *P* ∗ *A* ∧ *B′* = *P* ∗ *?B* **..**
from *this*(*1*) **show** *?thesis*
proof

from ∗∗ **have** $A' = P * A$ **..**
thus *row-echelon* $A = P * A$ **by** (*simp add*: *row-echelon-def* ∗)
qed
qed

lemma *row-space-row-echelon*[*simp*]: *row-space* (*row-echelon* A) = *row-space* A
proof −
obtain P **where** ∗: $P \in$ *Units* (*ring-mat TYPE*(′*a*::*field*) (*dim-row* A) *Nil*) **and** ∗∗: *row-echelon* $A = P * A$
by (*rule row-echelon-transform*)
from ∗ **have** *row-space* $(P * A)$ = *row-space* A **by** (*rule row-space-mult-unit*)
thus *?thesis* **by** (*simp only*: ∗∗)
qed

lemma *row-echelon-pivot-fun*:
obtains f **where** *pivot-fun* (*row-echelon* A) f (*dim-col* (*row-echelon* A))
proof −
let *?B* = 1_m (*dim-row* A)
have $A \in$ *carrier-mat* (*dim-row* A) (*dim-col* A) **by** *simp*
moreover from *surj-pair* **obtain** A' B' **where** ∗: *gauss-jordan* A *?B* = (A', B') **by** *metis*
ultimately have *row-echelon-form* A' **by** (*rule gauss-jordan-row-echelon*)
then obtain f **where** *pivot-fun* A' f (*dim-col* A') **unfolding** *row-echelon-form-def* **..**
hence *pivot-fun* (*row-echelon* A) f (*dim-col* (*row-echelon* A)) **by** (*simp add*: *row-echelon-def* ∗)
thus *?thesis* **..**
qed

lemma *distinct-nzrows-row-echelon*: *distinct* (*nzrows* (*row-echelon* A))
unfolding *nzrows-def*
proof (*rule distinct-filterI*, *simp del*: *dim-row-echelon*)
let *?B* = *row-echelon* A
fix *i j*::*nat*
assume $i < j$ **and** $j <$ *dim-row ?B*
hence $i \neq j$ **and** $i <$ *dim-row ?B* **by** *simp-all*
assume *ri*: *row ?B i* $\neq 0_v$ (*dim-col ?B*) **and** *rj*: *row ?B j* $\neq 0_v$ (*dim-col ?B*)
obtain f **where** *pf*: *pivot-fun ?B f* (*dim-col ?B*) **by** (*fact row-echelon-pivot-fun*)
from *rj* **have** $f\ j <$ *dim-col ?B* **by** (*simp only*: *row-not-zero-iff-pivot-fun*[*OF pf* ‹$j <$ *dim-row ?B*›])
from - *pf* ‹$j <$ *dim-row ?B*› *this* ‹$i <$ *dim-row ?B*› ‹$i \neq j$› **have** ∗: *?B* $$ (*i*, *f j*) = *0*
by (*rule pivot-funD*(*5*), *intro refl*)
show *row ?B i* $\neq$ *row ?B j*
proof
assume *row ?B i* = *row ?B j*
hence *row ?B i* $ (*f j*) = *row ?B j* $ (*f j*) **by** *simp*
with ‹$i <$ *dim-row ?B*› ‹$j <$ *dim-row ?B*› ‹$f\ j <$ *dim-col ?B*› **have** *?B* $$ (*i*, *f j*) = *?B* $$ (*j*, *f j*) **by** *simp*

also from - *pf* ‹*j* < *dim-row ?B*› ‹*f j* < *dim-col ?B*› **have** ... = *1* **by** (*rule pivot-funD*, *intro refl*)
finally show *False* **by** (*simp add*: ∗)
qed
qed

15.3 Converting Between Polynomials and Macaulay Matrices

definition *poly-to-row* :: *'a list* ⇒ (*'a* $\Rightarrow_0$ *'b*::*zero*) ⇒ *'b vec* **where**
poly-to-row ts p = *vec-of-list* (*map* (*lookup p*) *ts*)

definition *polys-to-mat* :: *'a list* ⇒ (*'a* $\Rightarrow_0$ *'b*::*zero*) *list* ⇒ *'b mat* **where**
polys-to-mat ts ps = *mat-of-rows* (*length ts*) (*map* (*poly-to-row ts*) *ps*)

definition *list-to-fun* :: *'a list* ⇒ (*'b*::*zero*) *list* ⇒ *'a* ⇒ *'b* **where**
list-to-fun ts cs t = (*case map-of* (*zip ts cs*) *t of Some c* ⇒ *c* | *None* ⇒ *0*)

definition *list-to-poly* :: *'a list* ⇒ *'b list* ⇒ (*'a* $\Rightarrow_0$ *'b*::*zero*) **where**
list-to-poly ts cs = *Abs-poly-mapping* (*list-to-fun ts cs*)

definition *row-to-poly* :: *'a list* ⇒ *'b vec* ⇒ (*'a* $\Rightarrow_0$ *'b*::*zero*) **where**
row-to-poly ts r = *list-to-poly ts* (*list-of-vec r*)

definition *mat-to-polys* :: *'a list* ⇒ *'b mat* ⇒ (*'a* $\Rightarrow_0$ *'b*::*zero*) *list* **where**
mat-to-polys ts A = *map* (*row-to-poly ts*) (*rows A*)

lemma *dim-poly-to-row*: *dim-vec* (*poly-to-row ts p*) = *length ts*
by (*simp add*: *poly-to-row-def*)

lemma *poly-to-row-index*:
assumes *i* < *length ts*
shows *poly-to-row ts p* $ *i* = *lookup p* (*ts* ! *i*)
by (*simp add*: *poly-to-row-def vec-of-list-index assms*)

context *term-powerprod*
begin

lemma *poly-to-row-scalar-mult*:
assumes *keys p* ⊆ *set ts*
shows *row-to-poly ts* (*c* $\cdot_v$ (*poly-to-row ts p*)) = *c* · *p*
proof −
have *eq*: (*vec* (*length ts*) (λ*i*. *c* ∗ *poly-to-row ts p* $ *i*)) =
(*vec* (*length ts*) (λ*i*. *c* ∗ *lookup p* (*ts* ! *i*)))
by (*rule vec-cong*, *rule*, *simp only*: *poly-to-row-index*)
have ∗: *list-to-fun ts* (*list-of-vec* (*c* $\cdot_v$ (*poly-to-row ts p*))) = (λ*t*. *c* ∗ *lookup p t*)
proof (*rule*, *simp add*: *list-to-fun-def smult-vec-def dim-poly-to-row eq*,
simp add: *map-upt*[*of* λ*x*. *c* ∗ *lookup p x*] *map-of-zip-map*, *rule*)
fix *t*

assume *t* ∉ *set ts*
with *assms*(*1*) **have** *t* ∉ *keys p* **by** *auto*
thus *c* ∗ *lookup p t* = *0* **by** (*simp add*: *in-keys-iff*)
qed
have ∗∗: *lookup* (*Abs-poly-mapping* (*list-to-fun ts* (*list-of-vec* (*c* $\cdot_v$ (*poly-to-row ts p*))))) =
(λ*t*. *c* ∗ *lookup p t*)
proof (*simp only*: ∗, *rule Abs-poly-mapping-inverse*, *simp*, *rule finite-subset*, *rule*, *simp*)
fix *t*
assume *c* ∗ *lookup p t* ≠ *0*
hence *lookup p t* ≠ *0* **using** *mult-not-zero* **by** *blast*
thus *t* ∈ *keys p* **by** (*simp add*: *in-keys-iff*)
qed (*fact finite-keys*)
show *?thesis* **unfolding** *row-to-poly-def*
by (*rule poly-mapping-eqI*) (*simp only*: *list-to-poly-def* ∗∗ *lookup-map-scale*)
qed

lemma *poly-to-row-to-poly*:
assumes *keys p* ⊆ *set ts*
shows *row-to-poly ts* (*poly-to-row ts p*) = (*p*::′*t* $\Rightarrow_0$ ′*b*::*semiring-1*)
proof −
have *1* $\cdot_v$ (*poly-to-row ts p*) = *poly-to-row ts p* **by** *simp*
thus *?thesis* **using** *poly-to-row-scalar-mult*[*OF assms*, *of 1*] **by** *simp*
qed

lemma *lookup-list-to-poly*: *lookup* (*list-to-poly ts cs*) = *list-to-fun ts cs*
unfolding *list-to-poly-def*
proof (*rule Abs-poly-mapping-inverse*, *rule*, *rule finite-subset*)
show {*x*. *list-to-fun ts cs x* ≠ *0*} ⊆ *set ts*
proof (*rule*, *simp*)
fix *t*
assume *list-to-fun ts cs t* ≠ *0*
then obtain *c* **where** *map-of* (*zip ts cs*) *t* = *Some c* **unfolding** *list-to-fun-def* **by** *fastforce*
thus *t* ∈ *set ts* **by** (*meson in-set-zipE map-of-SomeD*)
qed
qed *simp*

lemma *list-to-fun-Nil* [*simp*]: *list-to-fun* [] *cs* = *0*
by (*simp only*: *zero-fun-def*, *rule*, *simp add*: *list-to-fun-def*)

lemma *list-to-poly-Nil* [*simp*]: *list-to-poly* [] *cs* = *0*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-list-to-poly*)

lemma *row-to-poly-Nil* [*simp*]: *row-to-poly* [] *r* = *0*
by (*simp only*: *row-to-poly-def*, *fact list-to-poly-Nil*)

lemma *lookup-row-to-poly*:

assumes *distinct ts* **and** *dim-vec r = length ts* **and** *i < length ts*
shows *lookup (row-to-poly ts r) (ts ! i) = r $ i*
proof (*simp only*: *row-to-poly-def lookup-list-to-poly*)
from *assms(2) assms(3)* **have** *i < dim-vec r* **by** *simp*
have *map-of (zip ts (list-of-vec r)) (ts ! i) = Some ((list-of-vec r) ! i)*
by (*rule map-of-zip-nth*, *simp-all only*: *length-list-of-vec assms(2)*, *fact*, *fact*)
also have ... = *Some (r $ i)* **by** (*simp only*: *list-of-vec-index*)
finally show *list-to-fun ts (list-of-vec r) (ts ! i) = r $ i* **by** (*simp add*: *list-to-fun-def*)
qed

lemma *keys-row-to-poly*: *keys (row-to-poly ts r) ⊆ set ts*
proof
fix *t*
assume *t ∈ keys (row-to-poly ts r)*
hence *lookup (row-to-poly ts r) t ≠ 0* **by** (*simp add*: *in-keys-iff*)
thus *t ∈ set ts*
proof (*simp add*: *row-to-poly-def lookup-list-to-poly list-to-fun-def del*: *lookup-not-eq-zero-eq-in-keys split*: *option.splits*)
fix *c*
assume *map-of (zip ts (list-of-vec r)) t = Some c*
thus *t ∈ set ts* **by** (*meson in-set-zipE map-of-SomeD*)
qed
qed

lemma *lookup-row-to-poly-not-zeroE*:
assumes *lookup (row-to-poly ts r) t ≠ 0*
obtains *i* **where** *i < length ts* **and** *t = ts ! i*
proof −
from *assms* **have** *t ∈ keys (row-to-poly ts r)* **by** (*simp add*: *in-keys-iff*)
have *t ∈ set ts* **by** (*rule*, *fact*, *fact keys-row-to-poly*)
then obtain *i* **where** *i < length ts* **and** *t = ts ! i* **by** (*metis in-set-conv-nth*)
thus *?thesis* **..**
qed

lemma *row-to-poly-zero* [*simp*]: *row-to-poly ts* (0_v (*length ts*)) = (*0::'t* $\Rightarrow_0$ *'b::zero*)
proof −
have *eq*: *map (λ-. 0::'b) [0..<length ts] = map (λ-. 0) ts* **by** (*simp add*: *map-replicate-const*)
show *?thesis*
by (*simp add*: *row-to-poly-def zero-vec-def*, *rule poly-mapping-eqI*,
simp add: *lookup-list-to-poly list-to-fun-def eq map-of-zip-map*)
qed

lemma *row-to-poly-zeroD*:
assumes *distinct ts* **and** *dim-vec r = length ts* **and** *row-to-poly ts r = 0*
shows *r* = 0_v (*length ts*)
proof (*rule*, *simp-all add*: *assms(2)*)
fix *i*
assume *i < length ts*
from *assms(3)* **have** *0 = lookup (row-to-poly ts r) (ts ! i)* **by** *simp*

```
 also from assms(1) assms(2) ‹i < length ts› have ... = r $ i by (rule lookup-row-to-poly)
  finally show r $ i = 0 by simp
qed

lemma row-to-poly-inj:
  assumes distinct ts and dim-vec r1 = length ts and dim-vec r2 = length ts
    and row-to-poly ts r1 = row-to-poly ts r2
  shows r1 = r2
proof (rule, simp-all add: assms(2) assms(3))
  fix i
  assume i < length ts
  have r1 $ i = lookup (row-to-poly ts r1) (ts ! i)
    by (simp only: lookup-row-to-poly[OF assms(1) assms(2) ‹i < length ts›])
  also from assms(4) have ... = lookup (row-to-poly ts r2) (ts ! i) by simp
   also from assms(1) assms(3) ‹i < length ts› have ... = r2 $ i by (rule
lookup-row-to-poly)
  finally show r1 $ i = r2 $ i .
qed

lemma row-to-poly-vec-plus:
  assumes distinct ts and length ts = n
  shows row-to-poly ts (vec n (f1 + f2)) = row-to-poly ts (vec n f1) + row-to-poly
ts (vec n f2)
proof (rule poly-mapping-eqI)
  fix t
  show lookup (row-to-poly ts (vec n (f1 + f2))) t =
        lookup (row-to-poly ts (vec n f1) + row-to-poly ts (vec n f2)) t
    (is lookup ?l t = lookup (?r1 + ?r2) t)
  proof (cases t ∈ set ts)
    case True
   then obtain j where j: j < length ts and t: t = ts ! j by (metis in-set-conv-nth)
    have d1: dim-vec (vec n f1) = length ts and d2: dim-vec (vec n f2) = length ts
      and da: dim-vec (vec n (f1 + f2)) = length ts by (simp-all add: assms(2))
    from j have j': j < n by (simp only: assms(2))
    show ?thesis
      by (simp only: t lookup-add lookup-row-to-poly[OF assms(1) d1 j]
            lookup-row-to-poly[OF assms(1) d2 j] lookup-row-to-poly[OF assms(1)
da j] index-vec[OF j'],
          simp only: plus-fun-def)
  next
    case False
    with keys-row-to-poly[of ts vec n (f1 + f2)] keys-row-to-poly[of ts vec n f1]
      keys-row-to-poly[of ts vec n f2] have t ∉ keys ?l and t ∉ keys ?r1 and t ∉
keys ?r2
      by auto
    from this(2) this(3) have t ∉ keys (?r1 + ?r2)
      by (meson Poly-Mapping.keys-add UnE in-mono)
    with ‹t ∉ keys ?l› show ?thesis by (simp add: in-keys-iff)
  qed
```

qed

lemma *row-to-poly-vec-sum*:
  assumes *distinct ts* **and** *length ts* = *n*
  shows *row-to-poly ts* (*vec n* (λ*j*. ∑ *i*∈*I*. *f i j*)) = ((∑ *i*∈*I*. *row-to-poly ts* (*vec n* (*f i*)))::′*t* ⇒$_0$ ′*b*::*comm-monoid-add*)
proof (*cases finite I*)
  case *True*
  thus *?thesis*
  proof (*induct I*)
    case *empty*
    thus *?case* **by** (*simp add*: *zero-vec-def*[*symmetric*] *assms*(*2*)[*symmetric*])
  next
    case (*insert x I*)
    have *row-to-poly ts* (*vec n* (λ*j*. ∑ *i*∈*insert x I*. *f i j*)) = *row-to-poly ts* (*vec n* (λ*j*. *f x j* + (∑ *i*∈*I*. *f i j*)))
      by (*simp add*: *insert*(*1*) *insert*(*2*))
    also have ... = *row-to-poly ts* (*vec n* (*f x* + (λ*j*. (∑ *i*∈*I*. *f i j*)))) **by** (*simp only*: *plus-fun-def*)
    also from *assms* **have** ... = *row-to-poly ts* (*vec n* (*f x*)) + *row-to-poly ts* (*vec n* (λ*j*. (∑ *i*∈*I*. *f i j*)))
      by (*rule row-to-poly-vec-plus*)
    also have ... = *row-to-poly ts* (*vec n* (*f x*)) + (∑ *i*∈*I*. *row-to-poly ts* (*vec n* (*f i*)))
      by (*simp only*: *insert*(*3*))
    also have ... = (∑ *i*∈*insert x I*. *row-to-poly ts* (*vec n* (*f i*)))
      by (*simp add*: *insert*(*1*) *insert*(*2*))
    finally show *?case* **.**
  qed
next
  case *False*
  thus *?thesis* **by** (*simp add*: *zero-vec-def*[*symmetric*] *assms*(*2*)[*symmetric*])
qed

lemma *row-to-poly-smult*:
  assumes *distinct ts* **and** *dim-vec r* = *length ts*
  shows *row-to-poly ts* ($c \cdot_v r$) = *c* · (*row-to-poly ts r*)
proof (*rule poly-mapping-eqI*, *simp only*: *lookup-map-scale*)
  fix *t*
  show *lookup* (*row-to-poly ts* ($c \cdot_v r$)) *t* = *c* ∗ *lookup* (*row-to-poly ts r*) *t* (**is** *lookup ?l t* = *c* ∗ *lookup ?r t*)
  proof (*cases t* ∈ *set ts*)
    case *True*
    then obtain *j* **where** *j*: *j* < *length ts* **and** *t*: *t* = *ts* ! *j* **by** (*metis in-set-conv-nth*)
    from *assms*(*2*) **have** *dm*: *dim-vec* ($c \cdot_v r$) = *length ts* **by** *simp*
    from *j* **have** *j*′: *j* < *dim-vec r* **by** (*simp only*: *assms*(*2*))
    show *?thesis*
      by (*simp add*: *t lookup-row-to-poly*[*OF assms j*] *lookup-row-to-poly*[*OF assms*(*1*) *dm j*] *index-smult-vec*(*1*)[*OF j*′])

next
case *False*
with *keys-row-to-poly*[*of ts c* $\cdot_v$ *r*] *keys-row-to-poly*[*of ts r*] **have**
$t \notin$ *keys ?l* **and** $t \notin$ *keys ?r* **by** *auto*
thus *?thesis* **by** (*simp add*: *in-keys-iff*)
qed
qed

lemma *poly-to-row-Nil* [*simp*]: *poly-to-row* [] *p* = *vec 0 f*
proof −
have *dim-vec* (*poly-to-row* [] *p*) = *0* **by** (*simp add*: *dim-poly-to-row*)
thus *?thesis* **by** *auto*
qed

lemma *polys-to-mat-Nil* [*simp*]: *polys-to-mat ts* [] = *mat 0* (*length ts*) *f*
by (*simp add*: *polys-to-mat-def mat-eq-iff*)

lemma *dim-row-polys-to-mat*[*simp*]: *dim-row* (*polys-to-mat ts ps*) = *length ps*
by (*simp add*: *polys-to-mat-def*)

lemma *dim-col-polys-to-mat*[*simp*]: *dim-col* (*polys-to-mat ts ps*) = *length ts*
by (*simp add*: *polys-to-mat-def*)

lemma *polys-to-mat-index*:
assumes *i* < *length ps* **and** *j* < *length ts*
shows (*polys-to-mat ts ps*) $$ (*i*, *j*) = *lookup* (*ps* ! *i*) (*ts* ! *j*)
by (*simp add*: *polys-to-mat-def index-mat*(*1*)[*OF assms*] *mat-of-rows-def nth-map*[*OF assms*(*1*)],
rule poly-to-row-index, *fact*)

lemma *row-polys-to-mat*:
assumes *i* < *length ps*
shows *row* (*polys-to-mat ts ps*) *i* = *poly-to-row ts* (*ps* ! *i*)
proof −
have *row* (*polys-to-mat ts ps*) *i* = (*map* (*poly-to-row ts*) *ps*) ! *i* **unfolding** *polys-to-mat-def*
proof (*rule mat-of-rows-row*)
from *assms* **show** *i* < *length* (*map* (*poly-to-row ts*) *ps*) **by** *simp*
next
show *map* (*poly-to-row ts*) *ps* ! *i* ∈ *carrier-vec* (*length ts*) **unfolding** *nth-map*[*OF assms*]
by (*rule carrier-vecI*, *fact dim-poly-to-row*)
qed
also from *assms* **have** ... = *poly-to-row ts* (*ps* ! *i*) **by** (*rule nth-map*)
finally show *?thesis* **.**
qed

lemma *col-polys-to-mat*:
assumes *j* < *length ts*

```
  shows col (polys-to-mat ts ps) j = vec-of-list (map (λp. lookup p (ts ! j)) ps)
  by (simp add: vec-of-list-alt col-def, rule vec-cong, rule refl, simp add: polys-to-mat-index
assms)

lemma length-mat-to-polys[simp]: length (mat-to-polys ts A) = dim-row A
  by (simp add: mat-to-polys-def mat-to-list-def)

lemma mat-to-polys-nth:
  assumes i < dim-row A
  shows (mat-to-polys ts A) ! i = row-to-poly ts (row A i)
proof −
  from assms have i < length (rows A) by (simp only: length-rows)
  thus ?thesis by (simp add: mat-to-polys-def)
qed

lemma Keys-mat-to-polys: Keys (set (mat-to-polys ts A)) ⊆ set ts
proof
  fix t
  assume t ∈ Keys (set (mat-to-polys ts A))
  then obtain p where p ∈ set (mat-to-polys ts A) and t: t ∈ keys p by (rule
in-KeysE)
  from this(1) obtain i where i < length (mat-to-polys ts A) and p: p =
(mat-to-polys ts A) ! i
    by (metis in-set-conv-nth)
  from this(1) have i < dim-row A by simp
  with p have p = row-to-poly ts (row A i) by (simp only: mat-to-polys-nth)
  with t have t ∈ keys (row-to-poly ts (row A i)) by simp
  also have ... ⊆ set ts by (fact keys-row-to-poly)
  finally show t ∈ set ts .
qed

lemma polys-to-mat-to-polys:
  assumes Keys (set ps) ⊆ set ts
  shows mat-to-polys ts (polys-to-mat ts ps) = (ps::('t ⇒₀ 'b::semiring-1) list)
  unfolding mat-to-polys-def mat-to-list-def
proof (rule nth-equalityI, simp-all)
  fix i
  assume i < length ps
  have *: keys (ps ! i) ⊆ set ts
    using ‹i < length ps› assms keys-subset-Keys nth-mem by blast
  show row-to-poly ts (row (polys-to-mat ts ps) i) = ps ! i
    by (simp only: row-polys-to-mat[OF ‹i < length ps›] poly-to-row-to-poly[OF *])
qed

lemma mat-to-polys-to-mat:
  assumes distinct ts and length ts = dim-col A
  shows (polys-to-mat ts (mat-to-polys ts A)) = A
proof
  fix i j
```

assume *i*: *i* < *dim-row A* **and** *j*: *j* < *dim-col A*
hence *i′*: *i* < *length* (*mat-to-polys ts A*) **and** *j′*: *j* < *length ts* **by** (*simp*, *simp only*: *assms*(*2*))
have *r*: *dim-vec* (*row A i*) = *length ts* **by** (*simp add*: *assms*(*2*))
show *polys-to-mat ts* (*mat-to-polys ts A*) $$ (*i*, *j*) = *A* $$ (*i*, *j*)
by (*simp only*: *polys-to-mat-index*[*OF i′ j′*] *mat-to-polys-nth*[*OF* ‹*i* < *dim-row A*›]
lookup-row-to-poly[*OF assms*(*1*) *r j′*] *index-row*(*1*)[*OF i j*])
qed (*simp-all add*: *assms*)

15.4 Properties of Macaulay Matrices

lemma *row-to-poly-vec-times*:
assumes *distinct ts* **and** *length ts* = *dim-col A*
shows *row-to-poly ts* (*v* $_v$* *A*) = ((∑ *i*=*0*..<*dim-row A*. (*v* $ *i*) · (*row-to-poly ts* (*row A i*)))::′*t* ⇒$_0$ ′*b*::*comm-semiring-0*)
proof (*simp add*: *mult-vec-mat-def scalar-prod-def row-to-poly-vec-sum*[*OF assms*], *rule sum.cong*, *rule*)
fix *i*
assume *i* ∈ {*0*..<*dim-row A*}
hence *i* < *dim-row A* **by** *simp*
have *dim-vec* (*row A i*) = *length ts* **by** (*simp add*: *assms*(*2*))
have ∗: *vec* (*dim-col A*) (λ*j*. *col A j* $ *i*) = *vec* (*dim-col A*) (λ*j*. *A* $$ (*i*, *j*))
by (*rule vec-cong*, *rule refl*, *simp add*: ‹*i* < *dim-row A*›)
have *vec* (*dim-col A*) (λ*j*. *v* $ *i* ∗ *col A j* $ *i*) = *v* $ *i* ·$_v$ *vec* (*dim-col A*) (λ*j*. *col A j* $ *i*)
by (*simp only*: *vec-scalar-mult-fun*)
also have ... = *v* $ *i* ·$_v$ (*row A i*) **by** (*simp only*: ∗ *row-def*[*symmetric*])
finally show *row-to-poly ts* (*vec* (*dim-col A*) (λ*j*. *v* $ *i* ∗ *col A j* $ *i*)) =
(*v* $ *i*) · (*row-to-poly ts* (*row A i*))
by (*simp add*: *row-to-poly-smult*[*OF assms*(*1*) ‹*dim-vec* (*row A i*) = *length ts*›])
qed

lemma *vec-times-polys-to-mat*:
assumes *Keys* (*set ps*) ⊆ *set ts* **and** *v* ∈ *carrier-vec* (*length ps*)
shows *row-to-poly ts* (*v* $_v$* (*polys-to-mat ts ps*)) = (∑ (*c*, *p*)←*zip* (*list-of-vec v*) *ps*. *c* · *p*)
(**is** *?l* = *?r*)
proof −
from *assms* **have** ∗: *dim-vec v* = *length ps* **by** (*simp only*: *carrier-dim-vec*)
have *eq*: *map* (λ*i*. *v* • *col* (*polys-to-mat ts ps*) *i*) [*0*..<*length ts*] =
map (λ*s*. *v* • (*vec-of-list* (*map* (λ*p*. *lookup p s*) *ps*))) *ts*
proof (*rule nth-equalityI*, *simp-all*)
fix *i*
assume *i* < *length ts*
hence *col* (*polys-to-mat ts ps*) *i* = *vec-of-list* (*map* (λ*p*. *lookup p* (*ts* ! *i*)) *ps*)
by (*rule col-polys-to-mat*)
thus *v* • *col* (*polys-to-mat ts ps*) *i* = *v* • *map-vec* (λ*p*. *lookup p* (*ts* ! *i*)) (*vec-of-list ps*)

```
      by simp
  qed
  show ?thesis
  proof (rule poly-mapping-eqI, simp add: mult-vec-mat-def row-to-poly-def lookup-list-to-poly
      eq list-to-fun-def map-of-zip-map lookup-sum-list o-def, intro conjI impI)
    fix t
    assume t ∈ set ts
    have v · vec-of-list (map (λp. lookup p t) ps) =
          (∑(c, p)←zip (list-of-vec v) ps. lookup (c · p) t)
    proof (simp add: scalar-prod-def vec-of-list-index)
      have (∑i = 0..<length ps. v $ i * lookup (ps ! i) t) =
            (∑i = 0..<length ps. (list-of-vec v) ! i * lookup (ps ! i) t)
        by (rule sum.cong, rule refl, simp add: *)
      also have ... = (∑(c, p)←zip (list-of-vec v) ps. c * lookup p t)
          by (simp only: sum-set-upt-eq-sum-list, rule sum-list-upt-zip, simp only:
length-list-of-vec *)
      finally show (∑i = 0..<length ps. v $ i * lookup (ps ! i) t) =
                (∑(c, p)←zip (list-of-vec v) ps. c * lookup p t) .
    qed
    thus v · map-vec (λp. lookup p t) (vec-of-list ps) =
          (∑x←zip (list-of-vec v) ps. lookup (case x of (c, x) ⇒ c · x) t)
     by (metis (mono-tags, lifting) case-prod-conv cond-case-prod-eta vec-of-list-map)
  next
    fix t
    assume t ∉ set ts
    with assms(1) have t ∉ Keys (set ps) by auto
    have (∑(c, p)←zip (list-of-vec v) ps. lookup (c · p) t) = 0
    proof (rule sum-list-zeroI, rule, simp)
      fix x
      assume x ∈ (λ(c, p). c * lookup p t) ` set (zip (list-of-vec v) ps)
      then obtain c p where cp: (c, p) ∈ set (zip (list-of-vec v) ps)
        and x: x = c * lookup p t by auto
      from cp have p ∈ set ps by (rule set-zip-rightD)
      with ‹t ∉ Keys (set ps)› have t ∉ keys p by (auto intro: in-KeysI)
      thus x = 0 by (simp add: x in-keys-iff)
    qed
    thus (∑x←zip (list-of-vec v) ps. lookup (case x of (c, x) ⇒ c · x) t) = 0
      by (metis (mono-tags, lifting) case-prod-conv cond-case-prod-eta)
  qed
qed

lemma row-space-subset-phull:
  assumes Keys (set ps) ⊆ set ts
  shows row-to-poly ts ` row-space (polys-to-mat ts ps) ⊆ phull (set ps)
    (is ?r ⊆ ?h)
proof
  fix q
  assume q ∈ ?r
  then obtain x where x1: x ∈ row-space (polys-to-mat ts ps)
```

and *q1*: *q* = *row-to-poly ts x* **..**
from *x1* **obtain** *v* **where** *v*: *v* ∈ *carrier-vec* (*dim-row* (*polys-to-mat ts ps*)) **and** *x*: *x* = *v* $_v$∗ *polys-to-mat ts ps*
by (*rule row-spaceE*)
from *v* **have** *v* ∈ *carrier-vec* (*length ps*) **by** (*simp only*: *dim-row-polys-to-mat*)
thm *vec-times-polys-to-mat*
with *x q1* **have** *q*: *q* = ($\sum$(*c*, *p*)←*zip* (*list-of-vec v*) *ps*. *c* · *p*)
by (*simp add*: *vec-times-polys-to-mat*[*OF assms*])
show *q* ∈ *?h* **unfolding** *q* **by** (*rule phull.span-listI*)
qed

lemma *phull-subset-row-space*:
assumes *Keys* (*set ps*) ⊆ *set ts*
shows *phull* (*set ps*) ⊆ *row-to-poly ts* ‘ *row-space* (*polys-to-mat ts ps*)
(**is** *?h* ⊆ *?r*)
proof
fix *q*
assume *q* ∈ *?h*
then obtain *cs* **where** *l*: *length cs* = *length ps* **and** *q*: *q* = ($\sum$(*c*, *p*)←*zip cs ps*. *c* · *p*)
by (*rule phull.span-listE*)
let *?v* = *vec-of-list cs*
from *l* **have** ∗: *?v* ∈ *carrier-vec* (*length ps*) **by** (*simp only*: *carrier-dim-vec dim-vec-of-list*)
let *?q* = *?v* $_v$∗ *polys-to-mat ts ps*
show *q* ∈ *?r*
proof
show *q* = *row-to-poly ts ?q*
by (*simp add*: *vec-times-polys-to-mat*[*OF assms* ∗] *q list-vec*)
next
show *?q* ∈ *row-space* (*polys-to-mat ts ps*) **by** (*rule row-spaceI*, *rule*)
qed
qed

lemma *row-space-eq-phull*:
assumes *Keys* (*set ps*) ⊆ *set ts*
shows *row-to-poly ts* ‘ *row-space* (*polys-to-mat ts ps*) = *phull* (*set ps*)
by (*rule*, *rule row-space-subset-phull*, *fact*, *rule phull-subset-row-space*, *fact*)

lemma *row-space-row-echelon-eq-phull*:
assumes *Keys* (*set ps*) ⊆ *set ts*
shows *row-to-poly ts* ‘ *row-space* (*row-echelon* (*polys-to-mat ts ps*)) = *phull* (*set ps*)
by (*simp add*: *row-space-eq-phull*[*OF assms*])

lemma *phull-row-echelon*:
assumes *Keys* (*set ps*) ⊆ *set ts* **and** *distinct ts*
shows *phull* (*set* (*mat-to-polys ts* (*row-echelon* (*polys-to-mat ts ps*)))) = *phull* (*set ps*)

proof −
  have *len-ts*: *length ts* = *dim-col* (*row-echelon* (*polys-to-mat ts ps*)) **by** *simp*
  have ∗: *Keys* (*set* (*mat-to-polys ts* (*row-echelon* (*polys-to-mat ts ps*)))) ⊆ *set ts*
    by (*fact Keys-mat-to-polys*)
  show *?thesis*
    by (*simp only*: *row-space-eq-phull*[*OF* ∗, *symmetric*] *mat-to-polys-to-mat*[*OF assms*(*2*) *len-ts*],
        rule row-space-row-echelon-eq-phull, *fact*)
qed

lemma *pmdl-row-echelon*:
  assumes *Keys* (*set ps*) ⊆ *set ts* **and** *distinct ts*
  shows *pmdl* (*set* (*mat-to-polys ts* (*row-echelon* (*polys-to-mat ts ps*)))) = *pmdl* (*set ps*)
    (**is** *?l* = *?r*)
proof
  show *?l* ⊆ *?r*
    by (*rule pmdl.span-subset-spanI*, *rule subset-trans*, *rule phull.span-superset*,
      simp only: *phull-row-echelon*[*OF assms*] *phull-subset-module*)
next
  show *?r* ⊆ *?l*
    by (*rule pmdl.span-subset-spanI*, *rule subset-trans*, *rule phull.span-superset*,
        simp only: *phull-row-echelon*[*OF assms*, *symmetric*] *phull-subset-module*)
qed

end

context *ordered-term*
begin

lemma *lt-row-to-poly-pivot-fun*:
  assumes *card S* = *dim-col* (*A*::′*b*::*semiring-1 mat*) **and** *pivot-fun A f* (*dim-col A*)
    and *i* < *dim-row A* **and** *f i* < *dim-col A*
  shows *lt* ((*mat-to-polys* (*pps-to-list S*) *A*) ! *i*) = (*pps-to-list S*) ! (*f i*)
proof −
  let *?ts* = *pps-to-list S*
  have *len-ts*: *length ?ts* = *dim-col A* **by** (*simp add*: *length-pps-to-list assms*(*1*))
  show *?thesis*
  proof (*simp add*: *mat-to-polys-nth*[*OF assms*(*3*)], *rule lt-eqI*)
    have *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *f i*) = (*row A i*) $ (*f i*)
      by (*rule lookup-row-to-poly*, *fact distinct-pps-to-list*, *simp-all add*: *len-ts assms*(*4*))
    also have ... = *A* $$ (*i*, *f i*) **using** *assms*(*3*) *assms*(*4*) **by** *simp*
    also have ... = *1* **by** (*rule pivot-funD*, *rule refl*, *fact*+)
    finally show *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *f i*) ≠ *0* **by** *simp*
  next
    fix *u*
    assume *a*: *lookup* (*row-to-poly ?ts* (*row A i*)) *u* ≠ *0*

then obtain *j* **where** *j*: *j* < *length ?ts* **and** *u*: *u* = *?ts* ! *j*
by (*rule lookup-row-to-poly-not-zeroE*)
from *j* **have** *j* < *card S* **and** *j* < *dim-col A* **by** (*simp only*: *length-pps-to-list*, *simp only*: *len-ts*)
from *a* **have** *0* ≠ *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *j*) **by** (*simp add*: *u*)
also have *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *j*) = (*row A i*) $ *j*
by (*rule lookup-row-to-poly*, *fact distinct-pps-to-list*, *simp add*: *len-ts*, *fact*)
finally have *A* $$ (*i*, *j*) ≠ *0* **using** *assms*(*3*) ‹*j* < *dim-col A*› **by** *simp*
from - ‹*j* < *card S*› **show** *u* $\preceq_t$ *?ts* ! *f i* **unfolding** *u*
proof (*rule pps-to-list-nth-leI*)
show *f i* ≤ *j*
proof (*rule ccontr*)
assume ¬ *f i* ≤ *j*
hence *j* < *f i* **by** *simp*
have *A* $$ (*i*, *j*) = *0* **by** (*rule pivot-funD*, *rule refl*, *fact*+)
with ‹*A* $$ (*i*, *j*) ≠ *0*› **show** *False* **..**
qed
qed
qed
qed

lemma *lc-row-to-poly-pivot-fun*:
assumes *card S* = *dim-col* (*A*::′*b*::*semiring-1 mat*) **and** *pivot-fun A f* (*dim-col A*)
and *i* < *dim-row A* **and** *f i* < *dim-col A*
shows *lc* ((*mat-to-polys* (*pps-to-list S*) *A*) ! *i*) = *1*
proof −
let *?ts* = *pps-to-list S*
have *len-ts*: *length ?ts* = *dim-col A* **by** (*simp only*: *length-pps-to-list assms*(*1*))
have *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *f i*) = (*row A i*) $ (*f i*)
by (*rule lookup-row-to-poly*, *fact distinct-pps-to-list*, *simp-all add*: *len-ts assms*(*4*))
also have ... = *A* $$ (*i*, *f i*) **using** *assms*(*3*) *assms*(*4*) **by** *simp*
finally have *eq*: *lookup* (*row-to-poly ?ts* (*row A i*)) (*?ts* ! *f i*) = *A* $$ (*i*, *f i*) **.**
show *?thesis*
by (*simp only*: *lc-def lt-row-to-poly-pivot-fun*[*OF assms*], *simp only*: *mat-to-polys-nth*[*OF assms*(*3*)] *eq*,
rule pivot-funD, *rule refl*, *fact*+)
qed

lemma *lt-row-to-poly-pivot-fun-less*:
assumes *card S* = *dim-col* (*A*::′*b*::*semiring-1 mat*) **and** *pivot-fun A f* (*dim-col A*)
and *i1* < *i2* **and** *i2* < *dim-row A* **and** *f i1* < *dim-col A* **and** *f i2* < *dim-col A*
shows (*pps-to-list S*) ! (*f i2*) $\prec_t$ (*pps-to-list S*) ! (*f i1*)
proof −
let *?ts* = *pps-to-list S*
have *len-ts*: *length ?ts* = *dim-col A* **by** (*simp add*: *length-pps-to-list assms*(*1*))
from *assms*(*3*) *assms*(*4*) **have** *i1* < *dim-row A* **by** *simp*
show *?thesis*

by (*rule pps-to-list-nth-lessI*, *rule pivot-fun-mono-strict*[**where** *?f=f*], *fact*, *fact*, *fact*, *fact*,
simp only: *assms*(*1*) *assms*(*6*))
qed

lemma *lt-row-to-poly-pivot-fun-eqD*:
assumes *card S* = *dim-col* (*A*::′*b*::*semiring-1 mat*) **and** *pivot-fun A f* (*dim-col A*)
and *i1* < *dim-row A* **and** *i2* < *dim-row A* **and** *f i1* < *dim-col A* **and** *f i2* < *dim-col A*
and (*pps-to-list S*) ! (*f i1*) = (*pps-to-list S*) ! (*f i2*)
shows *i1* = *i2*
proof (*rule linorder-cases*)
assume *i1* < *i2*
from *assms*(*1*) *assms*(*2*) *this assms*(*4*) *assms*(*5*) *assms*(*6*) **have**
(*pps-to-list S*) ! (*f i2*) $\prec_t$ (*pps-to-list S*) ! (*f i1*) **by** (*rule lt-row-to-poly-pivot-fun-less*)
with *assms*(*7*) **show** *?thesis* **by** *auto*
next
assume *i2* < *i1*
from *assms*(*1*) *assms*(*2*) *this assms*(*3*) *assms*(*6*) *assms*(*5*) **have**
(*pps-to-list S*) ! (*f i1*) $\prec_t$ (*pps-to-list S*) ! (*f i2*) **by** (*rule lt-row-to-poly-pivot-fun-less*)
with *assms*(*7*) **show** *?thesis* **by** *auto*
qed

lemma *lt-row-to-poly-pivot-in-keysD*:
assumes *card S* = *dim-col* (*A*::′*b*::*semiring-1 mat*) **and** *pivot-fun A f* (*dim-col A*)
and *i1* < *dim-row A* **and** *i2* < *dim-row A* **and** *f i1* < *dim-col A*
and (*pps-to-list S*) ! (*f i1*) ∈ *keys* ((*mat-to-polys* (*pps-to-list S*) *A*) ! *i2*)
shows *i1* = *i2*
proof (*rule ccontr*)
assume *i1* ≠ *i2*
hence *i2* ≠ *i1* **by** *simp*
let *?ts* = *pps-to-list S*
have *len-ts*: *length ?ts* = *dim-col A* **by** (*simp only*: *length-pps-to-list assms*(*1*))
from *assms*(*6*) **have** *0* ≠ *lookup* (*row-to-poly ?ts* (*row A i2*)) (*?ts* ! (*f i1*))
by (*auto simp*: *mat-to-polys-nth*[*OF assms*(*4*)])
also have *lookup* (*row-to-poly ?ts* (*row A i2*)) (*?ts* ! (*f i1*)) = (*row A i2*) $ (*f i1*)
by (*rule lookup-row-to-poly*, *fact distinct-pps-to-list*, *simp-all add*: *len-ts assms*(*5*))
finally have *A* $$ (*i2*, *f i1*) ≠ *0* **using** *assms*(*4*) *assms*(*5*) **by** *simp*
moreover have *A* $$ (*i2*, *f i1*) = *0* **by** (*rule pivot-funD*(*5*), *rule refl*, *fact*+)
ultimately show *False* **..**
qed

lemma *lt-row-space-pivot-fun*:
assumes *card S* = *dim-col* (*A*::′*b*::{*comm-semiring-0*,*semiring-1-no-zero-divisors*} *mat*)
and *pivot-fun A f* (*dim-col A*) **and** *p* ∈ *row-to-poly* (*pps-to-list S*) ‘ *row-space A* **and** *p* ≠ *0*

shows *lt p* ∈ *lt-set* (*set* (*mat-to-polys* (*pps-to-list S*) *A*))
proof −
let *?ts* = *pps-to-list S*
let *?I* = {*0*..<*dim-row A*}
have *len-ts*: *length ?ts* = *dim-col A* **by** (*simp add*: *length-pps-to-list assms*(*1*))
from *assms*(*3*) **obtain** *x* **where** *x* ∈ *row-space A* **and** *p*: *p* = *row-to-poly ?ts x*
..
from *this*(*1*) **obtain** *v* **where** *v* ∈ *carrier-vec* (*dim-row A*) **and** *x*: *x* = *v* $_v$∗ *A*
by (*rule row-spaceE*)

have *p′*: *p* = (∑ *i*∈*?I*. (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*)))
unfolding *p x* **by** (*rule row-to-poly-vec-times*, *fact distinct-pps-to-list*, *fact len-ts*)

have *lt* (∑ *i* = *0*..<*dim-row A*. (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*)))
∈ *lt-set* ((λ*i*. (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*))) ‘ {*0*..<*dim-row A*})
proof (*rule lt-sum-distinct-in-lt-set*, *rule*, *simp add*: *p′*[*symmetric*] ‹*p* ≠ *0*›)
fix *i1 i2*
let *?p1* = (*v* $ *i1*) · (*row-to-poly ?ts* (*row A i1*))
let *?p2* = (*v* $ *i2*) · (*row-to-poly ?ts* (*row A i2*))
assume *i1* ∈ *?I* **and** *i2* ∈ *?I*
hence *i1* < *dim-row A* **and** *i2* < *dim-row A* **by** *simp-all*

assume *?p1* ≠ *0*
hence *v* $ *i1* ≠ *0* **and** *row-to-poly ?ts* (*row A i1*) ≠ *0* **by** *auto*
hence *row A i1* ≠ 0_v (*length ?ts*) **by** *auto*
hence *f i1* < *dim-col A*
by (*simp add*: *len-ts row-not-zero-iff-pivot-fun*[*OF assms*(*2*) ‹*i1* < *dim-row A*›])
have *lt ?p1* = *lt* (*row-to-poly ?ts* (*row A i1*)) **by** (*rule lt-map-scale*, *fact*)
also have ... = *lt* ((*mat-to-polys ?ts A*) ! *i1*) **by** (*simp only*: *mat-to-polys-nth*[*OF* ‹*i1* < *dim-row A*›])
also have ... = *?ts* ! (*f i1*) **by** (*rule lt-row-to-poly-pivot-fun*, *fact*+)
finally have *lt1*: *lt ?p1* = *?ts* ! (*f i1*) **.**

assume *?p2* ≠ *0*
hence *v* $ *i2* ≠ *0* **and** *row-to-poly ?ts* (*row A i2*) ≠ *0* **by** *auto*
hence *row A i2* ≠ 0_v (*length ?ts*) **by** *auto*
hence *f i2* < *dim-col A*
by (*simp add*: *len-ts row-not-zero-iff-pivot-fun*[*OF assms*(*2*) ‹*i2* < *dim-row A*›])
have *lt ?p2* = *lt* (*row-to-poly ?ts* (*row A i2*)) **by** (*rule lt-map-scale*, *fact*)
also have ... = *lt* ((*mat-to-polys ?ts A*) ! *i2*) **by** (*simp only*: *mat-to-polys-nth*[*OF* ‹*i2* < *dim-row A*›])
also have ... = *?ts* ! (*f i2*) **by** (*rule lt-row-to-poly-pivot-fun*, *fact*+)
finally have *lt2*: *lt ?p2* = *?ts* ! (*f i2*) **.**

assume *lt ?p1* = *lt ?p2*
with *assms*(*1*) *assms*(*2*) ‹*i1* < *dim-row A*› ‹*i2* < *dim-row A*› ‹*f i1* < *dim-col*

A› ‹*f i2* < *dim-col A*›
  show *i1* = *i2* **unfolding** *lt1 lt2* **by** (*rule lt-row-to-poly-pivot-fun-eqD*)
 qed
 also have ... ⊆ *lt-set* ((λ*i*. *row-to-poly ?ts* (*row A i*)) ‘ {*0*..<*dim-row A*})
 proof
  fix *s*
  assume *s* ∈ *lt-set* ((λ*i*. (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*))) ‘ {*0*..<*dim-row A*})
  then obtain *f*
   where *f* ∈ (λ*i*. (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*))) ‘ {*0*..<*dim-row A*}
    and *f* ≠ *0* **and** *lt f* = *s* **by** (*rule lt-setE*)
  from *this*(*1*) **obtain** *i* **where** *i* ∈ {*0*..<*dim-row A*}
   and *f*: *f* = (*v* $ *i*) · (*row-to-poly ?ts* (*row A i*)) **..**
  from *this*(*2*) ‹*f* ≠ *0*› **have** *v* $ *i* ≠ *0* **and** ∗∗: *row-to-poly ?ts* (*row A i*) ≠ *0* **by** *auto*
  from ‹*lt f* = *s*› **have** *s* = *lt* ((*v* $ *i*) · (*row-to-poly ?ts* (*row A i*))) **by** (*simp only*: *f*)
  also from ‹*v* $ *i* ≠ *0*› **have** ... = *lt* (*row-to-poly ?ts* (*row A i*)) **by** (*rule lt-map-scale*)
  finally have *s*: *s* = *lt* (*row-to-poly ?ts* (*row A i*)) **.**
  show *s* ∈ *lt-set* ((λ*i*. *row-to-poly ?ts* (*row A i*)) ‘ {*0*..<*dim-row A*})
   unfolding *s* **by** (*rule lt-setI*, *rule*, *rule refl*, *fact*+)
 qed
 also have ... = *lt-set* ((λ*r*. *row-to-poly ?ts r*) ‘ (*row A* ‘ {*0*..<*dim-row A*}))
  by (*simp only*: *image-comp o-def*)
 also have ... = *lt-set* (*set* (*map* (λ*r*. *row-to-poly ?ts r*) (*map* (*row A*) [*0*..<*dim-row A*])))
  by (*metis image-set set-upt*)
 also have ... = *lt-set* (*set* (*mat-to-polys ?ts A*)) **by** (*simp only*: *mat-to-polys-def rows-def*)
 finally show *?thesis* **unfolding** *p′* **.**
qed

15.5 Functions *Macaulay-mat* and *Macaulay-list*

definition *Macaulay-mat* :: ($'t \Rightarrow_0 'b$) *list* ⇒ *'b*::*field mat*
 where *Macaulay-mat ps* = *polys-to-mat* (*Keys-to-list ps*) *ps*

definition *Macaulay-list* :: ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t \Rightarrow_0 'b$::*field*) *list*
 where *Macaulay-list ps* =
    filter (λ*p*. *p* ≠ *0*) (*mat-to-polys* (*Keys-to-list ps*) (*row-echelon* (*Macaulay-mat ps*)))

lemma *dim-Macaulay-mat*[*simp*]:
 dim-row (*Macaulay-mat ps*) = *length ps*
 dim-col (*Macaulay-mat ps*) = *card* (*Keys* (*set ps*))
 by (*simp-all add*: *Macaulay-mat-def length-Keys-to-list*)

lemma *Macaulay-list-Nil* [*simp*]: *Macaulay-list* [] = ([]::($'t \Rightarrow_0 'b$::*field*) *list*) (**is** *?l*

= -)
proof −
  have *length ?l* ≤ *length* (*mat-to-polys* (*Keys-to-list* ([]::('t ⇒₀ 'b) list))
      (*row-echelon* (*Macaulay-mat* ([]::('t ⇒₀ 'b) list))))
    unfolding *Macaulay-list-def* **by** (*fact length-filter-le*)
  also have ... = *0* **by** *simp*
  finally show *?thesis* **by** *simp*
qed

lemma *set-Macaulay-list*:
  set (*Macaulay-list ps*) =
    set (*mat-to-polys* (*Keys-to-list ps*) (*row-echelon* (*Macaulay-mat ps*))) − {*0*}
  by (*auto simp add*: *Macaulay-list-def*)

lemma *Keys-Macaulay-list*: *Keys* (*set* (*Macaulay-list ps*)) ⊆ *Keys* (*set ps*)
proof −
  have *Keys* (*set* (*Macaulay-list ps*)) ⊆ *set* (*Keys-to-list ps*)
    by (*simp only*: *set-Macaulay-list Keys-minus-zero*, *fact Keys-mat-to-polys*)
  also have ... = *Keys* (*set ps*) **by** (*fact set-Keys-to-list*)
  finally show *?thesis* **.**
qed

lemma *in-Macaulay-listE*:
  assumes *p* ∈ *set* (*Macaulay-list ps*)
   and *pivot-fun* (*row-echelon* (*Macaulay-mat ps*)) *f* (*dim-col* (*row-echelon* (*Macaulay-mat ps*)))
  obtains *i* **where** *i* < *dim-row* (*row-echelon* (*Macaulay-mat ps*))
    and *p* = (*mat-to-polys* (*Keys-to-list ps*) (*row-echelon* (*Macaulay-mat ps*))) ! *i*
    and *f i* < *dim-col* (*row-echelon* (*Macaulay-mat ps*))
proof −
  let *?ts* = *Keys-to-list ps*
  let *?A* = *Macaulay-mat ps*
  let *?E* = *row-echelon ?A*

  from *assms*(*1*) **have** *p* ∈ *set* (*mat-to-polys ?ts ?E*) − {*0*} **by** (*simp add*: *set-Macaulay-list*)
  hence *p* ∈ *set* (*mat-to-polys ?ts ?E*) **and** *p* ≠ *0* **by** *auto*
  from *this*(*1*) **obtain** *i* **where** *i* < *length* (*mat-to-polys ?ts ?E*) **and** *p*: *p* = (*mat-to-polys ?ts ?E*) ! *i*
    by (*metis in-set-conv-nth*)
  from *this*(*1*) **have** *i* < *dim-row ?E* **and** *i* < *dim-row ?A* **by** *simp-all*

  from *this*(*1*) *p* **show** *?thesis*
  proof
    from ‹*p* ≠ *0*› **have** *0* ≠ (*mat-to-polys ?ts ?E*) ! *i* **by** (*simp only*: *p*)
    also have (*mat-to-polys ?ts ?E*) ! *i* = *row-to-poly ?ts* (*row ?E i*)
      by (*simp only*: *Macaulay-list-def mat-to-polys-nth*[*OF* ‹*i* < *dim-row ?E*›])
    finally have ∗: *row-to-poly ?ts* (*row ?E i*) ≠ *0* **by** *simp*
    have *row ?E i* ≠ 0_v (*length ?ts*)

```
    proof
      assume row ?E i = 0_v (length ?ts)
      with * show False by simp
    qed
    hence row ?E i ≠ 0_v (dim-col ?E) by (simp add: length-Keys-to-list)
    thus f i < dim-col ?E
      by (simp only: row-not-zero-iff-pivot-fun[OF assms(2) ‹i < dim-row ?E›])
  qed
qed

lemma phull-Macaulay-list: phull (set (Macaulay-list ps)) = phull (set ps)
proof –
  have *: Keys (set ps) ⊆ set (Keys-to-list ps)
    by (simp add: set-Keys-to-list)
  have phull (set (Macaulay-list ps)) =
        phull (set (mat-to-polys (Keys-to-list ps) (row-echelon (Macaulay-mat ps))))
    by (simp only: set-Macaulay-list phull.span-Diff-zero)
  also have ... = phull (set ps)
   by (simp only: Macaulay-mat-def phull-row-echelon[OF * distinct-Keys-to-list])
  finally show ?thesis .
qed

lemma pmdl-Macaulay-list: pmdl (set (Macaulay-list ps)) = pmdl (set ps)
proof –
  have *: Keys (set ps) ⊆ set (Keys-to-list ps)
    by (simp add: set-Keys-to-list)
  have pmdl (set (Macaulay-list ps)) =
        pmdl (set (mat-to-polys (Keys-to-list ps) (row-echelon (Macaulay-mat ps))))
    by (simp only: set-Macaulay-list pmdl.span-Diff-zero)
  also have ... = pmdl (set ps)
   by (simp only: Macaulay-mat-def pmdl-row-echelon[OF * distinct-Keys-to-list])
  finally show ?thesis .
qed

lemma Macaulay-list-is-monic-set: is-monic-set (set (Macaulay-list ps))
proof (rule is-monic-setI)
  let ?ts = Keys-to-list ps
  let ?E = row-echelon (Macaulay-mat ps)

  fix p
  assume p ∈ set (Macaulay-list ps)
  obtain h where pivot-fun ?E h (dim-col ?E) by (rule row-echelon-pivot-fun)
  with ‹p ∈ set (Macaulay-list ps)› obtain i where i < dim-row ?E
    and p: p = (mat-to-polys ?ts ?E) ! i and h i < dim-col ?E
    by (rule in-Macaulay-listE)

  show lc p = 1 unfolding p Keys-to-list-eq-pps-to-list
    by (rule lc-row-to-poly-pivot-fun, simp, fact+)
qed
```

lemma *Macaulay-list-not-zero*: *0* ∉ *set* (*Macaulay-list ps*)
  by (*simp add*: *Macaulay-list-def*)

lemma *Macaulay-list-distinct-lt*:
  assumes *x* ∈ *set* (*Macaulay-list ps*) **and** *y* ∈ *set* (*Macaulay-list ps*)
    and *x* ≠ *y*
  shows *lt x* ≠ *lt y*
proof
  let *?S* = *Keys* (*set ps*)
  let *?ts* = *Keys-to-list ps*
  let *?E* = *row-echelon* (*Macaulay-mat ps*)

  assume *lt x* = *lt y*
  obtain *h* **where** *pf*: *pivot-fun ?E h* (*dim-col ?E*) **by** (*rule row-echelon-pivot-fun*)
  with *assms*(*1*) **obtain** *i1* **where** *i1* < *dim-row ?E*
    and *x*: *x* = (*mat-to-polys ?ts ?E*) ! *i1* **and** *h i1* < *dim-col ?E*
    by (*rule in-Macaulay-listE*)
  from *assms*(*2*) *pf* **obtain** *i2* **where** *i2* < *dim-row ?E*
    and *y*: *y* = (*mat-to-polys ?ts ?E*) ! *i2* **and** *h i2* < *dim-col ?E*
    by (*rule in-Macaulay-listE*)

  have *lt x* = *?ts* ! (*h i1*)
    by (*simp only*: *x Keys-to-list-eq-pps-to-list*, *rule lt-row-to-poly-pivot-fun*, *simp*, *fact+*)
  moreover have *lt y* = *?ts* ! (*h i2*)
    by (*simp only*: *y Keys-to-list-eq-pps-to-list*, *rule lt-row-to-poly-pivot-fun*, *simp*, *fact+*)
  ultimately have *?ts* ! (*h i1*) = *?ts* ! (*h i2*) **by** (*simp only*: ‹*lt x* = *lt y*›)
  hence *pps-to-list* (*Keys* (*set ps*)) ! *h i1* = *pps-to-list* (*Keys* (*set ps*)) ! *h i2*
    by (*simp only*: *Keys-to-list-eq-pps-to-list*)

  have *i1* = *i2*
  proof (*rule lt-row-to-poly-pivot-fun-eqD*)
    show *card ?S* = *dim-col ?E* **by** *simp*
  qed *fact+*
  hence *x* = *y* **by** (*simp only*: *x y*)
  with ‹*x* ≠ *y*› **show** *False* **..**
qed

lemma *Macaulay-list-lt*:
  assumes *p* ∈ *phull* (*set ps*) **and** *p* ≠ *0*
  obtains *g* **where** *g* ∈ *set* (*Macaulay-list ps*) **and** *g* ≠ *0* **and** *lt p* = *lt g*
proof −
  let *?S* = *Keys* (*set ps*)
  let *?ts* = *Keys-to-list ps*
  let *?E* = *row-echelon* (*Macaulay-mat ps*)
  let *?gs* = *mat-to-polys ?ts ?E*
  have *finite ?S* **by** (*rule finite-Keys*, *rule*)

have *?S* ⊆ *set ?ts* **by** (*simp only*: *set-Keys-to-list*)

from *assms*(*1*) ‹*?S* ⊆ *set ?ts*› **have** *p* ∈ *row-to-poly ?ts* ‘ *row-space ?E*
by (*simp only*: *Macaulay-mat-def row-space-row-echelon-eq-phull*[*symmetric*])
hence *p* ∈ *row-to-poly* (*pps-to-list ?S*) ‘ *row-space ?E*
by (*simp only*: *Keys-to-list-eq-pps-to-list*)

obtain *f* **where** *pivot-fun ?E f* (*dim-col ?E*) **by** (*rule row-echelon-pivot-fun*)

have *lt p* ∈ *lt-set* (*set ?gs*) **unfolding** *Keys-to-list-eq-pps-to-list*
by (*rule lt-row-space-pivot-fun*, *simp*, *fact*+)
then obtain *g* **where** *g* ∈ *set ?gs* **and** *g* ≠ *0* **and** *lt g* = *lt p* **by** (*rule lt-setE*)

show *?thesis*
proof
from ‹*g* ∈ *set ?gs*› ‹*g* ≠ *0*› **show** *g* ∈ *set* (*Macaulay-list ps*) **by** (*simp add*: *set-Macaulay-list*)
next
from ‹*lt g* = *lt p*› **show** *lt p* = *lt g* **by** *simp*
qed *fact*
qed

end

end

16 Faugère's F4 Algorithm

theory *F4*
imports *Macaulay-Matrix Algorithm-Schema*
begin

This theory implements Faugère's F4 algorithm based on *gd-term.gb-schema-direct*.

16.1 Symbolic Preprocessing

context *gd-term*
begin

definition *sym-preproc-aux-term1* :: ('*a* ⇒ *nat*) ⇒ ((('*t* $\Rightarrow_0$ '*b*) *list* × '*t list* × '*t list* × ('*t* $\Rightarrow_0$ '*b*) *list*) ×
(('*t* $\Rightarrow_0$ '*b*) *list* × '*t list* × '*t list* × ('*t* $\Rightarrow_0$ '*b*) *list*)) *set*
where *sym-preproc-aux-term1 d* =
{((*gs1*, *ks1*, *ts1*, *fs1*), (*gs2*::('*t* $\Rightarrow_0$ '*b*) *list*, *ks2*, *ts2*, *fs2*)). ∃ *t2*∈*set ts2*. ∀ *t1*∈*set ts1*. *t1* $\prec_t$ *t2*}

definition *sym-preproc-aux-term2* :: ('*a* ⇒ *nat*) ⇒ ((('*t* $\Rightarrow_0$ '*b*::*zero*) *list* × '*t list* × '*t list* × ('*t* $\Rightarrow_0$ '*b*) *list*) ×

$(('t \Rightarrow_0 'b)$ *list* $\times$ *'t list* $\times$ *'t list* $\times$ $('t \Rightarrow_0 'b)$ *list*$))$ *set*
where *sym-preproc-aux-term2 d* =
{((*gs1*, *ks1*, *ts1*, *fs1*), (*gs2*::($'t \Rightarrow_0 'b$) *list*, *ks2*, *ts2*, *fs2*)). *gs1* = *gs2* ∧
dgrad-set-le d (*pp-of-term* ‘ *set ts1*) (*pp-of-term* ‘ (*Keys* (*set gs2*) ∪ *set ts2*))}

definition *sym-preproc-aux-term*
where *sym-preproc-aux-term d* = *sym-preproc-aux-term1 d* ∩ *sym-preproc-aux-term2 d*

lemma *wfp-on-ord-term-strict*:
assumes *dickson-grading d*
shows *wfp-on* ($\prec_t$) (*pp-of-term* −‘ *dgrad-set d m*)
proof (*rule wfp-onI-min*)
fix *x Q*
assume *x* ∈ *Q* **and** *Q* ⊆ *pp-of-term* −‘ *dgrad-set d m*
from *wf-dickson-less-v*[*OF assms*, *of m*] ‹*x* ∈ *Q*› **obtain** *z*
where *z* ∈ *Q* **and** ∗: ⋀*y*. *dickson-less-v d m y z* ⟹ *y* ∉ *Q* **by** (*rule wfE-min*[*to-pred*], *blast*)
from *this*(*1*) ‹*Q* ⊆ *pp-of-term* −‘ *dgrad-set d m*› **have** *z* ∈ *pp-of-term* −‘ *dgrad-set d m* ..
show ∃ *z*∈*Q*. ∀ *y* ∈ *pp-of-term* −‘ *dgrad-set d m*. *y* $\prec_t$ *z* ⟶ *y* ∉ *Q*
proof (*intro bexI ballI impI*, *rule* ∗)
fix *y*
assume *y* ∈ *pp-of-term* −‘ *dgrad-set d m* **and** *y* $\prec_t$ *z*
from *this*(*1*) ‹*z* ∈ *pp-of-term* −‘ *dgrad-set d m*› **have** *d* (*pp-of-term y*) ≤ *m* **and** *d* (*pp-of-term z*) ≤ *m*
by (*simp-all add*: *dgrad-set-def*)
thus *dickson-less-v d m y z* **using** ‹*y* $\prec_t$ *z*› **by** (*rule dickson-less-vI*)
qed *fact*
qed

lemma *sym-preproc-aux-term1-wf-on*:
assumes *dickson-grading d*
shows *wfp-on* (λ*x y*. (*x*, *y*) ∈ *sym-preproc-aux-term1 d*) {*x*. *set* (*fst* (*snd* (*snd x*))) ⊆ *pp-of-term* −‘ *dgrad-set d m*}
proof (*rule wfp-onI-min*)
let *?B* = *pp-of-term* −‘ *dgrad-set d m*
let *?A* = {*x*::(($'t \Rightarrow_0 'b$) *list* × *'t list* × *'t list* × ($'t \Rightarrow_0 'b$) *list*). *set* (*fst* (*snd* (*snd x*))) ⊆ *?B*}
have *A-sub-Pow*: *set* ‘ *fst* ‘ *snd* ‘ *snd* ‘ *?A* ⊆ *Pow ?B* **by** *auto*
fix *x Q*
assume *x* ∈ *Q* **and** *Q* ⊆ *?A*
let *?Q* = {*ord-term-lin.Max* (*set* (*fst* (*snd* (*snd q*)))) | *q*. *q* ∈ *Q* ∧ *fst* (*snd* (*snd q*)) ≠ []}
show ∃ *z*∈*Q*. ∀ *y*∈{*x*. *set* (*fst* (*snd* (*snd x*))) ⊆ *?B*}. (*y*, *z*) ∈ *sym-preproc-aux-term1 d* ⟶ *y* ∉ *Q*
proof (*cases* ∃ *z*∈*Q*. *fst* (*snd* (*snd z*)) = [])

case *True*
then obtain z **where** $z \in Q$ **and** *fst* (*snd* (*snd* z)) = [] **..**
show *?thesis*
proof (*intro bexI ballI impI*)
fix y
assume (y, z) $\in$ *sym-preproc-aux-term1 d*
then obtain t **where** $t \in$ *set* (*fst* (*snd* (*snd* z))) **unfolding** *sym-preproc-aux-term1-def* **by** *auto*
with ‹*fst* (*snd* (*snd* z)) = []› **show** $y \notin Q$ **by** *simp*
qed *fact*
next
case *False*
hence ∗: $q \in Q \Longrightarrow$ *fst* (*snd* (*snd* q)) $\neq$ [] **for** q **by** *blast*
with ‹$x \in Q$› **have** *fst* (*snd* (*snd* x)) $\neq$ [] **by** *simp*
from *assms* **have** *wfp-on* ($\prec_t$) *?B* **by** (*rule wfp-on-ord-term-strict*)
moreover from ‹$x \in Q$› ‹*fst* (*snd* (*snd* x)) $\neq$ []›
have *ord-term-lin.Max* (*set* (*fst* (*snd* (*snd* x)))) $\in$ *?Q* **by** *blast*
moreover have *?Q* $\subseteq$ *?B*
proof (*rule*, *simp*, *elim exE conjE*, *simp*)
fix *a b c d0*
assume (*a*, *b*, *c*, *d0*) $\in Q$ **and** $c \neq$ []
from *this*(*1*) ‹$Q \subseteq$ *?A*› **have** (*a*, *b*, *c*, *d0*) $\in$ *?A* **..**
hence *pp-of-term* ‘ *set c* $\subseteq$ *dgrad-set d m* **by** *auto*
moreover have *pp-of-term* (*ord-term-lin.Max* (*set c*)) $\in$ *pp-of-term* ‘ *set c*
proof
from ‹$c \neq$ []› **show** *ord-term-lin.Max* (*set c*) $\in$ *set c* **by** *simp*
qed (*fact refl*)
ultimately show *pp-of-term* (*ord-term-lin.Max* (*set c*)) $\in$ *dgrad-set d m* **..**
qed
ultimately obtain t **where** $t \in$ *?Q* **and** *min*: $\bigwedge s.\ s \prec_t t \Longrightarrow s \notin$ *?Q* **by** (*rule wfp-onE-min*) *blast*
from *this*(*1*) **obtain** z **where** $z \in Q$ **and** *fst* (*snd* (*snd* z)) $\neq$ []
and t: t = *ord-term-lin.Max* (*set* (*fst* (*snd* (*snd* z)))) **by** *blast*
show *?thesis*
proof (*intro bexI ballI impI*, *rule*)
fix y
assume $y \in$ *?A* **and** (y, z) $\in$ *sym-preproc-aux-term1 d* **and** $y \in Q$
from *this*(*2*) **obtain** t' **where** $t' \in$ *set* (*fst* (*snd* (*snd* z)))
and ∗∗: $\bigwedge s.\ s \in$ *set* (*fst* (*snd* (*snd* y))) $\Longrightarrow s \prec_t t'$
unfolding *sym-preproc-aux-term1-def* **by** *auto*
from ‹$y \in Q$› **have** *fst* (*snd* (*snd* y)) $\neq$ [] **by** (*rule* ∗)
with ‹$y \in Q$› **have** *ord-term-lin.Max* (*set* (*fst* (*snd* (*snd* y)))) $\in$ *?Q* (**is** *?s* $\in$ -)
by *blast*
from ‹*fst* (*snd* (*snd* y)) $\neq$ []› **have** *?s* $\in$ *set* (*fst* (*snd* (*snd* y))) **by** *simp*
hence *?s* $\prec_t t'$ **by** (*rule* ∗∗)
also from ‹$t' \in$ *set* (*fst* (*snd* (*snd* z)))› **have** $t' \preceq_t t$ **unfolding** t
using ‹*fst* (*snd* (*snd* z)) $\neq$ []› **by** *simp*
finally have *?s* $\notin$ *?Q* **by** (*rule min*)

from *this* ‹*?s* ∈ *?Q*› **show** *False* **..**
qed *fact*
qed
qed

lemma *sym-preproc-aux-term-wf*:
assumes *dickson-grading d*
shows *wf* (*sym-preproc-aux-term d*)
proof (*rule wfI-min*)
fix *x*::((*'t* $\Rightarrow_0$ *'b*) *list* × *'t list* × *'t list* × (*'t* $\Rightarrow_0$ *'b*) *list*) **and** *Q*
assume *x* ∈ *Q*
let *?A* = *Keys* (*set* (*fst x*)) ∪ *set* (*fst* (*snd* (*snd x*)))
have *finite ?A* **by** (*simp add*: *finite-Keys*)
hence *finite* (*pp-of-term* ‘ *?A*) **by** (*rule finite-imageI*)
then obtain *m* **where** *pp-of-term* ‘ *?A* ⊆ *dgrad-set d m* **by** (*rule dgrad-set-exhaust*)
hence *A*: *?A* ⊆ *pp-of-term* −‘ *dgrad-set d m* **by** *blast*
let *?B* = *pp-of-term* −‘ *dgrad-set d m*
let *?Q* = {*q* ∈ *Q*. *Keys* (*set* (*fst q*)) ∪ *set* (*fst* (*snd* (*snd q*))) ⊆ *?B*}
from *assms* **have** *wfp-on* (λ*x y*. (*x*, *y*) ∈ *sym-preproc-aux-term1 d*) {*x*. *set* (*fst* (*snd* (*snd x*))) ⊆ *?B*}
by (*rule sym-preproc-aux-term1-wf-on*)
moreover from ‹*x* ∈ *Q*› *A* **have** *x* ∈ *?Q* **by** *simp*
moreover have *?Q* ⊆ {*x*. *set* (*fst* (*snd* (*snd x*))) ⊆ *?B*} **by** *auto*
ultimately obtain *z* **where** *z* ∈ *?Q*
and ∗: ⋀*y*. (*y*, *z*) ∈ *sym-preproc-aux-term1 d* ⟹ *y* ∉ *?Q* **by** (*rule wfp-onE-min*) *blast*
from *this*(*1*) **have** *z* ∈ *Q* **and** *Keys* (*set* (*fst z*)) ∪ *set* (*fst* (*snd* (*snd z*))) ⊆ *?B* **by** *simp-all*
from *this*(*2*) **have** *a*: *pp-of-term* ‘ (*Keys* (*set* (*fst z*)) ∪ *set* (*fst* (*snd* (*snd z*)))) ⊆ *dgrad-set d m*
by *blast*
show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *sym-preproc-aux-term d* ⟶ *y* ∉ *Q*
proof (*intro bexI allI impI*)
fix *y*
assume (*y*, *z*) ∈ *sym-preproc-aux-term d*
hence (*y*, *z*) ∈ *sym-preproc-aux-term1 d* **and** (*y*, *z*) ∈ *sym-preproc-aux-term2 d*
by (*simp-all add*: *sym-preproc-aux-term-def*)
from *this*(*2*) **have** *fst y* = *fst z*
and *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*snd* (*snd y*)))) (*pp-of-term* ‘ (*Keys* (*set* (*fst z*)) ∪ *set* (*fst* (*snd* (*snd z*)))))
by (*auto simp add*: *sym-preproc-aux-term2-def*)
from *this*(*2*) *a* **have** *pp-of-term* ‘ (*set* (*fst* (*snd* (*snd y*)))) ⊆ *dgrad-set d m*
by (*rule dgrad-set-le-dgrad-set*)
hence *Keys* (*set* (*fst y*)) ∪ *set* (*fst* (*snd* (*snd y*))) ⊆ *?B*
using *a* **by** (*auto simp add*: ‹*fst y* = *fst z*›)
moreover from ‹(*y*, *z*) ∈ *sym-preproc-aux-term1 d*› **have** *y* ∉ *?Q* **by** (*rule* ∗)
ultimately show *y* ∉ *Q* **by** *simp*
qed *fact*

qed

primrec *sym-preproc-addnew* :: ($'t \Rightarrow_0 'b$::*semiring-1*) *list* ⇒ *'t list* ⇒ ($'t \Rightarrow_0 'b$) *list* ⇒ *'t* ⇒
('t *list* × ($'t \Rightarrow_0 'b$) *list*) **where**
sym-preproc-addnew [] *vs fs* - = (*vs*, *fs*)|
sym-preproc-addnew (*g* # *gs*) *vs fs v* =
(*if lt g* $adds_t$ *v then*
(*let f* = *monom-mult 1* (*pp-of-term v* − *lp g*) *g in*
sym-preproc-addnew gs (*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail f*))) (*insert-list f fs*) *v*
)
else
sym-preproc-addnew gs vs fs v
)

lemma *fst-sym-preproc-addnew-less*:
assumes $\bigwedge u.\ u \in$ *set vs* $\Longrightarrow u \prec_t v$
and $u \in$ *set* (*fst* (*sym-preproc-addnew gs vs fs v*))
shows $u \prec_t v$
using *assms*
proof (*induct gs arbitrary*: *fs vs*)
case *Nil*
from *Nil*(*2*) **have** $u \in$ *set vs* **by** *simp*
thus *?case* **by** (*rule Nil*(*1*))
next
case (*Cons g gs*)
from *Cons*(*3*) **show** *?case*
proof (*simp add*: *Let-def split*: *if-splits*)
let *?t* = *pp-of-term v* − *lp g*
assume *lt g* $adds_t$ *v*
assume $u \in$ *set* (*fst* (*sym-preproc-addnew gs*
(*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail* (*monom-mult 1 ?t g*))))
(*insert-list* (*monom-mult 1 ?t g*) *fs*) *v*))
with - **show** *?thesis*
proof (*rule Cons*(*1*))
fix *u*
assume $u \in$ *set* (*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail* (*monom-mult 1 ?t g*))))
hence $u \in$ *set vs* ∨ $u \in$ *keys* (*tail* (*monom-mult 1 ?t g*))
by (*simp add*: *set-merge-wrt keys-to-list-def set-pps-to-list*)
thus $u \prec_t v$
proof
assume $u \in$ *set vs*
thus *?thesis* **by** (*rule Cons*(*2*))
next
assume $u \in$ *keys* (*tail* (*monom-mult 1 ?t g*))
hence $u \prec_t$ *lt* (*monom-mult 1 ?t g*) **by** (*rule keys-tail-less-lt*)
also have ... $\preceq_t$ *?t* ⊕ *lt g* **by** (*rule lt-monom-mult-le*)

also from ‹*lt g adds*$_t$ *v*› **have** ... = *v*
by (*metis add-diff-cancel-right′ adds-termE pp-of-term-splus*)
finally show *?thesis* **.**
qed
qed
next
assume *u* ∈ *set* (*fst* (*sym-preproc-addnew gs vs fs v*))
with *Cons*(*2*) **show** *?thesis* **by** (*rule Cons*(*1*))
qed
qed

lemma *fst-sym-preproc-addnew-dgrad-set-le*:
assumes *dickson-grading d*
shows *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*sym-preproc-addnew gs vs fs v*))) (*pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs*)))
proof (*induct gs arbitrary*: *fs vs*)
case *Nil*
show *?case* **by** (*auto intro*: *dgrad-set-le-subset*)
next
case (*Cons g gs*)
show *?case*
proof (*simp add*: *Let-def*, *intro conjI impI*)
assume *lt g adds*$_t$ *v*
let *?t* = *pp-of-term v* − *lp g*
let *?vs* = *merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail* (*monom-mult 1 ?t g*)))
let *?fs* = *insert-list* (*monom-mult 1 ?t g*) *fs*
from *Cons* **have** *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*sym-preproc-addnew gs ?vs ?fs v*)))
(*pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*)))
proof (*rule dgrad-set-le-trans*)
show *dgrad-set-le d* (*pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set ?vs*)))
(*pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*)))
unfolding *dgrad-set-le-def set-merge-wrt set-keys-to-list*
proof (*intro ballI*)
fix *s*
assume *s* ∈ *pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs* ∪ *keys* (*tail* (*monom-mult 1 ?t g*))))
hence *s* ∈ *pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs*)) ∪ *pp-of-term* ‘ *keys* (*tail* (*monom-mult 1 ?t g*))
by *auto*
thus ∃ *t* ∈ *pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*)). *d s* ≤ *d t*
proof
assume *s* ∈ *pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs*))
thus *?thesis* **by** (*auto simp add*: *Keys-insert*)
next
assume *s* ∈ *pp-of-term* ‘ *keys* (*tail* (*monom-mult 1 ?t g*))
hence *s* ∈ *pp-of-term* ‘ *keys* (*monom-mult 1 ?t g*) **by** (*auto simp add*:

keys-tail)
from *this keys-monom-mult-subset* **have** $s \in$ *pp-of-term* ‘ (⊕) *?t* ‘ *keys g* **by** *blast*
then obtain *u* **where** $u \in$ *keys g* **and** *s*: *s* = *pp-of-term* (*?t* ⊕ *u*) **by** *blast*
have *d s* = *d ?t* ∨ *d s* = *d* (*pp-of-term u*) **unfolding** *s pp-of-term-splus*
using *dickson-gradingD1*[*OF assms*] **by** *auto*
thus *?thesis*
proof
from ‹*lt g* $adds_t$ *v*› **have** *lp g adds pp-of-term v* **by** (*simp add*: *adds-term-def*)
assume *d s* = *d ?t*
also from *assms* ‹*lp g adds pp-of-term v*› **have** ... ≤ *d* (*pp-of-term v*)
by (*rule dickson-grading-minus*)
finally show *?thesis* **by** *blast*
next
assume *d s* = *d* (*pp-of-term u*)
moreover from ‹$u \in$ *keys g*› **have** $u \in$ *Keys* (*insert g* (*set gs*)) **by** (*simp add*: *Keys-insert*)
ultimately show *?thesis* **by** *auto*
qed
qed
qed
qed
thus *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*sym-preproc-addnew gs ?vs ?fs v*)))
(*insert* (*pp-of-term v*) (*pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *set vs*)))
by *simp*
next
from *Cons* **show** *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*sym-preproc-addnew gs vs fs v*)))
(*insert* (*pp-of-term v*) (*pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *set vs*)))
proof (*rule dgrad-set-le-trans*)
show *dgrad-set-le d* (*pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs*)))
(*insert* (*pp-of-term v*) (*pp-of-term* ‘ (*Keys* (*insert g* (*set gs*)) ∪ *set vs*)))
by (*rule dgrad-set-le-subset*, *auto simp add*: *Keys-def*)
qed
qed
qed

lemma *components-fst-sym-preproc-addnew-subset*:
component-of-term ‘ *set* (*fst* (*sym-preproc-addnew gs vs fs v*)) ⊆ *component-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs*))
proof (*induct gs arbitrary*: *fs vs*)
case *Nil*
show *?case* **by** (*auto intro*: *dgrad-set-le-subset*)
next
case (*Cons g gs*)

show *?case*
proof (*simp add*: *Let-def*, *intro conjI impI*)
assume *lt g adds*$_t$ *v*
let *?t* = *pp-of-term v* − *lp g*
let *?vs* = *merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail* (*monom-mult 1 ?t g*)))
let *?fs* = *insert-list* (*monom-mult 1 ?t g*) *fs*
from *Cons* **have** *component-of-term* ' *set* (*fst* (*sym-preproc-addnew gs ?vs ?fs v*)) ⊆
component-of-term ' (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*))
proof (*rule subset-trans*)
show *component-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set ?vs*)) ⊆
component-of-term ' (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*))
unfolding *set-merge-wrt set-keys-to-list*
proof
fix *k*
assume *k* ∈ *component-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs* ∪ *keys* (*tail* (*monom-mult 1 ?t g*))))
hence *k* ∈ *component-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs*)) ∪ *component-of-term* ' *keys* (*tail* (*monom-mult 1 ?t g*))
by *auto*
thus *k* ∈ *component-of-term* ' (*Keys* (*insert g* (*set gs*)) ∪ *insert v* (*set vs*))
proof
assume *k* ∈ *component-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs*))
thus *?thesis* **by** (*auto simp add*: *Keys-insert*)
next
assume *k* ∈ *component-of-term* ' *keys* (*tail* (*monom-mult 1 ?t g*))
hence *k* ∈ *component-of-term* ' *keys* (*monom-mult 1 ?t g*) **by** (*auto simp add*: *keys-tail*)
from *this keys-monom-mult-subset* **have** *k* ∈ *component-of-term* ' (⊕) *?t* ' *keys g* **by** *blast*
also have ... ⊆ *component-of-term* ' *keys g* **using** *component-of-term-splus* **by** *fastforce*
finally show *?thesis* **by** (*simp add*: *image-Un Keys-insert*)
qed
qed
qed
thus *component-of-term* ' *set* (*fst* (*sym-preproc-addnew gs ?vs ?fs v*)) ⊆
insert (*component-of-term v*) (*component-of-term* ' (*Keys* (*insert g* (*set gs*)) ∪ *set vs*))
by *simp*
next
from *Cons* **show** *component-of-term* ' *set* (*fst* (*sym-preproc-addnew gs vs fs v*)) ⊆
insert (*component-of-term v*) (*component-of-term* ' (*Keys* (*insert g* (*set gs*)) ∪ *set vs*))
proof (*rule subset-trans*)
show *component-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs*)) ⊆
insert (*component-of-term v*) (*component-of-term* ' (*Keys* (*insert g* (*set gs*)) ∪ *set vs*))

    by (*auto simp add*: *Keys-def*)
  qed
 qed
qed

lemma *fst-sym-preproc-addnew-superset*: *set vs* ⊆ *set* (*fst* (*sym-preproc-addnew gs vs fs v*))
proof (*induct gs arbitrary*: *vs fs*)
 case *Nil*
 show *?case* **by** *simp*
next
 case (*Cons g gs*)
 show *?case*
 proof (*simp add*: *Let-def*, *intro conjI impI*)
  let *?t* = *pp-of-term v* − *lp g*
  define *f* **where** *f* = *monom-mult 1 ?t g*
  have *set vs* ⊆ *set* (*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail f*))) **by** (*auto simp add*: *set-merge-wrt*)
  thus *set vs* ⊆ *set* (*fst* (*sym-preproc-addnew gs*
        (*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail f*))) (*insert-list f fs*) *v*))
   using *Cons* **by** (*rule subset-trans*)
 next
  show *set vs* ⊆ *set* (*fst* (*sym-preproc-addnew gs vs fs v*)) **by** (*fact Cons*)
 qed
qed

lemma *snd-sym-preproc-addnew-superset*: *set fs* ⊆ *set* (*snd* (*sym-preproc-addnew gs vs fs v*))
proof (*induct gs arbitrary*: *vs fs*)
 case *Nil*
 show *?case* **by** *simp*
next
 case (*Cons g gs*)
 show *?case*
 proof (*simp add*: *Let-def*, *intro conjI impI*)
  let *?t* = *pp-of-term v* − *lp g*
  define *f* **where** *f* = *monom-mult 1 ?t g*
  have *set fs* ⊆ *set* (*insert-list f fs*) **by** (*auto simp add*: *set-insert-list*)
  thus *set fs* ⊆ *set* (*snd* (*sym-preproc-addnew gs*
        (*merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail f*))) (*insert-list f fs*) *v*))
   using *Cons* **by** (*rule subset-trans*)
 next
  show *set fs* ⊆ *set* (*snd* (*sym-preproc-addnew gs vs fs v*)) **by** (*fact Cons*)
 qed
qed

lemma *in-snd-sym-preproc-addnewE*:

assumes $p \in set$ (*snd* (*sym-preproc-addnew gs vs fs v*))
assumes *1*: $p \in set\ fs \Longrightarrow thesis$
assumes *2*: $\bigwedge g\ s.\ g \in set\ gs \Longrightarrow p = monom\text{-}mult\ 1\ s\ g \Longrightarrow thesis$
shows *thesis*
using *assms*
proof (*induct gs arbitrary*: *vs fs thesis*)
case *Nil*
from *Nil*(*1*) **have** $p \in set\ fs$ **by** *simp*
thus *?case* **by** (*rule Nil*(*2*))
next
case (*Cons g gs*)
from *Cons*(*2*) **show** *?case*
proof (*simp add*: *Let-def split*: *if-splits*)
define *f* **where** $f = monom\text{-}mult\ 1\ (pp\text{-}of\text{-}term\ v - lp\ g)\ g$
define *ts'* **where** $ts' = merge\text{-}wrt\ (\succ_t)\ vs\ (keys\text{-}to\text{-}list\ (tail\ f))$
define *fs'* **where** $fs' = insert\text{-}list\ f\ fs$
assume $p \in set$ (*snd* (*sym-preproc-addnew gs ts' fs' v*))
thus *?thesis*
proof (*rule Cons*(*1*))
assume $p \in set\ fs'$
hence $p = f \lor p \in set\ fs$ **by** (*simp add*: *fs'-def set-insert-list*)
thus *?thesis*
proof
assume $p = f$
have $g \in set\ (g \# gs)$ **by** *simp*
from *this* ‹$p = f$› **show** *?thesis* **unfolding** *f-def* **by** (*rule Cons*(*4*))
next
assume $p \in set\ fs$
thus *?thesis* **by** (*rule Cons*(*3*))
qed
next
fix *h s*
assume $h \in set\ gs$
hence $h \in set\ (g \# gs)$ **by** *simp*
moreover assume $p = monom\text{-}mult\ 1\ s\ h$
ultimately show *thesis* **by** (*rule Cons*(*4*))
qed
next
assume $p \in set$ (*snd* (*sym-preproc-addnew gs vs fs v*))
moreover note *Cons*(*3*)
moreover have $h \in set\ gs \Longrightarrow p = monom\text{-}mult\ 1\ s\ h \Longrightarrow thesis$ **for** *h s*
proof −
assume $h \in set\ gs$
hence $h \in set\ (g \# gs)$ **by** *simp*
moreover assume $p = monom\text{-}mult\ 1\ s\ h$
ultimately show *thesis* **by** (*rule Cons*(*4*))
qed
ultimately show *?thesis* **by** (*rule Cons*(*1*))
qed

qed

lemma *sym-preproc-addnew-pmdl*:
pmdl (*set gs* ∪ *set* (*snd* (*sym-preproc-addnew gs vs fs v*))) = *pmdl* (*set gs* ∪ *set fs*)
(**is** *pmdl* (*set gs* ∪ *?l*) = *?r*)
proof
have *set gs* ⊆ *set gs* ∪ *set fs* **by** *simp*
also have ... ⊆ *?r* **by** (*fact pmdl.span-superset*)
finally have *set gs* ⊆ *?r* **.**
moreover have *?l* ⊆ *?r*
proof
fix *p*
assume *p* ∈ *?l*
thus *p* ∈ *?r*
proof (*rule in-snd-sym-preproc-addnewE*)
assume *p* ∈ *set fs*
hence *p* ∈ *set gs* ∪ *set fs* **by** *simp*
thus *?thesis* **by** (*rule pmdl.span-base*)
next
fix *g s*
assume *g* ∈ *set gs* **and** *p*: *p* = *monom-mult 1 s g*
from *this*(*1*) ‹*set gs* ⊆ *?r*› **have** *g* ∈ *?r* **..**
thus *?thesis* **unfolding** *p* **by** (*rule pmdl-closed-monom-mult*)
qed
qed
ultimately have *set gs* ∪ *?l* ⊆ *?r* **by** *blast*
thus *pmdl* (*set gs* ∪ *?l*) ⊆ *?r* **by** (*rule pmdl.span-subset-spanI*)
next
from *snd-sym-preproc-addnew-superset* **have** *set gs* ∪ *set fs* ⊆ *set gs* ∪ *?l* **by** *blast*
thus *?r* ⊆ *pmdl* (*set gs* ∪ *?l*) **by** (*rule pmdl.span-mono*)
qed

lemma *Keys-snd-sym-preproc-addnew*:
Keys (*set* (*snd* (*sym-preproc-addnew gs vs fs v*))) ∪ *insert v* (*set vs*) =
Keys (*set fs*) ∪ *insert v* (*set* (*fst* (*sym-preproc-addnew gs vs* (*fs*::(*'t* $\Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*) *v*)))
proof (*induct gs arbitrary*: *vs fs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons g gs*)
from *Cons* **have** *eq*: *insert v* (*Keys* (*set* (*snd* (*sym-preproc-addnew gs ts' fs' v*))) ∪ *set ts'*) =
insert v (*Keys* (*set fs'*) ∪ *set* (*fst* (*sym-preproc-addnew gs ts' fs' v*)))
for *ts' fs'* **by** *simp*
show *?case*

proof (*simp add*: *Let-def eq*, *rule*)
assume *lt g* $adds_t$ *v*
let *?t* = *pp-of-term v* − *lp g*
define *f* **where** *f* = *monom-mult 1 ?t g*
define *ts′* **where** *ts′* = *merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail f*))
define *fs′* **where** *fs′* = *insert-list f fs*
have *keys* (*tail f*) = *keys f* − {*v*}
proof (*cases g* = *0*)
case *True*
hence *f* = *0* **by** (*simp add*: *f-def*)
thus *?thesis* **by** *simp*
next
case *False*
hence *lt f* = *?t* ⊕ *lt g* **by** (*simp add*: *f-def lt-monom-mult*)
also from ‹*lt g* $adds_t$ *v*› **have** ... = *v*
by (*metis add-diff-cancel-right′ adds-termE pp-of-term-splus*)
finally show *?thesis* **by** (*simp add*: *keys-tail*)
qed
hence *ts′*: *set ts′* = *set vs* ∪ (*keys f* − {*v*})
by (*simp add*: *ts′-def set-merge-wrt set-keys-to-list*)
have *fs′*: *set fs′* = *insert f* (*set fs*) **by** (*simp add*: *fs′-def set-insert-list*)
hence *f* ∈ *set fs′* **by** *simp*
from *this snd-sym-preproc-addnew-superset* **have** *f* ∈ *set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*)) **..**
hence *keys f* ⊆ *Keys* (*set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*))) **by** (*rule keys-subset-Keys*)
hence *insert v* (*Keys* (*set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*))) ∪ *set vs*) =
insert v (*Keys* (*set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*))) ∪ *set ts′*)
by (*auto simp add*: *ts′*)
also have ... = *insert v* (*Keys* (*set fs′*) ∪ *set* (*fst* (*sym-preproc-addnew gs ts′ fs′ v*)))
by (*fact eq*)
also have ... = *insert v* (*Keys* (*set fs*) ∪ *set* (*fst* (*sym-preproc-addnew gs ts′ fs′ v*)))
proof −
{
fix *u*
assume *u* ≠ *v* **and** *u* ∈ *keys f*
hence *u* ∈ *set ts′* **by** (*simp add*: *ts′*)
from *this fst-sym-preproc-addnew-superset* **have** *u* ∈ *set* (*fst* (*sym-preproc-addnew gs ts′ fs′ v*)) **..**
}
thus *?thesis* **by** (*auto simp add*: *fs′ Keys-insert*)
qed
finally show *insert v* (*Keys* (*set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*))) ∪ *set vs*) =
insert v (*Keys* (*set fs*) ∪ *set* (*fst* (*sym-preproc-addnew gs ts′ fs′ v*))) **.**
qed
qed

lemma *sym-preproc-addnew-complete*:
assumes $g \in set\ gs$ **and** $lt\ g\ adds_t\ v$
shows *monom-mult 1* (*pp-of-term v* − *lp g*) $g \in set$ (*snd* (*sym-preproc-addnew gs vs fs v*))
using *assms*(*1*)
proof (*induct gs arbitrary*: *vs fs*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons h gs*)
let *?t* = *pp-of-term v* − *lp g*
show *?case*
proof (*cases h* = *g*)
case *True*
show *?thesis*
proof (*simp add*: *True assms*(*2*) *Let-def*)
define *f* **where** *f* = *monom-mult 1 ?t g*
define *ts′* **where** *ts′* = *merge-wrt* ($\succ_t$) *vs* (*keys-to-list* (*tail* (*monom-mult 1 ?t g*)))
have $f \in set$ (*insert-list f fs*) **by** (*simp add*: *set-insert-list*)
with *snd-sym-preproc-addnew-superset* **show** $f \in set$ (*snd* (*sym-preproc-addnew gs ts′* (*insert-list f fs*) *v*)) **..**
qed
next
case *False*
with *Cons*(*2*) **have** $g \in set\ gs$ **by** *simp*
hence ∗: *monom-mult 1 ?t g* ∈ *set* (*snd* (*sym-preproc-addnew gs ts′ fs′ v*)) **for** *ts′ fs′*
by (*rule Cons*(*1*))
show *?thesis* **by** (*simp add*: *Let-def* ∗)
qed
qed

function *sym-preproc-aux* :: (*′t* $\Rightarrow_0$ *′b*::*semiring-1*) *list* ⇒ *′t list* ⇒ (*′t list* × (*′t* $\Rightarrow_0$ *′b*) *list*) ⇒
(*′t list* × (*′t* $\Rightarrow_0$ *′b*) *list*) **where**
sym-preproc-aux gs ks (*vs*, *fs*) =
(*if vs* = [] *then*
(*ks*, *fs*)
else
let v = *ord-term-lin.max-list vs*; *vs′* = *removeAll v vs in*
sym-preproc-aux gs (*ks* @ [*v*]) (*sym-preproc-addnew gs vs′ fs v*)
)
by *pat-completeness auto*
termination proof −
from *ex-dgrad* **obtain** *d*::*′a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
let *?R* = (*sym-preproc-aux-term d*)::(((*′t* $\Rightarrow_0$ *′b*) *list* × *′t list* × *′t list* × (*′t* $\Rightarrow_0$ *′b*) *list*) ×

('t ⇒₀ 'b) *list* × 't *list* × 't *list* × ('t ⇒₀ 'b) *list*) *set*
show *?thesis*
proof
from *dg* **show** *wf ?R* **by** (*rule sym-preproc-aux-term-wf*)
next
fix *gs*::('t ⇒₀ 'b) *list* **and** *ks vs fs v vs'*
assume *vs* ≠ [] **and** *v* = *ord-term-lin.max-list vs* **and** *vs'*: *vs'* = *removeAll v vs*
from *this*(*1*, *2*) **have** *v*: *v* = *ord-term-lin.Max* (*set vs*)
by (*simp add*: *ord-term-lin.max-list-Max*)
obtain *vs0 fs0* **where** *eq*: *sym-preproc-addnew gs vs' fs v* = (*vs0*, *fs0*) **by** *fastforce*
show ((*gs*, *ks* @ [*v*], *sym-preproc-addnew gs vs' fs v*), (*gs*, *ks*, *vs*, *fs*)) ∈ *?R*
proof (*simp add*: *eq sym-preproc-aux-term-def sym-preproc-aux-term1-def sym-preproc-aux-term2-def*, *intro conjI bexI ballI*)
fix *w*
assume *w* ∈ *set vs0*
show $w \prec_t v$
proof (*rule fst-sym-preproc-addnew-less*)
fix *u*
assume *u* ∈ *set vs'*
thus $u \prec_t v$ **unfolding** *vs' v set-removeAll* **using** *ord-term-lin.antisym-conv1* **by** *fastforce*
next
from ‹*w* ∈ *set vs0*› **show** *w* ∈ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*)) **by** (*simp add*: *eq*)
qed
next
from ‹*vs* ≠ []› **show** *v* ∈ *set vs* **by** (*simp add*: *v*)
next
from *dg* **have** *dgrad-set-le d* (*pp-of-term* ‘ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*)))
(*pp-of-term* ‘ (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
by (*rule fst-sym-preproc-addnew-dgrad-set-le*)
moreover have *insert v* (*set vs'*) = *set vs* **by** (*auto simp add*: *vs' v* ‹*vs* ≠ []›)
ultimately show *dgrad-set-le d* (*pp-of-term* ‘ *set vs0*) (*pp-of-term* ‘ (*Keys* (*set gs*) ∪ *set vs*))
by (*simp add*: *eq*)
qed
qed
qed

lemma *sym-preproc-aux-Nil*: *sym-preproc-aux gs ks* ([], *fs*) = (*ks*, *fs*)
by *simp*

lemma *sym-preproc-aux-sorted*:
assumes *sorted-wrt* ($\succ_t$) (*v* # *vs*)
shows *sym-preproc-aux gs ks* (*v* # *vs*, *fs*) = *sym-preproc-aux gs* (*ks* @ [*v*]) (*sym-preproc-addnew gs vs fs v*)
proof −

from *assms* **have** ∗: $u \in set\ vs \Longrightarrow u \prec_t v$ **for** *u* **by** *simp*
have *ord-term-lin.max-list* (*v* # *vs*) = *ord-term-lin.Max* (*set* (*v* # *vs*))
by (*simp add*: *ord-term-lin.max-list-Max del*: *ord-term-lin.max-list.simps*)
also have ... = *v*
proof (*rule ord-term-lin.Max-eqI*)
fix *s*
assume $s \in set\ (v \# vs)$
hence $s = v \vee s \in set\ vs$ **by** *simp*
thus $s \preceq_t v$
proof
assume $s = v$
thus *?thesis* **by** *simp*
next
assume $s \in set\ vs$
hence $s \prec_t v$ **by** (*rule* ∗)
thus *?thesis* **by** *simp*
qed
next
show $v \in set\ (v \# vs)$ **by** *simp*
qed *rule*
finally have *eq1*: *ord-term-lin.max-list* (*v* # *vs*) = *v* **.**
have *eq2*: *removeAll v* (*v* # *vs*) = *vs*
proof (*simp*, *rule removeAll-id*, *rule*)
assume $v \in set\ vs$
hence $v \prec_t v$ **by** (*rule* ∗)
thus *False* **..**
qed
show *?thesis* **by** (*simp only*: *sym-preproc-aux.simps eq1 eq2 Let-def*, *simp*)
qed

lemma *sym-preproc-aux-induct* [*consumes 0*, *case-names base rec*]:
assumes *base*: ⋀*ks fs*. *P ks* [] *fs* (*ks*, *fs*)
and *rec*: ⋀*ks vs fs v vs'*. *vs* ≠ [] ⟹ *v* = *ord-term-lin.Max* (*set vs*) ⟹ *vs'* = *removeAll v vs* ⟹
P (*ks* @ [*v*]) (*fst* (*sym-preproc-addnew gs vs' fs v*)) (*snd* (*sym-preproc-addnew gs vs' fs v*))
(*sym-preproc-aux gs* (*ks* @ [*v*]) (*sym-preproc-addnew gs vs' fs v*)) ⟹
P ks vs fs (*sym-preproc-aux gs* (*ks* @ [*v*]) (*sym-preproc-addnew gs vs' fs v*))
shows *P ks vs fs* (*sym-preproc-aux gs ks* (*vs*, *fs*))
proof −
from *ex-dgrad* **obtain** *d*::'*a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
let *?R* = (*sym-preproc-aux-term d*)::((('$t \Rightarrow_0$ '*b*) *list* × '*t list* × '*t list* × ('$t \Rightarrow_0$ '*b*) *list*) × ('$t \Rightarrow_0$ '*b*) *list* × '*t list* × '*t list* × ('$t \Rightarrow_0$ '*b*) *list*) *set*
define *args* **where** *args* = (*gs*, *ks*, *vs*, *fs*)
from *dg* **have** *wf ?R* **by** (*rule sym-preproc-aux-term-wf*)
hence *fst args* = *gs* ⟹ *P* (*fst* (*snd args*)) (*fst* (*snd* (*snd args*))) (*snd* (*snd* (*snd

args)))
 (*sym-preproc-aux gs* (*fst* (*snd args*)) (*snd* (*snd args*)))
proof *induct*
fix *x*
assume *IH'*: $\bigwedge$*y*. (*y*, *x*) ∈ *sym-preproc-aux-term d* ⟹ *fst y* = *gs* ⟹
P (*fst* (*snd y*)) (*fst* (*snd* (*snd y*))) (*snd* (*snd* (*snd y*)))
(*sym-preproc-aux gs* (*fst* (*snd y*)) (*snd* (*snd y*)))
assume *fst x* = *gs*
then obtain *x0* **where** *x*: *x* = (*gs*, *x0*) **by** (*meson eq-fst-iff*)
obtain *ks x1* **where** *x0*: *x0* = (*ks*, *x1*) **by** (*meson case-prodE case-prodI2*)
obtain *vs fs* **where** *x1*: *x1* = (*vs*, *fs*) **by** (*meson case-prodE case-prodI2*)
from *IH'* **have** *IH*: $\bigwedge$*ks' n*. ((*gs*, *ks'*, *n*), (*gs*, *ks*, *vs*, *fs*)) ∈ *sym-preproc-aux-term d* ⟹
P ks' (*fst n*) (*snd n*) (*sym-preproc-aux gs ks' n*)
unfolding *x x0 x1* **by** *fastforce*
show *P* (*fst* (*snd x*)) (*fst* (*snd* (*snd x*))) (*snd* (*snd* (*snd x*)))
(*sym-preproc-aux gs* (*fst* (*snd x*)) (*snd* (*snd x*)))
proof (*simp add*: *x x0 x1 Let-def*, *intro conjI impI*)
show *P ks* [] *fs* (*ks*, *fs*) **by** (*fact base*)
next
assume *vs* ≠ []
define *v* **where** *v* = *ord-term-lin.max-list vs*
from ‹*vs* ≠ []› **have** *v-alt*: *v* = *ord-term-lin.Max* (*set vs*) **unfolding** *v-def*
by (*rule ord-term-lin.max-list-Max*)
define *vs'* **where** *vs'* = *removeAll v vs*
show *P ks vs fs* (*sym-preproc-aux gs* (*ks* @ [*v*]) (*sym-preproc-addnew gs vs' fs v*))
proof (*rule rec*, *fact* ‹*vs* ≠ []›, *fact v-alt*, *fact vs'-def*)
let *?n* = *sym-preproc-addnew gs vs' fs v*
obtain *vs0 fs0* **where** *eq*: *?n* = (*vs0*, *fs0*) **by** *fastforce*
show *P* (*ks* @ [*v*]) (*fst ?n*) (*snd ?n*) (*sym-preproc-aux gs* (*ks* @ [*v*]) *?n*)
proof (*rule IH*,
simp add: *eq sym-preproc-aux-term-def sym-preproc-aux-term1-def sym-preproc-aux-term2-def*,
intro conjI bexI ballI)
fix *s*
assume *s* ∈ *set vs0*
show $s \prec_t v$
proof (*rule fst-sym-preproc-addnew-less*)
fix *u*
assume *u* ∈ *set vs'*
thus $u \prec_t v$ **unfolding** *vs'-def v-alt set-removeAll* **using** *ord-term-lin.antisym-conv1*
by *fastforce*
next
from ‹*s* ∈ *set vs0*› **show** *s* ∈ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*)) **by** (*simp add*: *eq*)
qed
next
from ‹*vs* ≠ []› **show** *v* ∈ *set vs* **by** (*simp add*: *v-alt*)

next
from *dg* **have** *dgrad-set-le d* (*pp-of-term* ' *set* (*fst* (*sym-preproc-addnew gs vs' fs v*)))
(*pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
by (*rule fst-sym-preproc-addnew-dgrad-set-le*)
moreover have *insert v* (*set vs'*) = *set vs* **by** (*auto simp add*: *vs'-def v-alt* ‹*vs* ≠ []›)
ultimately show *dgrad-set-le d* (*pp-of-term* ' *set vs0*) (*pp-of-term* ' (*Keys* (*set gs*) ∪ *set vs*))
by (*simp add*: *eq*)
qed
qed
qed
qed
thus *?thesis* **by** (*simp add*: *args-def*)
qed

lemma *fst-sym-preproc-aux-sorted-wrt*:
assumes *sorted-wrt* ($\succ_t$) *ks* **and** $\bigwedge$*k v*. *k* ∈ *set ks* ⟹ *v* ∈ *set vs* ⟹ *v* $\prec_t$ *k*
shows *sorted-wrt* ($\succ_t$) (*fst* (*sym-preproc-aux gs ks* (*vs*, *fs*)))
using *assms*
proof (*induct gs ks vs fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
from *base*(*1*) **show** *?case* **by** *simp*
next
case (*rec ks vs fs v vs'*)
from *rec*(*1*) **have** *v* ∈ *set vs* **by** (*simp add*: *rec*(*2*))
from *rec*(*1*) **have** ∗: $\bigwedge$*u*. *u* ∈ *set vs'* ⟹ *u* $\prec_t$ *v* **unfolding** *rec*(*2*, *3*) *set-removeAll*
using *ord-term-lin.antisym-conv3* **by** *force*
show *?case*
proof (*rule rec*(*4*))
show *sorted-wrt* ($\succ_t$) (*ks* @ [*v*])
proof (*simp add*: *sorted-wrt-append rec*(*5*), *rule*)
fix *k*
assume *k* ∈ *set ks*
from *this* ‹*v* ∈ *set vs*› **show** *v* $\prec_t$ *k* **by** (*rule rec*(*6*))
qed
next
fix *k u*
assume *k* ∈ *set* (*ks* @ [*v*]) **and** *u* ∈ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*))
from ∗ *this*(*2*) **have** *u* $\prec_t$ *v* **by** (*rule fst-sym-preproc-addnew-less*)
from ‹*k* ∈ *set* (*ks* @ [*v*])› **have** *k* ∈ *set ks* ∨ *k* = *v* **by** *auto*
thus *u* $\prec_t$ *k*
proof
assume *k* ∈ *set ks*
from *this* ‹*v* ∈ *set vs*› **have** *v* $\prec_t$ *k* **by** (*rule rec*(*6*))
with ‹*u* $\prec_t$ *v*› **show** *?thesis* **by** *simp*
next
assume *k* = *v*

with ‹$u \prec_t v$› **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *fst-sym-preproc-aux-complete*:
assumes *Keys* (*set* (*fs*::($'t \Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*)) = *set ks* ∪ *set vs*
shows *set* (*fst* (*sym-preproc-aux gs ks* (*vs*, *fs*))) = *Keys* (*set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*))))
using *assms*
proof (*induct gs ks vs fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
thus *?case* **by** *simp*
next
case (*rec ks vs fs v vs'*)
from *rec*(*1*) **have** *v* ∈ *set vs* **by** (*simp add*: *rec*(*2*))
hence *eq*: *insert v* (*set vs'*) = *set vs* **by** (*auto simp add*: *rec*(*3*))
also from *rec*(*5*) **have** ... ⊆ *Keys* (*set fs*) **by** *simp*
also from *snd-sym-preproc-addnew-superset* **have** ... ⊆ *Keys* (*set* (*snd* (*sym-preproc-addnew gs vs' fs v*)))
by (*rule Keys-mono*)
finally have ... = ... ∪ (*insert v* (*set vs'*)) **by** *blast*
also have ... = *Keys* (*set fs*) ∪ *insert v* (*set* (*fst* (*sym-preproc-addnew gs vs' fs v*)))
by (*fact Keys-snd-sym-preproc-addnew*)
also have ... = (*set ks* ∪ (*insert v* (*set vs'*))) ∪ (*insert v* (*set* (*fst* (*sym-preproc-addnew gs vs' fs v*))))
by (*simp only*: *rec*(*5*) *eq*)
also have ... = *set* (*ks* @ [*v*]) ∪ (*set vs'* ∪ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*))) **by** *auto*
also from *fst-sym-preproc-addnew-superset* **have** ... = *set* (*ks* @ [*v*]) ∪ *set* (*fst* (*sym-preproc-addnew gs vs' fs v*))
by *blast*
finally show *?case* **by** (*rule rec*(*4*))
qed

lemma *snd-sym-preproc-aux-superset*: *set fs* ⊆ *set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*)))
proof (*induct fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
show *?case* **by** *simp*
next
case (*rec ks vs fs v vs'*)
from *snd-sym-preproc-addnew-superset rec*(*4*) **show** *?case* **by** (*rule subset-trans*)
qed

lemma *in-snd-sym-preproc-auxE*:
assumes *p* ∈ *set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*)))

assumes *1*: *p* ∈ *set fs* ⟹ *thesis*
assumes *2*: ⋀*g t*. *g* ∈ *set gs* ⟹ *p* = *monom-mult 1 t g* ⟹ *thesis*
shows *thesis*
using *assms*
proof (*induct gs ks vs fs arbitrary*: *thesis rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
from *base*(*1*) **have** *p* ∈ *set fs* **by** *simp*
thus *?case* **by** (*rule base*(*2*))
next
case (*rec ks vs fs v vs'*)
from *rec*(*5*) **show** *?case*
proof (*rule rec*(*4*))
assume *p* ∈ *set* (*snd* (*sym-preproc-addnew gs vs' fs v*))
thus *?thesis*
proof (*rule in-snd-sym-preproc-addnewE*)
assume *p* ∈ *set fs*
thus *?thesis* **by** (*rule rec*(*6*))
next
fix *g s*
assume *g* ∈ *set gs* **and** *p* = *monom-mult 1 s g*
thus *?thesis* **by** (*rule rec*(*7*))
qed
next
fix *g t*
assume *g* ∈ *set gs* **and** *p* = *monom-mult 1 t g*
thus *?thesis* **by** (*rule rec*(*7*))
qed
qed

lemma *snd-sym-preproc-aux-pmdl*:
pmdl (*set gs* ∪ *set* (*snd* (*sym-preproc-aux gs ks* (*ts*, *fs*)))) = *pmdl* (*set gs* ∪ *set fs*)
proof (*induct fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
show *?case* **by** *simp*
next
case (*rec ks vs fs v vs'*)
from *rec*(*4*) *sym-preproc-addnew-pmdl* **show** *?case* **by** (*rule trans*)
qed

lemma *snd-sym-preproc-aux-dgrad-set-le*:
assumes *dickson-grading d* **and** *set vs* ⊆ *Keys* (*set* (*fs*::(*'t* $\Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*))
shows *dgrad-set-le d* (*pp-of-term* ‘ *Keys* (*set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*))))) (*pp-of-term* ‘ *Keys* (*set gs* ∪ *set fs*))
using *assms*(*2*)
proof (*induct fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
show *?case* **by** (*rule dgrad-set-le-subset*, *simp add*: *Keys-Un image-Un*)

next
  case (*rec ks vs fs v vs'*)
  let *?n* = *sym-preproc-addnew gs vs' fs v*
  from *rec*(*1*) **have** *v* ∈ *set vs* **by** (*simp add*: *rec*(*2*))
  hence *set-vs*: *insert v* (*set vs'*) = *set vs* **by** (*auto simp add*: *rec*(*3*))
  from *rec*(*5*) **have** *eq*: *Keys* (*set fs*) ∪ (*Keys* (*set gs*) ∪ *set vs*) = *Keys* (*set gs*) ∪ *Keys* (*set fs*)
    by *blast*
  have *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set* (*snd* (*sym-preproc-aux gs* (*ks* @ [*v*]) *?n*))))
                        (*pp-of-term* ' *Keys* (*set gs* ∪ *set* (*snd ?n*)))
  proof (*rule rec*(*4*))
    have *set* (*fst ?n*) ⊆ *Keys* (*set* (*snd ?n*)) ∪ *insert v* (*set vs'*)
      by (*simp only*: *Keys-snd-sym-preproc-addnew*, *blast*)
    also have ... = *Keys* (*set* (*snd ?n*)) ∪ (*set vs*) **by** (*simp only*: *set-vs*)
    also have ... ⊆ *Keys* (*set* (*snd ?n*))
    proof –
      {
        fix *u*
        assume *u* ∈ *set vs*
        with *rec*(*5*) **have** *u* ∈ *Keys* (*set fs*) **..**
        then obtain *f* **where** *f* ∈ *set fs* **and** *u* ∈ *keys f* **by** (*rule in-KeysE*)
        from *this*(*1*) *snd-sym-preproc-addnew-superset* **have** *f* ∈ *set* (*snd ?n*) **..**
        with ‹*u* ∈ *keys f*› **have** *u* ∈ *Keys* (*set* (*snd ?n*)) **by** (*rule in-KeysI*)
      }
      thus *?thesis* **by** *auto*
    qed
    finally show *set* (*fst ?n*) ⊆ *Keys* (*set* (*snd ?n*)) **.**
  qed
  also have *dgrad-set-le d* ... (*pp-of-term* ' *Keys* (*set gs* ∪ *set fs*))
  proof (*simp only*: *image-Un Keys-Un dgrad-set-le-Un*, *rule*)
    show *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set gs*)) (*pp-of-term* ' *Keys* (*set gs*) ∪ *pp-of-term* ' *Keys* (*set fs*))
      by (*rule dgrad-set-le-subset*, *simp*)
  next
    have *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set* (*snd ?n*))) (*pp-of-term* ' (*Keys* (*set fs*) ∪ *insert v* (*set* (*fst ?n*))))
      by (*rule dgrad-set-le-subset*, *auto simp only*: *Keys-snd-sym-preproc-addnew*[*symmetric*])
    also have *dgrad-set-le d* ... (*pp-of-term* ' *Keys* (*set fs*) ∪ *pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
    proof (*simp only*: *dgrad-set-le-Un image-Un*, *rule*)
      show *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set fs*))
            (*pp-of-term* ' *Keys* (*set fs*) ∪ (*pp-of-term* ' *Keys* (*set gs*) ∪ *pp-of-term* ' *insert v* (*set vs'*)))
        by (*rule dgrad-set-le-subset*, *blast*)
    next
      have *dgrad-set-le d* (*pp-of-term* ' {*v*}) (*pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
        by (*rule dgrad-set-le-subset*, *simp*)

moreover from *assms*(*1*) **have** *dgrad-set-le d* (*pp-of-term* ' *set* (*fst ?n*)) (*pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
by (*rule fst-sym-preproc-addnew-dgrad-set-le*)
ultimately have *dgrad-set-le d* (*pp-of-term* ' ({*v*} ∪ *set* (*fst ?n*))) (*pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
by (*simp only*: *dgrad-set-le-Un image-Un*)
also have *dgrad-set-le d* (*pp-of-term* ' (*Keys* (*set gs*) ∪ *insert v* (*set vs'*)))
(*pp-of-term* ' (*Keys* (*set fs*) ∪ (*Keys* (*set gs*) ∪ *insert v* (*set vs'*))))
by (*rule dgrad-set-le-subset*, *blast*)
finally show *dgrad-set-le d* (*pp-of-term* ' *insert v* (*set* (*fst ?n*)))
(*pp-of-term* ' *Keys* (*set fs*) ∪ (*pp-of-term* ' *Keys* (*set gs*) ∪ *pp-of-term* ' *insert v* (*set vs'*)))
by (*simp add*: *image-Un*)
qed
finally show *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set* (*snd ?n*))) (*pp-of-term* ' *Keys* (*set gs*) ∪ *pp-of-term* ' *Keys* (*set fs*))
by (*simp only*: *set-vs eq*, *metis eq image-Un*)
qed
finally show *?case* **.**
qed

lemma *components-snd-sym-preproc-aux-subset*:
assumes *set vs* ⊆ *Keys* (*set* (*fs*::(*'t* $\Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*))
shows *component-of-term* ' *Keys* (*set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*)))) ⊆
component-of-term ' *Keys* (*set gs* ∪ *set fs*)
using *assms*
proof (*induct fs rule*: *sym-preproc-aux-induct*)
case (*base ks fs*)
show *?case* **by** (*simp add*: *Keys-Un image-Un*)
next
case (*rec ks vs fs v vs'*)
let *?n* = *sym-preproc-addnew gs vs' fs v*
from *rec*(*1*) **have** *v* ∈ *set vs* **by** (*simp add*: *rec*(*2*))
hence *set-vs*: *insert v* (*set vs'*) = *set vs* **by** (*auto simp add*: *rec*(*3*))
from *rec*(*5*) **have** *eq*: *Keys* (*set fs*) ∪ (*Keys* (*set gs*) ∪ *set vs*) = *Keys* (*set gs*) ∪ *Keys* (*set fs*)
by *blast*
have *component-of-term* ' *Keys* (*set* (*snd* (*sym-preproc-aux gs* (*ks* @ [*v*]) *?n*))) ⊆
component-of-term ' *Keys* (*set gs* ∪ *set* (*snd ?n*))
proof (*rule rec*(*4*))
have *set* (*fst ?n*) ⊆ *Keys* (*set* (*snd ?n*)) ∪ *insert v* (*set vs'*)
by (*simp only*: *Keys-snd-sym-preproc-addnew*, *blast*)
also have ... = *Keys* (*set* (*snd ?n*)) ∪ (*set vs*) **by** (*simp only*: *set-vs*)
also have ... ⊆ *Keys* (*set* (*snd ?n*))
proof −
{
fix *u*
assume *u* ∈ *set vs*

with *rec*(*5*) **have** $u \in$ *Keys* (*set fs*) **..**
then obtain *f* **where** $f \in$ *set fs* **and** $u \in$ *keys f* **by** (*rule in-KeysE*)
from *this*(*1*) *snd-sym-preproc-addnew-superset* **have** $f \in$ *set* (*snd ?n*) **..**
with ‹$u \in$ *keys f*› **have** $u \in$ *Keys* (*set* (*snd ?n*)) **by** (*rule in-KeysI*)
}
thus *?thesis* **by** *auto*
qed
finally show *set* (*fst ?n*) $\subseteq$ *Keys* (*set* (*snd ?n*)) **.**
qed
also have ... $\subseteq$ *component-of-term* ' *Keys* (*set gs* $\cup$ *set fs*)
proof (*simp only*: *image-Un Keys-Un Un-subset-iff*, *rule*, *fact Un-upper1*)
have *component-of-term* ' *Keys* (*set* (*snd ?n*)) $\subseteq$ *component-of-term* ' (*Keys* (*set fs*) $\cup$ *insert v* (*set* (*fst ?n*)))
by (*auto simp only*: *Keys-snd-sym-preproc-addnew*[*symmetric*])
also have ... $\subseteq$ *component-of-term* ' *Keys* (*set fs*) $\cup$ *component-of-term* ' (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*))
proof (*simp only*: *Un-subset-iff image-Un*, *rule*, *fact Un-upper1*)
have *component-of-term* ' $\{v\}$ $\subseteq$ *component-of-term* ' (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*))
by *simp*
moreover have *component-of-term* ' *set* (*fst ?n*) $\subseteq$ *component-of-term* ' (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*))
by (*rule components-fst-sym-preproc-addnew-subset*)
ultimately have *component-of-term* ' ($\{v\}$ $\cup$ *set* (*fst ?n*)) $\subseteq$ *component-of-term* ' (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*))
by (*simp only*: *Un-subset-iff image-Un*)
also have *component-of-term* ' (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*)) $\subseteq$
component-of-term ' (*Keys* (*set fs*) $\cup$ (*Keys* (*set gs*) $\cup$ *insert v* (*set vs'*)))
by *blast*
finally show *component-of-term* ' *insert v* (*set* (*fst ?n*)) $\subseteq$
component-of-term ' *Keys* (*set fs*) $\cup$
(*component-of-term* ' *Keys* (*set gs*) $\cup$ *component-of-term* ' *insert v* (*set vs'*))
by (*simp add*: *image-Un*)
qed
finally show *component-of-term* ' *Keys* (*set* (*snd ?n*)) $\subseteq$
component-of-term ' *Keys* (*set gs*) $\cup$ *component-of-term* ' *Keys* (*set fs*)
by (*simp only*: *set-vs eq*, *metis eq image-Un*)
qed
finally show *?case* **.**
qed

lemma *snd-sym-preproc-aux-complete*:
assumes $\bigwedge u'\ g'.\ u' \in$ *Keys* (*set fs*) $\Longrightarrow u' \notin$ *set vs* $\Longrightarrow g' \in$ *set gs* $\Longrightarrow$ *lt* g' $adds_t$ u' $\Longrightarrow$
monom-mult 1 (*pp-of-term* u' $-$ *lp* g') $g' \in$ *set fs*
assumes $u \in$ *Keys* (*set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*)))) **and** $g \in$ *set gs*

```
and lt g adds_t u
  shows monom-mult (1::'b::semiring-1-no-zero-divisors) (pp-of-term u − lp g) g
∈
          set (snd (sym-preproc-aux gs ks (vs, fs)))
  using assms
proof (induct fs rule: sym-preproc-aux-induct)
  case (base ks fs)
  from base(2) have u ∈ Keys (set fs) by simp
  from this - base(3, 4) have monom-mult 1 (pp-of-term u − lp g) g ∈ set fs
  proof (rule base(1))
    show u ∉ set [] by simp
  qed
  thus ?case by simp
next
  case (rec ks vs fs v vs')
  from rec(1) have v ∈ set vs by (simp add: rec(2))
  hence set-ts: set vs = insert v (set vs') by (auto simp add: rec(3))

  let ?n = sym-preproc-addnew gs vs' fs v
  from - rec(6, 7, 8) show ?case
  proof (rule rec(4))
    fix v' g'
    assume v' ∈ Keys (set (snd ?n)) and v' ∉ set (fst ?n) and g' ∈ set gs and lt
g' adds_t v'
    from this(1) Keys-snd-sym-preproc-addnew have v' ∈ Keys (set fs) ∪ insert v
(set (fst ?n))
      by blast
    with ‹v' ∉ set (fst ?n)› have disj: v' ∈ Keys (set fs) ∨ v' = v by blast
    show monom-mult 1 (pp-of-term v' − lp g') g' ∈ set (snd ?n)
    proof (cases v' = v)
      case True
      from ‹g' ∈ set gs› ‹lt g' adds_t v'› show ?thesis
        unfolding True by (rule sym-preproc-addnew-complete)
    next
      case False
      with disj have v' ∈ Keys (set fs) by simp
      moreover have v' ∉ set vs
      proof
        assume v' ∈ set vs
        hence v' ∈ set vs' using False by (simp add: rec(3))
        with fst-sym-preproc-addnew-superset have v' ∈ set (fst ?n) ..
        with ‹v' ∉ set (fst ?n)› show False ..
      qed
      ultimately have monom-mult 1 (pp-of-term v' − lp g') g' ∈ set fs
        using ‹g' ∈ set gs› ‹lt g' adds_t v'› by (rule rec(5))
      with snd-sym-preproc-addnew-superset show ?thesis ..
    qed
  qed
qed
```

definition *sym-preproc* :: ($'t \Rightarrow_0 'b$::*semiring-1*) *list* ⇒ ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t$ *list* ×
($'t \Rightarrow_0 'b$) *list*)
  where *sym-preproc gs fs* = *sym-preproc-aux gs* [] (*Keys-to-list fs*, *fs*)

lemma *sym-preproc-Nil* [*simp*]: *sym-preproc gs* [] = ([], [])
  by (*simp add*: *sym-preproc-def*)

lemma *fst-sym-preproc*:
 fst (*sym-preproc gs fs*) = *Keys-to-list* (*snd* (*sym-preproc gs* (*fs*::($'t \Rightarrow_0 'b$::*semiring-1-no-zero-divisors*) *list*)))
proof −
  let *?a* = *fst* (*sym-preproc gs fs*)
  let *?b* = *Keys-to-list* (*snd* (*sym-preproc gs fs*))
  have *antisymp* ($\succ_t$) **unfolding** *antisymp-def* **by** *fastforce*
  have *irreflp* ($\succ_t$) **by** (*simp add*: *irreflp-def*)
  moreover have *transp* ($\succ_t$) **unfolding** *transp-def* **by** *fastforce*
  moreover have *s1*: *sorted-wrt* ($\succ_t$) *?a* **unfolding** *sym-preproc-def*
    by (*rule fst-sym-preproc-aux-sorted-wrt*, *simp-all*)
  ultimately have *d1*: *distinct ?a* **by** (*rule distinct-sorted-wrt-irrefl*)
  have *s2*: *sorted-wrt* ($\succ_t$) *?b* **by** (*fact Keys-to-list-sorted-wrt*)
 with ‹*irreflp* ($\succ_t$)› ‹*transp* ($\succ_t$)› **have** *d2*: *distinct ?b* **by** (*rule distinct-sorted-wrt-irrefl*)
  from ‹*antisymp* ($\succ_t$)› *s1 d1 s2 d2* **show** *?thesis*
  proof (*rule sorted-wrt-distinct-set-unique*)
    show *set ?a* = *set ?b* **unfolding** *set-Keys-to-list sym-preproc-def*
      by (*rule fst-sym-preproc-aux-complete*, *simp add*: *set-Keys-to-list*)
  qed
qed

lemma *snd-sym-preproc-superset*: *set fs* ⊆ *set* (*snd* (*sym-preproc gs fs*))
  by (*simp only*: *sym-preproc-def snd-conv*, *fact snd-sym-preproc-aux-superset*)

lemma *in-snd-sym-preprocE*:
  assumes *p* ∈ *set* (*snd* (*sym-preproc gs fs*))
  assumes *1*: *p* ∈ *set fs* ⟹ *thesis*
  assumes *2*: ⋀*g t*. *g* ∈ *set gs* ⟹ *p* = *monom-mult 1 t g* ⟹ *thesis*
  shows *thesis*
 using *assms* **unfolding** *sym-preproc-def snd-conv* **by** (*rule in-snd-sym-preproc-auxE*)

lemma *snd-sym-preproc-pmdl*: *pmdl* (*set gs* ∪ *set* (*snd* (*sym-preproc gs fs*))) =
pmdl (*set gs* ∪ *set fs*)
  unfolding *sym-preproc-def snd-conv* **by** (*fact snd-sym-preproc-aux-pmdl*)

lemma *snd-sym-preproc-dgrad-set-le*:
  assumes *dickson-grading d*
  shows *dgrad-set-le d* (*pp-of-term* ‘ *Keys* (*set* (*snd* (*sym-preproc gs fs*))))
          (*pp-of-term* ‘ *Keys* (*set gs* ∪ *set* (*fs*::($'t \Rightarrow_0 'b$::*semiring-1-no-zero-divisors*) *list*)))
  unfolding *sym-preproc-def snd-conv* **using** *assms*

proof (*rule snd-sym-preproc-aux-dgrad-set-le*)
  show *set* (*Keys-to-list fs*) ⊆ *Keys* (*set fs*) **by** (*simp add*: *set-Keys-to-list*)
qed

corollary *snd-sym-preproc-dgrad-p-set-le*:
  assumes *dickson-grading d*
  shows *dgrad-p-set-le d* (*set* (*snd* (*sym-preproc gs fs*))) (*set gs* ∪ *set* (*fs*::(*'t* $\Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*))
  unfolding *dgrad-p-set-le-def*
proof −
  from *assms* **show** *dgrad-set-le d* (*pp-of-term* ' *Keys* (*set* (*snd* (*sym-preproc gs fs*)))) (*pp-of-term* ' *Keys* (*set gs* ∪ *set fs*))
    by (*rule snd-sym-preproc-dgrad-set-le*)
qed

lemma *components-snd-sym-preproc-subset*:
  component-of-term ' *Keys* (*set* (*snd* (*sym-preproc gs fs*))) ⊆
    component-of-term ' *Keys* (*set gs* ∪ *set* (*fs*::(*'t* $\Rightarrow_0$ *'b*::*semiring-1-no-zero-divisors*) *list*))
  unfolding *sym-preproc-def snd-conv*
  by (*rule components-snd-sym-preproc-aux-subset*, *simp add*: *set-Keys-to-list*)

lemma *snd-sym-preproc-complete*:
  assumes *v* ∈ *Keys* (*set* (*snd* (*sym-preproc gs fs*))) **and** *g* ∈ *set gs* **and** *lt g* $adds_t$ *v*
  shows *monom-mult* (*1*::*'b*::*semiring-1-no-zero-divisors*) (*pp-of-term v* − *lp g*) *g* ∈ *set* (*snd* (*sym-preproc gs fs*))
  using - *assms* **unfolding** *sym-preproc-def snd-conv*
proof (*rule snd-sym-preproc-aux-complete*)
  fix *u'* **and** *g'*::*'t* $\Rightarrow_0$ *'b*
  assume *u'* ∈ *Keys* (*set fs*) **and** *u'* ∉ *set* (*Keys-to-list fs*)
  thus *monom-mult 1* (*pp-of-term u'* − *lp g'*) *g'* ∈ *set fs* **by** (*simp add*: *set-Keys-to-list*)
qed

end

16.2 *lin-red*

context *ordered-term*
begin

definition *lin-red* :: (*'t* $\Rightarrow_0$ *'b*::*field*) *set* ⇒ (*'t* $\Rightarrow_0$ *'b*) ⇒ (*'t* $\Rightarrow_0$ *'b*) ⇒ *bool*
  where *lin-red F p q* ≡ (∃*f*∈*F*. *red-single p q f 0*)

lin-red is a restriction of *red*, where the reductor (*f*) may only be multiplied by a constant factor, i. e. where the power-product is *0*.

lemma *lin-redI*:
  assumes *f* ∈ *F* **and** *red-single p q f 0*
  shows *lin-red F p q*

```
  unfolding lin-red-def using assms ..

lemma lin-redE:
  assumes lin-red F p q
  obtains f::'t ⇒₀ 'b::field where f ∈ F and red-single p q f 0
proof −
  from assms obtain f where f ∈ F and t: red-single p q f 0 unfolding lin-red-def
by blast
  thus ?thesis ..
qed

lemma lin-red-imp-red:
  assumes lin-red F p q
  shows red F p q
proof −
  from assms obtain f where f ∈ F and red-single p q f 0 by (rule lin-redE)
  thus ?thesis by (rule red-setI)
qed

lemma lin-red-Un: lin-red (F ∪ G) p q = (lin-red F p q ∨ lin-red G p q)
proof
  assume lin-red (F ∪ G) p q
  then obtain f where f ∈ F ∪ G and r: red-single p q f 0 by (rule lin-redE)
  from this(1) show lin-red F p q ∨ lin-red G p q
  proof
    assume f ∈ F
    from this r have lin-red F p q by (rule lin-redI)
    thus ?thesis ..
  next
    assume f ∈ G
    from this r have lin-red G p q by (rule lin-redI)
    thus ?thesis ..
  qed
next
  assume lin-red F p q ∨ lin-red G p q
  thus lin-red (F ∪ G) p q
  proof
    assume lin-red F p q
    then obtain f where f ∈ F and r: red-single p q f 0 by (rule lin-redE)
    from this(1) have f ∈ F ∪ G by simp
    from this r show ?thesis by (rule lin-redI)
  next
    assume lin-red G p q
    then obtain g where g ∈ G and r: red-single p q g 0 by (rule lin-redE)
    from this(1) have g ∈ F ∪ G by simp
    from this r show ?thesis by (rule lin-redI)
  qed
qed
```

lemma *lin-red-imp-red-rtrancl*:
assumes (*lin-red F*)** *p q*
shows (*red F*)** *p q*
using *assms*
proof *induct*
case *base*
show *?case* **..**
next
case (*step y z*)
from *step*(*2*) **have** *red F y z* **by** (*rule lin-red-imp-red*)
with *step*(*3*) **show** *?case* **..**
qed

lemma *phull-closed-lin-red*:
assumes *phull B* ⊆ *phull A* **and** *p* ∈ *phull A* **and** *lin-red B p q*
shows *q* ∈ *phull A*
proof −
from *assms*(*3*) **obtain** *f* **where** *f* ∈ *B* **and** *red-single p q f 0* **by** (*rule lin-redE*)
hence *q*: *q* = *p* − (*lookup p* (*lt f*) / *lc f*) · *f*
by (*simp add*: *red-single-def term-simps map-scale-eq-monom-mult*)
have *q* − *p* ∈ *phull B*
by (*simp add*: *q*, *rule phull.span-neg*, *rule phull.span-scale*, *rule phull.span-base*, *fact* ‹*f* ∈ *B*›)
with *assms*(*1*) **have** *q* − *p* ∈ *phull A* **..**
from *this assms*(*2*) **have** (*q* − *p*) + *p* ∈ *phull A* **by** (*rule phull.span-add*)
thus *?thesis* **by** *simp*
qed

16.3 Reduction

definition *Macaulay-red* :: *'t list* ⇒ (*'t* $\Rightarrow_0$ *'b*) *list* ⇒ (*'t* $\Rightarrow_0$ *'b*::*field*) *list*
where *Macaulay-red vs fs* =
(*let lts* = *map lt* (*filter* (λ*p*. *p* ≠ *0*) *fs*) *in*
filter (λ*p*. *p* ≠ *0* ∧ *lt p* ∉ *set lts*) (*mat-to-polys vs* (*row-echelon* (*polys-to-mat vs fs*)))
)

Macaulay-red vs fs auto-reduces (w. r. t. *lin-red*) the given list *fs* and returns those non-zero polynomials whose leading terms are not in *lt-set* (*set fs*). Argument *vs* is expected to be *Keys-to-list fs*; this list is passed as an argument to *Macaulay-red*, because it can be efficiently computed by symbolic preprocessing.

lemma *Macaulay-red-alt*:
Macaulay-red (*Keys-to-list fs*) *fs* = *filter* (λ*p*. *lt p* ∉ *lt-set* (*set fs*)) (*Macaulay-list fs*)
proof −
have {*x* ∈ *set fs*. *x* ≠ *0*} = *set fs* − {*0*} **by** *blast*
thus *?thesis* **by** (*simp add*: *Macaulay-red-def Macaulay-list-def Macaulay-mat-def lt-set-def Let-def*)

qed

lemma *set-Macaulay-red*:
set (*Macaulay-red* (*Keys-to-list fs*) *fs*) = *set* (*Macaulay-list fs*) − {*p*. *lt p* ∈ *lt-set* (*set fs*)}
by (*auto simp add*: *Macaulay-red-alt*)

lemma *Keys-Macaulay-red*: *Keys* (*set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)) ⊆ *Keys* (*set fs*)
proof −
have *Keys* (*set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)) ⊆ *Keys* (*set* (*Macaulay-list fs*))
unfolding *set-Macaulay-red* **by** (*fact Keys-minus*)
also have ... ⊆ *Keys* (*set fs*) **by** (*fact Keys-Macaulay-list*)
finally show *?thesis* **.**
qed

end

context *gd-term*
begin

lemma *Macaulay-red-reducible*:
assumes *f* ∈ *phull* (*set fs*) **and** *F* ⊆ *set fs* **and** *lt-set F* = *lt-set* (*set fs*)
shows (*lin-red* (*F* ∪ *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)))** *f 0*
proof −
define *A* **where** *A* = *F* ∪ *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)

have *phull-A*: *phull A* ⊆ *phull* (*set fs*)
proof (*rule phull.span-subset-spanI*, *simp add*: *A-def*, *rule*)
have *F* ⊆ *phull F* **by** (*rule phull.span-superset*)
also from *assms*(*2*) **have** ... ⊆ *phull* (*set fs*) **by** (*rule phull.span-mono*)
finally show *F* ⊆ *phull* (*set fs*) **.**
next
have *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*) ⊆ *set* (*Macaulay-list fs*)
by (*auto simp add*: *set-Macaulay-red*)
also have ... ⊆ *phull* (*set* (*Macaulay-list fs*)) **by** (*rule phull.span-superset*)
also have ... = *phull* (*set fs*) **by** (*rule phull-Macaulay-list*)
finally show *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*) ⊆ *phull* (*set fs*) **.**
qed

have *lt-A*: *p* ∈ *phull* (*set fs*) ⟹ *p* ≠ *0* ⟹ (⋀*g*. *g* ∈ *A* ⟹ *g* ≠ *0* ⟹ *lt g* = *lt p* ⟹ *thesis*) ⟹ *thesis*
for *p thesis*
proof −
assume *p* ∈ *phull* (*set fs*) **and** *p* ≠ *0*
then obtain *g* **where** *g-in*: *g* ∈ *set* (*Macaulay-list fs*) **and** *g* ≠ *0* **and** *lt p* = *lt g*
by (*rule Macaulay-list-lt*)

assume ∗: $\bigwedge g.\ g \in A \Longrightarrow g \neq 0 \Longrightarrow lt\ g = lt\ p \Longrightarrow thesis$
show *?thesis*
proof (*cases g* ∈ *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*))
case *True*
hence *g* ∈ *A* **by** (*simp add*: *A-def*)
from *this* ‹*g* ≠ *0*› ‹*lt p* = *lt g*›[*symmetric*] **show** *?thesis* **by** (*rule* ∗)
next
case *False*
with *g-in* **have** *lt g* ∈ *lt-set* (*set fs*) **by** (*simp add*: *set-Macaulay-red*)
also have ... = *lt-set F* **by** (*simp only*: *assms*(*3*))
finally obtain *g'* **where** *g'* ∈ *F* **and** *g'* ≠ *0* **and** *lt g'* = *lt g* **by** (*rule lt-setE*)
from *this*(*1*) **have** *g'* ∈ *A* **by** (*simp add*: *A-def*)
moreover note ‹*g'* ≠ *0*›
moreover have *lt g'* = *lt p* **by** (*simp only*: ‹*lt p* = *lt g*› ‹*lt g'* = *lt g*›)
ultimately show *?thesis* **by** (*rule* ∗)
qed
qed

from *assms*(*2*) *finite-set* **have** *finite F* **by** (*rule finite-subset*)
from *this finite-set* **have** *fin-A*: *finite A* **unfolding** *A-def* **by** (*rule finite-UnI*)

from *ex-dgrad* **obtain** *d*::*'a* ⇒ *nat* **where** *dg*: *dickson-grading d* **..**
from *fin-A* **have** *finite* (*insert f A*) **..**
then obtain *m* **where** *insert f A* ⊆ *dgrad-p-set d m* **by** (*rule dgrad-p-set-exhaust*)
hence *A-sub*: *A* ⊆ *dgrad-p-set d m* **and** *f* ∈ *dgrad-p-set d m* **by** *simp-all*
from *dg* **have** *wfP* (*dickson-less-p d m*) **by** (*rule wf-dickson-less-p*)
from *this assms*(*1*) ‹*f* ∈ *dgrad-p-set d m*› **show** (*lin-red A*)** *f 0*
proof (*induct f*)
fix *p*
assume *IH*: $\bigwedge q.$ *dickson-less-p d m q p* ⟹ *q* ∈ *phull* (*set fs*) ⟹ *q* ∈ *dgrad-p-set d m* ⟹
(*lin-red A*)** *q 0*
and *p* ∈ *phull* (*set fs*) **and** *p* ∈ *dgrad-p-set d m*
show (*lin-red A*)** *p 0*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*p* ∈ *phull* (*set fs*)› **obtain** *g* **where** *g* ∈ *A* **and** *g* ≠ *0* **and** *lt g* = *lt p* **by** (*rule lt-A*)
define *q* **where** *q* = *p* − *monom-mult* (*lc p* / *lc g*) *0 g*
from ‹*g* ∈ *A*› **have** *lr*: *lin-red A p q*
proof (*rule lin-redI*)
show *red-single p q g 0*
by (*simp add*: *red-single-def* ‹*lt g* = *lt p*› *lc-def*[*symmetric*] *q-def* ‹*g* ≠ *0*› *lc-not-0*[*OF False*] *term-simps*)
qed
moreover have (*lin-red A*)** *q 0*

```
      proof −
        from lr have red: red A p q by (rule lin-red-imp-red)
        with dg A-sub ‹p ∈ dgrad-p-set d m› have q ∈ dgrad-p-set d m by (rule
dgrad-p-set-closed-red)
        moreover from red have q ≺p p by (rule red-ord)
        ultimately have dickson-less-p d m q p using ‹p ∈ dgrad-p-set d m›
          by (simp add: dickson-less-p-def)
        moreover from phull-A ‹p ∈ phull (set fs)› lr have q ∈ phull (set fs)
          by (rule phull-closed-lin-red)
        ultimately show ?thesis using ‹q ∈ dgrad-p-set d m› by (rule IH)
      qed
      ultimately show ?thesis by fastforce
    qed
  qed
qed

primrec pdata-pairs-to-list :: ('t, 'b::field, 'c) pdata-pair list ⇒ ('t ⇒0 'b) list
where
  pdata-pairs-to-list [] = []|
  pdata-pairs-to-list (p # ps) =
    (let f = fst (fst p); g = fst (snd p); lf = lp f; lg = lp g; l = lcs lf lg in
      (monom-mult (1 / lc f) (l − lf) f) # (monom-mult (1 / lc g) (l − lg) g) #
      (pdata-pairs-to-list ps)
    )

lemma in-pdata-pairs-to-listI1:
  assumes (f, g) ∈ set ps
  shows monom-mult (1 / lc (fst f)) ((lcs (lp (fst f)) (lp (fst g))) − (lp (fst f)))
            (fst f) ∈ set (pdata-pairs-to-list ps) (is ?m ∈ -)
  using assms
proof (induct ps)
  case Nil
  thus ?case by simp
next
  case (Cons p ps)
  from Cons(2) have p = (f, g) ∨ (f, g) ∈ set ps by auto
  thus ?case
  proof
    assume p = (f, g)
    show ?thesis by (simp add: ‹p = (f, g)› Let-def)
  next
    assume (f, g) ∈ set ps
    hence ?m ∈ set (pdata-pairs-to-list ps) by (rule Cons(1))
    thus ?thesis by (simp add: Let-def)
  qed
qed

lemma in-pdata-pairs-to-listI2:
  assumes (f, g) ∈ set ps
```

shows *monom-mult* (*1* / *lc* (*fst g*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst g*)))
(*fst g*) ∈ *set* (*pdata-pairs-to-list ps*) (**is** *?m* ∈ -)
using *assms*
proof (*induct ps*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons p ps*)
from *Cons*(*2*) **have** *p* = (*f*, *g*) ∨ (*f*, *g*) ∈ *set ps* **by** *auto*
thus *?case*
proof
assume *p* = (*f*, *g*)
show *?thesis* **by** (*simp add*: ‹*p* = (*f*, *g*)› *Let-def*)
next
assume (*f*, *g*) ∈ *set ps*
hence *?m* ∈ *set* (*pdata-pairs-to-list ps*) **by** (*rule Cons*(*1*))
thus *?thesis* **by** (*simp add*: *Let-def*)
qed
qed

lemma *in-pdata-pairs-to-listE*:
assumes *h* ∈ *set* (*pdata-pairs-to-list ps*)
obtains *f g* **where** (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *h* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
using *assms*
proof (*induct ps arbitrary*: *thesis*)
case *Nil*
from *Nil*(*2*) **show** *?case* **by** *simp*
next
case (*Cons p ps*)
let *?f* = *fst* (*fst p*)
let *?g* = *fst* (*snd p*)
let *?lf* = *lp ?f*
let *?lg* = *lp ?g*
let *?l* = *lcs ?lf ?lg*
from *Cons*(*3*) **have** *h* = *monom-mult* (*1* / *lc ?f*) (*?l* − *?lf*) *?f* ∨ *h* = *monom-mult* (*1* / *lc ?g*) (*?l* − *?lg*) *?g* ∨
h ∈ *set* (*pdata-pairs-to-list ps*)
by (*simp add*: *Let-def*)
thus *?case*
proof (*elim disjE*)
assume *h*: *h* = *monom-mult* (*1* / *lc ?f*) (*?l* − *?lf*) *?f*
have (*fst p*, *snd p*) ∈ *set* (*p* # *ps*) **by** *simp*
hence (*fst p*, *snd p*) ∈ *set* (*p* # *ps*) ∨ (*snd p*, *fst p*) ∈ *set* (*p* # *ps*) **..**
from *this h* **show** *?thesis* **by** (*rule Cons*(*2*))
next
assume *h*: *h* = *monom-mult* (*1* / *lc ?g*) (*?l* − *?lg*) *?g*
have (*fst p*, *snd p*) ∈ *set* (*p* # *ps*) **by** *simp*

hence (*snd p*, *fst p*) ∈ *set* (*p* # *ps*) ∨ (*fst p*, *snd p*) ∈ *set* (*p* # *ps*) **..**
moreover from *h* **have** *h* = *monom-mult* (*1* / *lc ?g*) ((*lcs ?lg ?lf*) − *?lg*) *?g*
by (*simp only*: *lcs-comm*)
ultimately show *?thesis* **by** (*rule Cons*(*2*))
next
assume *h-in*: *h* ∈ *set* (*pdata-pairs-to-list ps*)
obtain *f g* **where** (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *h*: *h* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
by (*rule Cons*(*1*), *assumption*, *intro h-in*)
from *this*(*1*) **have** (*f*, *g*) ∈ *set* (*p* # *ps*) ∨ (*g*, *f*) ∈ *set* (*p* # *ps*) **by** *auto*
from *this h* **show** *?thesis* **by** (*rule Cons*(*2*))
qed
qed

definition *f4-red-aux* :: (′*t*, ′*b*::*field*, ′*c*) *pdata list* ⇒ (′*t*, ′*b*, ′*c*) *pdata-pair list* ⇒ (′*t* $\Rightarrow_0$ ′*b*) *list*
where *f4-red-aux bs ps* =
(*let aux* = *sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*) *in Macaulay-red* (*fst aux*) (*snd aux*))

f4-red-aux only takes two arguments, since it does not distinguish between those elements of the current basis that are known to be a Gröbner basis (called *gs* in *Groebner-Bases.Algorithm-Schema*) and the remaining ones.

lemma *f4-red-aux-not-zero*: *0* ∉ *set* (*f4-red-aux bs ps*)
by (*simp add*: *f4-red-aux-def Let-def fst-sym-preproc set-Macaulay-red set-Macaulay-list*)

lemma *f4-red-aux-irredudible*:
assumes *h* ∈ *set* (*f4-red-aux bs ps*) **and** *b* ∈ *set bs* **and** *fst b* ≠ *0*
shows ¬ *lt* (*fst b*) $adds_t$ *lt h*
proof
from *assms*(*1*) *f4-red-aux-not-zero* **have** *h* ≠ *0* **by** *metis*
hence *lt h* ∈ *keys h* **by** (*rule lt-in-keys*)
also from *assms*(*1*) **have** ... ⊆ *Keys* (*set* (*f4-red-aux bs ps*)) **by** (*rule keys-subset-Keys*)
also have ... ⊆ *Keys* (*set* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
(**is** - ⊆ *Keys* (*set ?s*)) **by** (*simp only*: *f4-red-aux-def Let-def fst-sym-preproc Keys-Macaulay-red*)
finally have *lt h* ∈ *Keys* (*set ?s*) **.**
moreover from *assms*(*2*) **have** *fst b* ∈ *set* (*map fst bs*) **by** *auto*
moreover assume *a*: *lt* (*fst b*) $adds_t$ *lt h*
ultimately have *monom-mult 1* (*lp h* − *lp* (*fst b*)) (*fst b*) ∈ *set ?s* (**is** *?m* ∈ -)
by (*rule snd-sym-preproc-complete*)
from *assms*(*3*) **have** *?m* ≠ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
with ‹*?m* ∈ *set ?s*› **have** *lt ?m* ∈ *lt-set* (*set ?s*) **by** (*rule lt-setI*)
moreover from *assms*(*3*) *a* **have** *lt ?m* = *lt h*
by (*simp add*: *lt-monom-mult*, *metis add-diff-cancel-right*′ *adds-termE pp-of-term-splus*)
ultimately have *lt h* ∈ *lt-set* (*set ?s*) **by** *simp*
moreover from *assms*(*1*) **have** *lt h* ∉ *lt-set* (*set ?s*)

by (*simp add*: *f4-red-aux-def Let-def fst-sym-preproc set-Macaulay-red*)
ultimately show *False* **by** *simp*
qed

lemma *f4-red-aux-dgrad-p-set-le*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* (*set* (*f4-red-aux bs ps*)) (*args-to-set* ([], *bs*, *ps*))
unfolding *dgrad-p-set-le-def dgrad-set-le-def*
proof
fix *s*
assume *s* ∈ *pp-of-term* ' *Keys* (*set* (*f4-red-aux bs ps*))
also have ... ⊆ *pp-of-term* ' *Keys* (*set* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
(**is** - ⊆ *pp-of-term* ' *Keys* (*set ?s*))
by (*rule image-mono*, *simp only*: *f4-red-aux-def Let-def fst-sym-preproc Keys-Macaulay-red*)
finally have *s* ∈ *pp-of-term* ' *Keys* (*set ?s*) **.**
with *snd-sym-preproc-dgrad-set-le*[*OF assms*] **obtain** *t*
where *t* ∈ *pp-of-term* ' *Keys* (*set* (*map fst bs*) ∪ *set* (*pdata-pairs-to-list ps*))
and *d s* ≤ *d t*
by (*rule dgrad-set-leE*)
from *this*(*1*) **have** *t* ∈ *pp-of-term* ' *Keys* (*fst* ' *set bs*) ∨ *t* ∈ *pp-of-term* ' *Keys* (*set* (*pdata-pairs-to-list ps*))
by (*simp add*: *Keys-Un image-Un*)
thus ∃ *t* ∈ *pp-of-term* ' *Keys* (*args-to-set* ([], *bs*, *ps*)). *d s* ≤ *d t*
proof
assume *t* ∈ *pp-of-term* ' *Keys* (*fst* ' *set bs*)
also have ... ⊆ *pp-of-term* ' *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *args-to-set-alt*)
finally have *t* ∈ *pp-of-term* ' *Keys* (*args-to-set* ([], *bs*, *ps*)) **.**
with ‹*d s* ≤ *d t*› **show** *?thesis* **..**
next
assume *t* ∈ *pp-of-term* ' *Keys* (*set* (*pdata-pairs-to-list ps*))
then obtain *p* **where** *p* ∈ *set* (*pdata-pairs-to-list ps*) **and** *t* ∈ *pp-of-term* ' *keys p*
by (*auto elim*: *in-KeysE*)
from *this*(*1*) **obtain** *f g* **where** *disj*: (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *p*: *p* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
by (*rule in-pdata-pairs-to-listE*)
from *disj* **have** *fst f* ∈ *args-to-set* ([], *bs*, *ps*) ∧ *fst g* ∈ *args-to-set* ([], *bs*, *ps*)
proof
assume (*f*, *g*) ∈ *set ps*
hence *f* ∈ *fst* ' *set ps* **and** *g* ∈ *snd* ' *set ps* **by** *force+*
hence *fst f* ∈ *fst* ' *fst* ' *set ps* **and** *fst g* ∈ *fst* ' *snd* ' *set ps* **by** *simp-all*
thus *?thesis* **by** (*simp add*: *args-to-set-def image-Un*)
next
assume (*g*, *f*) ∈ *set ps*
hence *f* ∈ *snd* ' *set ps* **and** *g* ∈ *fst* ' *set ps* **by** *force+*
hence *fst f* ∈ *fst* ' *snd* ' *set ps* **and** *fst g* ∈ *fst* ' *fst* ' *set ps* **by** *simp-all*

thus *?thesis* **by** (*simp add*: *args-to-set-def image-Un*)
qed
hence *fst f* ∈ *args-to-set* ([], *bs*, *ps*) **and** *fst g* ∈ *args-to-set* ([], *bs*, *ps*) **by** *simp-all*
hence *keys-f*: *keys* (*fst f*) ⊆ *Keys* (*args-to-set* ([], *bs*, *ps*))
and *keys-g*: *keys* (*fst g*) ⊆ *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*auto intro*!: *keys-subset-Keys*)
let *?lf* = *lp* (*fst f*)
let *?lg* = *lp* (*fst g*)
define *l* **where** *l* = *lcs ?lf ?lg*
have *pp-of-term* ' *keys p* ⊆ *pp-of-term* ' ((⊕) (*lcs ?lf ?lg* − *?lf*) ' *keys* (*fst f*)) **unfolding** *p*
using *keys-monom-mult-subset* **by** (*rule image-mono*)
with ‹*t* ∈ *pp-of-term* ' *keys p*› **have** *t* ∈ *pp-of-term* ' ((⊕) (*l* − *?lf*) ' *keys* (*fst f*)) **unfolding** *l-def* **..**
then obtain *t'* **where** *t'* ∈ *pp-of-term* ' *keys* (*fst f*) **and** *t*: *t* = (*l* − *?lf*) + *t'*
using *pp-of-term-splus* **by** *fastforce*
from *this*(*1*) **have** *fst f* ≠ *0* **by** *auto*
show *?thesis*
proof (*cases fst g* = *0*)
case *True*
hence *?lg* = *0* **by** (*simp add*: *lt-def min-term-def term-simps*)
hence *l* = *?lf* **by** (*simp add*: *l-def lcs-zero lcs-comm*)
hence *t* = *t'* **by** (*simp add*: *t*)
with ‹*d s* ≤ *d t*› **have** *d s* ≤ *d t'* **by** *simp*
moreover from ‹*t'* ∈ *pp-of-term* ' *keys* (*fst f*)› *keys-f* **have** *t'* ∈ *pp-of-term* ' *Keys* (*args-to-set* ([], *bs*, *ps*))
by *blast*
ultimately show *?thesis* **..**
next
case *False*
have *d t* = *d* (*l* − *?lf*) ∨ *d t* = *d t'*
by (*auto simp add*: *t dickson-gradingD1*[*OF assms*])
thus *?thesis*
proof
assume *d t* = *d* (*l* − *?lf*)
also from *assms* **have** ... ≤ *ord-class.max* (*d ?lf*) (*d ?lg*)
unfolding *l-def* **by** (*rule dickson-grading-lcs-minus*)
finally have *d s* ≤ *d ?lf* ∨ *d s* ≤ *d ?lg* **using** ‹*d s* ≤ *d t*› **by** *auto*
thus *?thesis*
proof
assume *d s* ≤ *d ?lf*
moreover have *lt* (*fst f*) ∈ *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*rule*, *rule lt-in-keys*, *fact*+)
ultimately show *?thesis* **by** *blast*
next
assume *d s* ≤ *d ?lg*
moreover have *lt* (*fst g*) ∈ *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*rule*, *rule lt-in-keys*, *fact*+)

ultimately show *?thesis* **by** *blast*
qed
next
assume $d\ t = d\ t'$
with ‹$d\ s \le d\ t$› **have** $d\ s \le d\ t'$ **by** *simp*
moreover from ‹$t' \in$ *pp-of-term* ‘ *keys* (*fst f*)› *keys-f* **have** $t' \in$ *pp-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
by *blast*
ultimately show *?thesis* **..**
qed
qed
qed
qed

lemma *components-f4-red-aux-subset*:
component-of-term ‘ *Keys* (*set* (*f4-red-aux bs ps*)) ⊆ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
proof
fix *k*
assume $k \in$ *component-of-term* ‘ *Keys* (*set* (*f4-red-aux bs ps*))
also have ... ⊆ *component-of-term* ‘ *Keys* (*set* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
by (*rule image-mono*, *simp only*: *f4-red-aux-def Let-def fst-sym-preproc Keys-Macaulay-red*)
also have ... ⊆ *component-of-term* ‘ *Keys* (*set* (*map fst bs*) ∪ *set* (*pdata-pairs-to-list ps*))
by (*fact components-snd-sym-preproc-subset*)
finally have $k \in$ *component-of-term* ‘ *Keys* (*fst* ‘ *set bs*) ∪ *component-of-term* ‘ *Keys* (*set* (*pdata-pairs-to-list ps*))
by (*simp add*: *image-Un Keys-Un*)
thus $k \in$ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
proof
assume $k \in$ *component-of-term* ‘ *Keys* (*fst* ‘ *set bs*)
also have ... ⊆ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*rule image-mono*, *rule Keys-mono*, *auto simp add*: *args-to-set-alt*)
finally show $k \in$ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*)) **.**
next
assume $k \in$ *component-of-term* ‘ *Keys* (*set* (*pdata-pairs-to-list ps*))
then obtain *p* **where** $p \in$ *set* (*pdata-pairs-to-list ps*) **and** $k \in$ *component-of-term* ‘ *keys p*
by (*auto elim*: *in-KeysE*)
from *this*(*1*) **obtain** *f g* **where** *disj*: (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *p*: *p* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
by (*rule in-pdata-pairs-to-listE*)
from *disj* **have** *fst f* ∈ *args-to-set* ([], *bs*, *ps*)
by (*simp add*: *args-to-set-alt*, *metis fst-conv image-eqI snd-conv*)
hence *fst f* ∈ *args-to-set* ([], *bs*, *ps*) **by** *simp*
hence *keys-f*: *keys* (*fst f*) ⊆ *Keys* (*args-to-set* ([], *bs*, *ps*))
by (*auto intro*!: *keys-subset-Keys*)

let *?lf* = *lp* (*fst f*)
let *?lg* = *lp* (*fst g*)
define *l* **where** *l* = *lcs ?lf ?lg*
have *component-of-term* ‘ *keys p* ⊆ *component-of-term* ‘ ((⊕) (*lcs ?lf ?lg* − *?lf*) ‘ *keys* (*fst f*))
unfolding *p* **using** *keys-monom-mult-subset* **by** (*rule image-mono*)
with ‹*k* ∈ *component-of-term* ‘ *keys p*› **have** *k* ∈ *component-of-term* ‘ ((⊕) (*l* − *?lf*) ‘ *keys* (*fst f*))
unfolding *l-def* **..**
hence *k* ∈ *component-of-term* ‘ *keys* (*fst f*) **using** *component-of-term-splus* **by** *fastforce*
with *keys-f* **show** *k* ∈ *component-of-term* ‘ *Keys* (*args-to-set* ([], *bs*, *ps*)) **by** *blast*
qed
qed

lemma *pmdl-f4-red-aux*: *set* (*f4-red-aux bs ps*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
proof −
have *set* (*f4-red-aux bs ps*) ⊆
set (*Macaulay-list* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
by (*auto simp add*: *f4-red-aux-def Let-def fst-sym-preproc set-Macaulay-red*)
also have ... ⊆ *pmdl* (*set* (*Macaulay-list* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*)))))
by (*fact pmdl.span-superset*)
also have ... = *pmdl* (*set* (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
by (*fact pmdl-Macaulay-list*)
also have ... ⊆ *pmdl* (*set* (*map fst bs*) ∪
set (*snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))))
by (*rule pmdl.span-mono*, *blast*)
also have ... = *pmdl* (*set* (*map fst bs*) ∪ *set* (*pdata-pairs-to-list ps*))
by (*fact snd-sym-preproc-pmdl*)
also have ... ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
proof (*rule pmdl.span-subset-spanI*, *simp only*: *Un-subset-iff*, *rule conjI*)
have *set* (*map fst bs*) ⊆ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-def*)
also have ... ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*)) **by** (*rule pmdl.span-superset*)
finally show *set* (*map fst bs*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*)) **.**
next
show *set* (*pdata-pairs-to-list ps*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
proof
fix *p*
assume *p* ∈ *set* (*pdata-pairs-to-list ps*)
then obtain *f g* **where** (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *p*: *p* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
by (*rule in-pdata-pairs-to-listE*)
from *this*(*1*) **have** *f* ∈ *fst* ‘ *set ps* ∪ *snd* ‘ *set ps* **by** *force*
hence *fst f* ∈ *args-to-set* ([], *bs*, *ps*) **by** (*auto simp add*: *args-to-set-alt*)
hence *fst f* ∈ *pmdl* (*args-to-set* ([], *bs*, *ps*)) **by** (*rule pmdl.span-base*)

thus *p* ∈ *pmdl* (*args-to-set* ([], *bs*, *ps*)) **unfolding** *p* **by** (*rule pmdl-closed-monom-mult*)
qed
qed
finally show *?thesis* **.**
qed

lemma *f4-red-aux-phull-reducible*:
assumes *set ps* ⊆ *set bs* × *set bs*
and *f* ∈ *phull* (*set* (*pdata-pairs-to-list ps*))
shows (*red* (*fst* ' *set bs* ∪ *set* (*f4-red-aux bs ps*)))** *f 0*
proof −
define *fs* **where** *fs* = *snd* (*sym-preproc* (*map fst bs*) (*pdata-pairs-to-list ps*))
have *set* (*pdata-pairs-to-list ps*) ⊆ *set fs* **unfolding** *fs-def* **by** (*fact snd-sym-preproc-superset*)
hence *phull* (*set* (*pdata-pairs-to-list ps*)) ⊆ *phull* (*set fs*) **by** (*rule phull.span-mono*)
with *assms*(*2*) **have** *f-in*: *f* ∈ *phull* (*set fs*) **..**
have *eq*: (*set fs*) ∪ *set* (*f4-red-aux bs ps*) = (*set fs*) ∪ *set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)
by (*simp add*: *f4-red-aux-def fs-def Let-def fst-sym-preproc*)

have (*lin-red* ((*set fs*) ∪ *set* (*f4-red-aux bs ps*)))** *f 0*
by (*simp only*: *eq*, *rule Macaulay-red-reducible*, *fact f-in*, *fact subset-refl*, *fact refl*)
thus *?thesis*
proof *induct*
case *base*
show *?case* **..**
next
case (*step y z*)
from *step*(*2*) **have** *red* (*fst* ' *set bs* ∪ *set* (*f4-red-aux bs ps*)) *y z* **unfolding** *lin-red-Un*
proof
assume *lin-red* (*set fs*) *y z*
then obtain *a* **where** *a* ∈ *set fs* **and** *r*: *red-single y z a 0* **by** (*rule lin-redE*)
from *this*(*1*) **obtain** *b c t* **where** *b* ∈ *fst* ' *set bs* **and** *a*: *a* = *monom-mult c t b* **unfolding** *fs-def*
proof (*rule in-snd-sym-preprocE*)
assume ∗: ⋀*b c t*. *b* ∈ *fst* ' *set bs* ⟹ *a* = *monom-mult c t b* ⟹ *thesis*
assume *a* ∈ *set* (*pdata-pairs-to-list ps*)
then obtain *f g* **where** (*f*, *g*) ∈ *set ps* ∨ (*g*, *f*) ∈ *set ps*
and *a*: *a* = *monom-mult* (*1* / *lc* (*fst f*)) ((*lcs* (*lp* (*fst f*)) (*lp* (*fst g*))) − (*lp* (*fst f*))) (*fst f*)
by (*rule in-pdata-pairs-to-listE*)
from *this*(*1*) **have** *f* ∈ *fst* ' *set ps* ∪ *snd* ' *set ps* **by** *force*
with *assms*(*1*) **have** *f* ∈ *set bs* **by** *fastforce*
hence *fst f* ∈ *fst* ' *set bs* **by** *simp*
from *this a* **show** *?thesis* **by** (*rule* ∗)
next
fix *g s*
assume ∗: ⋀*b c t*. *b* ∈ *fst* ' *set bs* ⟹ *a* = *monom-mult c t b* ⟹ *thesis*

assume *g* ∈ *set* (*map fst bs*)
hence *g* ∈ *fst* ' *set bs* **by** *simp*
moreover assume *a* = *monom-mult 1 s g*
ultimately show *?thesis* **by** (*rule* ∗)
qed
from *r* **have** *c* ≠ *0* **and** *b* ≠ *0* **by** (*simp-all add*: *a red-single-def monom-mult-eq-zero-iff*)
from *r* **have** *red-single y z b t*
by (*simp add*: *a red-single-def monom-mult-eq-zero-iff lt-monom-mult*[*OF* ‹*c* ≠ *0*› ‹*b* ≠ *0*›]
monom-mult-assoc term-simps)
with ‹*b* ∈ *fst* ' *set bs*› **have** *red* (*fst* ' *set bs*) *y z* **by** (*rule red-setI*)
thus *?thesis* **by** (*rule red-unionI1*)
next
assume *lin-red* (*set* (*f4-red-aux bs ps*)) *y z*
hence *red* (*set* (*f4-red-aux bs ps*)) *y z* **by** (*rule lin-red-imp-red*)
thus *?thesis* **by** (*rule red-unionI2*)
qed
with *step*(*3*) **show** *?case* **..**
qed
qed

corollary *f4-red-aux-spoly-reducible*:
assumes *set ps* ⊆ *set bs* × *set bs* **and** (*p*, *q*) ∈ *set ps*
shows (*red* (*fst* ' *set bs* ∪ *set* (*f4-red-aux bs ps*)))∗∗ (*spoly* (*fst p*) (*fst q*)) *0*
using *assms*(*1*)
proof (*rule f4-red-aux-phull-reducible*)
let *?lt* = *lp* (*fst p*)
let *?lq* = *lp* (*fst q*)
let *?l* = *lcs ?lt ?lq*
let *?p* = *monom-mult* (*1* / *lc* (*fst p*)) (*?l* − *?lt*) (*fst p*)
let *?q* = *monom-mult* (*1* / *lc* (*fst q*)) (*?l* − *?lq*) (*fst q*)
from *assms*(*2*) **have** *?p* ∈ *set* (*pdata-pairs-to-list ps*) **and** *?q* ∈ *set* (*pdata-pairs-to-list ps*)
by (*rule in-pdata-pairs-to-listI1*, *rule in-pdata-pairs-to-listI2*)
hence *?p* ∈ *phull* (*set* (*pdata-pairs-to-list ps*)) **and** *?q* ∈ *phull* (*set* (*pdata-pairs-to-list ps*))
by (*auto intro*: *phull.span-base*)
hence *?p* − *?q* ∈ *phull* (*set* (*pdata-pairs-to-list ps*)) **by** (*rule phull.span-diff*)
thus *spoly* (*fst p*) (*fst q*) ∈ *phull* (*set* (*pdata-pairs-to-list ps*))
by (*simp add*: *spoly-def Let-def phull.span-zero lc-def split*: *if-split*)
qed

definition *f4-red* :: (*'t*, *'b*::*field*, *'c*::*default*, *'d*) *complT*
where *f4-red gs bs ps sps data* = (*map* (*λh*. (*h*, *default*)) (*f4-red-aux* (*gs* @ *bs*) *sps*), *snd data*)

lemma *fst-set-fst-f4-red*: *fst* ' *set* (*fst* (*f4-red gs bs ps sps data*)) = *set* (*f4-red-aux* (*gs* @ *bs*) *sps*)
by (*simp add*: *f4-red-def*, *force*)

lemma *rcp-spec-f4-red*: *rcp-spec f4-red*
proof (*rule rcp-specI*)
  fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
  show *0* ∉ *fst* ' *set* (*fst* (*f4-red gs bs ps sps data*))
    by (*simp add*: *fst-set-fst-f4-red f4-red-aux-not-zero*)
next
  fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps h b* **and** *data*::*nat* × '*d*
  assume *h* ∈ *set* (*fst* (*f4-red gs bs ps sps data*)) **and** *b* ∈ *set gs* ∪ *set bs*
  from *this*(*1*) **have** *fst h* ∈ *fst* ' *set* (*fst* (*f4-red gs bs ps sps data*)) **by** *simp*
  hence *fst h* ∈ *set* (*f4-red-aux* (*gs* @ *bs*) *sps*) **by** (*simp only*: *fst-set-fst-f4-red*)
  moreover from ‹*b* ∈ *set gs* ∪ *set bs*› **have** *b* ∈ *set* (*gs* @ *bs*) **by** *simp*
  moreover assume *fst b* ≠ *0*
  ultimately show ¬ *lt* (*fst b*) $adds_t$ *lt* (*fst h*) **by** (*rule f4-red-aux-irredudible*)
next
  fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *d*::'*a* ⇒ *nat* **and** *data*::*nat* × '*d*
  assume *dickson-grading d*
  hence *dgrad-p-set-le d* (*set* (*f4-red-aux* (*gs* @ *bs*) *sps*)) (*args-to-set* ([], *gs* @ *bs*, *sps*))
    by (*fact f4-red-aux-dgrad-p-set-le*)
  also have ... = *args-to-set* (*gs*, *bs*, *sps*) **by** (*simp add*: *args-to-set-alt image-Un*)
  finally show *dgrad-p-set-le d* (*fst* ' *set* (*fst* (*f4-red gs bs ps sps data*))) (*args-to-set* (*gs*, *bs*, *sps*))
    by (*simp only*: *fst-set-fst-f4-red*)
next
  fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
  have *component-of-term* ' *Keys* (*set* (*f4-red-aux* (*gs* @ *bs*) *sps*)) ⊆
        component-of-term ' *Keys* (*args-to-set* ([], *gs* @ *bs*, *sps*))
    by (*fact components-f4-red-aux-subset*)
  also have ... = *component-of-term* ' *Keys* (*args-to-set* (*gs*, *bs*, *sps*))
    by (*simp add*: *args-to-set-alt image-Un*)
  finally show *component-of-term* ' *Keys* (*fst* ' *set* (*fst* (*f4-red gs bs ps sps data*))) ⊆
        component-of-term ' *Keys* (*args-to-set* (*gs*, *bs*, *sps*))
    by (*simp only*: *fst-set-fst-f4-red*)
next
  fix *gs bs*::('*t*, '*b*, '*c*) *pdata list* **and** *ps sps* **and** *data*::*nat* × '*d*
  have *set* (*f4-red-aux* (*gs* @ *bs*) *sps*) ⊆ *pmdl* (*args-to-set* ([], *gs* @ *bs*, *sps*))
    by (*fact pmdl-f4-red-aux*)
  also have ... = *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) **by** (*simp add*: *args-to-set-alt image-Un*)
  finally have *fst* ' *set* (*fst* (*f4-red gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*))
    by (*simp only*: *fst-set-fst-f4-red*)
  moreover {
    fix *p q* :: ('*t*, '*b*, '*c*) *pdata*
    assume *set sps* ⊆ *set bs* × (*set gs* ∪ *set bs*)
    hence *set sps* ⊆ *set* (*gs* @ *bs*) × *set* (*gs* @ *bs*) **by** *fastforce*
    moreover assume (*p*, *q*) ∈ *set sps*

ultimately have (*red* (*fst* ‘ *set* (*gs* @ *bs*) ∪ *set* (*f4-red-aux* (*gs* @ *bs*) *sps*)))** (*spoly* (*fst p*) (*fst q*)) *0*
by (*rule f4-red-aux-spoly-reducible*)
}
ultimately show
fst ‘ *set* (*fst* (*f4-red gs bs ps sps data*)) ⊆ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) ∧
(∀ (*p*, *q*)∈*set sps*.
set sps ⊆ *set bs* × (*set gs* ∪ *set bs*) ⟶
(*red* (*fst* ‘ (*set gs* ∪ *set bs*) ∪ *fst* ‘ *set* (*fst* (*f4-red gs bs ps sps data*))))** (*spoly* (*fst p*) (*fst q*)) *0*)
by (*auto simp add*: *image-Un fst-set-fst-f4-red*)
qed

lemmas *compl-struct-f4-red* = *compl-struct-rcp*[*OF rcp-spec-f4-red*]
lemmas *compl-pmdl-f4-red* = *compl-pmdl-rcp*[*OF rcp-spec-f4-red*]
lemmas *compl-conn-f4-red* = *compl-conn-rcp*[*OF rcp-spec-f4-red*]

16.4 Pair Selection

primrec *f4-sel-aux* :: *'a* ⇒ (*'t*, *'b*::*zero*, *'c*) *pdata-pair list* ⇒ (*'t*, *'b*, *'c*) *pdata-pair list* **where**
f4-sel-aux - [] = []|
f4-sel-aux t (*p* # *ps*) =
(*if* (*lcs* (*lp* (*fst* (*fst p*))) (*lp* (*fst* (*snd p*)))) = *t then*
p # (*f4-sel-aux t ps*)
else
[]
)

lemma *f4-sel-aux-subset*: *set* (*f4-sel-aux t ps*) ⊆ *set ps*
by (*induct ps*, *auto*)

primrec *f4-sel* :: (*'t*, *'b*::*zero*, *'c*, *'d*) *selT* **where**
f4-sel gs bs [] *data* = []|
f4-sel gs bs (*p* # *ps*) *data* = *p* # (*f4-sel-aux* (*lcs* (*lp* (*fst* (*fst p*))) (*lp* (*fst* (*snd p*)))) *ps*)

lemma *sel-spec-f4-sel*: *sel-spec f4-sel*
proof (*rule sel-specI*)
fix *gs bs* :: (*'t*, *'b*, *'c*) *pdata list* **and** *ps*::(*'t*, *'b*, *'c*) *pdata-pair list* **and** *data*::*nat* × *'d*
assume *ps* ≠ []
then obtain *p ps'* **where** *ps*: *ps* = *p* # *ps'* **by** (*meson list.exhaust*)
show *f4-sel gs bs ps data* ≠ [] ∧ *set* (*f4-sel gs bs ps data*) ⊆ *set ps*
proof
show *f4-sel gs bs ps data* ≠ [] **by** (*simp add*: *ps*)
next
from *f4-sel-aux-subset* **show** *set* (*f4-sel gs bs ps data*) ⊆ *set ps* **by** (*auto simp add*: *ps*)

qed
qed

16.5 The F4 Algorithm

The F4 algorithm is just *gb-schema-direct* with parameters instantiated by suitable functions.

lemma *struct-spec-f4*: *struct-spec f4-sel add-pairs-canon add-basis-canon f4-red*
using *sel-spec-f4-sel ap-spec-add-pairs-canon ab-spec-add-basis-sorted compl-struct-f4-red*
by (*rule struct-specI*)

definition *f4-aux* :: ('*t*, '*b*, '*c*) *pdata list* ⇒ *nat* × *nat* × '*d* ⇒ ('*t*, '*b*, '*c*) *pdata list* ⇒
('*t*, '*b*, '*c*) *pdata-pair list* ⇒ ('*t*, '*b*::*field*, '*c*::*default*) *pdata list*
where *f4-aux* = *gb-schema-aux f4-sel add-pairs-canon add-basis-canon f4-red*

lemmas *f4-aux-simps* [*code*] = *gb-schema-aux-simps*[*OF struct-spec-f4*, *folded f4-aux-def*]

definition *f4* :: ('*t*, '*b*, '*c*) *pdata' list* ⇒ '*d* ⇒ ('*t*, '*b*::*field*, '*c*::*default*) *pdata' list*
where *f4* = *gb-schema-direct f4-sel add-pairs-canon add-basis-canon f4-red*

lemmas *f4-simps* [*code*] = *gb-schema-direct-def*[*of f4-sel add-pairs-canon add-basis-canon f4-red*, *folded f4-def f4-aux-def*]

lemmas *f4-isGB* = *gb-schema-direct-isGB*[*OF struct-spec-f4 compl-conn-f4-red*, *folded f4-def*]

lemmas *f4-pmdl* = *gb-schema-direct-pmdl*[*OF struct-spec-f4 compl-pmdl-f4-red*, *folded f4-def*]

16.5.1 Special Case: *punit*

lemma (**in** *gd-term*) *struct-spec-f4-punit*: *punit.struct-spec punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*
using *punit.sel-spec-f4-sel ap-spec-add-pairs-punit-canon ab-spec-add-basis-sorted punit.compl-struct-f4-red*
by (*rule punit.struct-specI*)

definition *f4-aux-punit* :: ('*a*, '*b*, '*c*) *pdata list* ⇒ *nat* × *nat* × '*d* ⇒ ('*a*, '*b*, '*c*) *pdata list* ⇒
('*a*, '*b*, '*c*) *pdata-pair list* ⇒ ('*a*, '*b*::*field*, '*c*::*default*) *pdata list*
where *f4-aux-punit* = *punit.gb-schema-aux punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*

lemmas *f4-aux-punit-simps* [*code*] = *punit.gb-schema-aux-simps*[*OF struct-spec-f4-punit*, *folded f4-aux-punit-def*]

definition *f4-punit* :: ('*a*, '*b*, '*c*) *pdata' list* ⇒ '*d* ⇒ ('*a*, '*b*::*field*, '*c*::*default*) *pdata' list*

where *f4-punit* = *punit.gb-schema-direct punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*

lemmas *f4-punit-simps* [*code*] = *punit.gb-schema-direct-def*[*of punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*, *folded f4-punit-def f4-aux-punit-def*]

lemmas *f4-punit-isGB* = *punit.gb-schema-direct-isGB*[*OF struct-spec-f4-punit punit.compl-conn-f4-red*, *folded f4-punit-def*]

lemmas *f4-punit-pmdl* = *punit.gb-schema-direct-pmdl*[*OF struct-spec-f4-punit punit.compl-pmdl-f4-red*, *folded f4-punit-def*]

end

end

17 Sample Computations with the F4 Algorithm

theory *F4-Examples*
imports *F4 Algorithm-Schema-Impl Jordan-Normal-Form.Gauss-Jordan-IArray-Impl Code-Target-Rat*
begin

We only consider scalar polynomials here, but vector-polynomials could be handled, too.

17.1 Preparations

primrec *remdups-wrt-rev* :: ('*a* ⇒ '*b*) ⇒ '*a list* ⇒ '*b list* ⇒ '*a list* **where**
remdups-wrt-rev f [] *vs* = [] |
remdups-wrt-rev f (*x* # *xs*) *vs* =
(*let fx* = *f x in if List.member vs fx then remdups-wrt-rev f xs vs else x* # (*remdups-wrt-rev f xs* (*fx* # *vs*)))

lemma *remdups-wrt-rev-notin*: *v* ∈ *set vs* ⟹ *v* ∉ *f* ‘ *set* (*remdups-wrt-rev f xs vs*)
proof (*induct xs arbitrary*: *vs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons*(*2*) **have** *1*: *v* ∉ *f* ‘ *set* (*remdups-wrt-rev f xs vs*) **by** (*rule Cons*(*1*))
from *Cons*(*2*) **have** *v* ∈ *set* (*f x* # *vs*) **by** *simp*
hence *2*: *v* ∉ *f* ‘ *set* (*remdups-wrt-rev f xs* (*f x* # *vs*)) **by** (*rule Cons*(*1*))
from *Cons*(*2*) **show** *?case* **by** (*auto simp*: *Let-def 1 2 List.member-def*)
qed

lemma *distinct-remdups-wrt-rev*: *distinct* (*map f* (*remdups-wrt-rev f xs vs*))
proof (*induct xs arbitrary*: *vs*)

case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case* **by** (*simp add*: *Let-def Cons*(*1*) *remdups-wrt-rev-notin*)
qed

lemma *map-of-remdups-wrt-rev'*:
map-of (*remdups-wrt-rev fst xs vs*) *k* = *map-of* (*filter* (*λx. fst x* ∉ *set vs*) *xs*) *k*
proof (*induct xs arbitrary*: *vs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*simp add*: *Let-def List.member-def Cons*, *intro impI*)
assume *k* ≠ *fst x*
have *map-of* (*filter* (*λy. fst y* ≠ *fst x* ∧ *fst y* ∉ *set vs*) *xs*) =
map-of (*filter* (*λy. fst y* ≠ *fst x*) (*filter* (*λy. fst y* ∉ *set vs*) *xs*))
by (*simp only*: *filter-filter conj-commute*)
also have ... = *map-of* (*filter* (*λy. fst y* ∉ *set vs*) *xs*) |‘ {*y. y* ≠ *fst x*} **by** (*rule map-of-filter*)
finally show *map-of* (*filter* (*λy. fst y* ≠ *fst x* ∧ *fst y* ∉ *set vs*) *xs*) *k* =
map-of (*filter* (*λy. fst y* ∉ *set vs*) *xs*) *k*
by (*simp add*: *restrict-map-def* ‹*k* ≠ *fst x*›)
qed
qed

corollary *map-of-remdups-wrt-rev*: *map-of* (*remdups-wrt-rev fst xs* []) = *map-of xs*
by (*rule ext*, *simp add*: *map-of-remdups-wrt-rev'*)

lemma (**in** *term-powerprod*) *compute-list-to-poly* [*code*]:
list-to-poly ts cs = $distr_0$ *DRLEX* (*remdups-wrt-rev fst* (*zip ts cs*) [])
by (*rule poly-mapping-eqI*,
simp add: *lookup-list-to-poly list-to-fun-def* $distr_0$*-def oalist-of-list-ntm-def oa-ntm.lookup-oalist-of-list distinct-remdups-wrt-rev lookup-dflt-def map-of-remdups-wrt-rev*)

lemma (**in** *ordered-term*) *compute-Macaulay-list* [*code*]:
Macaulay-list ps =
(*let ts* = *Keys-to-list ps in*
filter (*λp. p* ≠ *0*) (*mat-to-polys ts* (*row-echelon* (*polys-to-mat ts ps*)))
)
by (*simp add*: *Macaulay-list-def Macaulay-mat-def Let-def*)

declare *conversep-iff* [*code*]

derive (*eq*) *ceq poly-mapping*
derive (*no*) *ccompare poly-mapping*

derive (*dlist*) *set-impl poly-mapping*
derive (*no*) *cenum poly-mapping*

derive (*eq*) *ceq rat*
derive (*no*) *ccompare rat*
derive (*dlist*) *set-impl rat*
derive (*no*) *cenum rat*

global-interpretation *punit'*: *gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit cmp-term*
rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = ($\lambda x.\ x$)
and *punit.component-of-term* = (λ-. ())
and *punit.monom-mult* = *monom-mult-punit*
and *punit.mult-scalar* = *mult-scalar-punit*
and *punit'.punit.min-term* = *min-term-punit*
and *punit'.punit.lt* = *lt-punit cmp-term*
and *punit'.punit.lc* = *lc-punit cmp-term*
and *punit'.punit.tail* = *tail-punit cmp-term*
and *punit'.punit.ord-p* = *ord-p-punit cmp-term*
and *punit'.punit.ord-strict-p* = *ord-strict-p-punit cmp-term*
and *punit'.punit.keys-to-list* = *keys-to-list-punit cmp-term*
for *cmp-term* :: (*'a*::*nat*, *'b*::{*nat*,*add-wellorder*}) *pp nat-term-order*

defines *max-punit* = *punit'.ordered-powerprod-lin.max*
and *max-list-punit* = *punit'.ordered-powerprod-lin.max-list*
and *find-adds-punit* = *punit'.punit.find-adds*
and *trd-aux-punit* = *punit'.punit.trd-aux*
and *trd-punit* = *punit'.punit.trd*
and *spoly-punit* = *punit'.punit.spoly*
and *count-const-lt-components-punit* = *punit'.punit.count-const-lt-components*
and *count-rem-components-punit* = *punit'.punit.count-rem-components*
and *const-lt-component-punit* = *punit'.punit.const-lt-component*
and *full-gb-punit* = *punit'.punit.full-gb*
and *add-pairs-single-sorted-punit* = *punit'.punit.add-pairs-single-sorted*
and *add-pairs-punit* = *punit'.punit.add-pairs*
and *canon-pair-order-aux-punit* = *punit'.punit.canon-pair-order-aux*
and *canon-basis-order-punit* = *punit'.punit.canon-basis-order*
and *new-pairs-sorted-punit* = *punit'.punit.new-pairs-sorted*
and *product-crit-punit* = *punit'.punit.product-crit*
and *chain-ncrit-punit* = *punit'.punit.chain-ncrit*
and *chain-ocrit-punit* = *punit'.punit.chain-ocrit*
and *apply-icrit-punit* = *punit'.punit.apply-icrit*
and *apply-ncrit-punit* = *punit'.punit.apply-ncrit*
and *apply-ocrit-punit* = *punit'.punit.apply-ocrit*
and *Keys-to-list-punit* = *punit'.punit.Keys-to-list*
and *sym-preproc-addnew-punit* = *punit'.punit.sym-preproc-addnew*
and *sym-preproc-aux-punit* = *punit'.punit.sym-preproc-aux*
and *sym-preproc-punit* = *punit'.punit.sym-preproc*

and *Macaulay-mat-punit* = *punit′.punit.Macaulay-mat*
and *Macaulay-list-punit* = *punit′.punit.Macaulay-list*
and *pdata-pairs-to-list-punit* = *punit′.punit.pdata-pairs-to-list*
and *Macaulay-red-punit* = *punit′.punit.Macaulay-red*
and *f4-sel-aux-punit* = *punit′.punit.f4-sel-aux*
and *f4-sel-punit* = *punit′.punit.f4-sel*
and *f4-red-aux-punit* = *punit′.punit.f4-red-aux*
and *f4-red-punit* = *punit′.punit.f4-red*
and *f4-aux-punit* = *punit′.punit.f4-aux-punit*
and *f4-punit* = *punit′.punit.f4-punit*
subgoal by (*fact gd-powerprod-ord-pp-punit*)
subgoal by (*fact punit-adds-term*)
subgoal by (*simp add*: *id-def*)
subgoal by (*fact punit-component-of-term*)
subgoal by (*simp only*: *monom-mult-punit-def*)
subgoal by (*simp only*: *mult-scalar-punit-def*)
subgoal using *min-term-punit-def* **by** *fastforce*
subgoal by (*simp only*: *lt-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *lc-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *tail-punit-def ord-pp-punit-alt*)
subgoal by (*simp only*: *ord-p-punit-def ord-pp-strict-punit-alt*)
subgoal by (*simp only*: *ord-strict-p-punit-def ord-pp-strict-punit-alt*)
subgoal by (*simp only*: *keys-to-list-punit-def ord-pp-punit-alt*)
done

17.2 Computations

experiment begin interpretation *trivariate*$_0$*-rat* **.**

lemma
lt-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *sparse*$_0$ [(*0*, *2*), (*2*, *3*)]
by *eval*

lemma
lc-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *1*
by *eval*

lemma
tail-punit DRLEX (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*) = *3* ∗ X^2 ∗ *Y*
by *eval*

lemma
ord-strict-p-punit DRLEX (X^2 ∗ *Z* ^ *4* − *2* ∗ *Y* ^ *3* ∗ Z^2) (X^2 ∗ *Z* ^ *7* + *2* ∗ *Y* ^ *3* ∗ Z^2)
by *eval*

lemma
f4-punit DRLEX
[

```
    (X² * Z ^ 4 − 2 * Y ^ 3 * Z², ()),
    (Y² * Z + 2 * Z ^ 3, ())
  ] () =
  [
    (X² * Y² * Z² + 4 * Y ^ 3 * Z², ()),
    (X² * Z ^ 4 − 2 * Y ^ 3 * Z², ()),
    (Y² * Z + 2 * Z ^ 3, ()),
    (X² * Y ^ 4 * Z + 4 * Y ^ 5 * Z, ())
  ]
  by eval
```

```
lemma
  f4-punit DRLEX
  [
    (X² + Y² + Z² − 1, ()),
    (X * Y − Z − 1, ()),
    (Y² + X, ()),
    (Z² + X, ())
  ] () =
  [
    (1, ())
  ]
  by eval
```

end

value [*code*] *length* (*f4-punit DRLEX* (*map* (λ*p*. (*p*, ())) ((*cyclic DRLEX 4*)::(- ⇒$_0$ *rat*) *list*)) ())

value [*code*] *length* (*f4-punit DRLEX* (*map* (λ*p*. (*p*, ())) ((*katsura DRLEX 2*)::(- ⇒$_0$ *rat*) *list*)) ())

end

18 Syzygies of Multivariate Polynomials

theory *Syzygy*
  imports *Groebner-Bases More-MPoly-Type-Class*
begin

In this theory we first introduce the general concept of *syzygies* in modules, and then provide a method for computing Gröbner bases of syzygy modules of lists of multivariate vector-polynomials. Since syzygies in this context are themselves represented by vector-polynomials, this method can be applied repeatedly to compute bases of syzygy modules of syzygies, and so on.

instance *nat* :: *comm-powerprod* ..

18.1 Syzygy Modules Generated by Sets

context *module*
begin

definition *rep* :: $('b \Rightarrow_0 'a) \Rightarrow 'b$
where *rep r* = ($\sum$ *v*∈*keys r*. *lookup r v* ∗*s v*)

definition *represents* :: $'b$ *set* ⇒ $('b \Rightarrow_0 'a) \Rightarrow 'b \Rightarrow$ *bool*
where *represents B r x* ⟷ (*keys r* ⊆ *B* ∧ *local.rep r* = *x*)

definition *syzygy-module* :: $'b$ *set* ⇒ $('b \Rightarrow_0 'a)$ *set*
where *syzygy-module B* = {*s*. *local.represents B s 0*}

end

hide-const (**open**) *real-vector.rep real-vector.represents real-vector.syzygy-module*

context *module*
begin

lemma *rep-monomial* [*simp*]: *rep* (*monomial c x*) = *c* ∗*s x*
proof −
have *sub*: *keys* (*monomial c x*) ⊆ {*x*} **by** *simp*
have *rep* (*monomial c x*) = ($\sum$ *v*∈{*x*}. *lookup* (*monomial c x*) *v* ∗*s v*) **unfolding** *rep-def*
by (*rule sum.mono-neutral-left*, *simp*, *fact sub*, *simp*)
also have ... = *c* ∗*s x* **by** *simp*
finally show *?thesis* **.**
qed

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

lemma *rep-uminus* [*simp*]: *rep* (− *r*) = − *rep r*
by (*simp add*: *keys-uminus sum-negf rep-def*)

lemma *rep-plus*: *rep* (*r* + *s*) = *rep r* + *rep s*
proof −
from *finite-keys finite-keys* **have** *fin*: *finite* (*keys r* ∪ *keys s*) **by** (*rule finite-UnI*)
from *fin* **have** *eq1*: ($\sum$ *v*∈*keys r* ∪ *keys s*. *lookup r v* ∗*s v*) = ($\sum$ *v*∈*keys r*. *lookup r v* ∗*s v*)
proof (*rule sum.mono-neutral-right*)
show ∀ *v*∈*keys r* ∪ *keys s* − *keys r*. *lookup r v* ∗*s v* = *0* **by** (*simp add*: *in-keys-iff*)
qed *simp*
from *fin* **have** *eq2*: ($\sum$ *v*∈*keys r* ∪ *keys s*. *lookup s v* ∗*s v*) = ($\sum$ *v*∈*keys s*. *lookup s v* ∗*s v*)
proof (*rule sum.mono-neutral-right*)
show ∀ *v*∈*keys r* ∪ *keys s* − *keys s*. *lookup s v* ∗*s v* = *0* **by** (*simp add*: *in-keys-iff*)

qed *simp*
have *rep* (*r* + *s*) = (∑ *v*∈*keys* (*r* + *s*). *lookup* (*r* + *s*) *v* ∗*s* *v*) **by** (*simp only*: *rep-def*)
also have ... = (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* (*r* + *s*) *v* ∗*s* *v*)
proof (*rule sum.mono-neutral-left*)
show ∀ *i*∈*keys* *r* ∪ *keys* *s* − *keys* (*r* + *s*). *lookup* (*r* + *s*) *i* ∗*s* *i* = *0* **by** (*simp add*: *in-keys-iff*)
qed (*auto simp*: *Poly-Mapping.keys-add*)
also have ... = (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *r* *v* ∗*s* *v*) + (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *s* *v* ∗*s* *v*)
by (*simp add*: *lookup-add scale-left-distrib sum.distrib*)
also have ... = *rep* *r* + *rep* *s* **by** (*simp only*: *eq1 eq2 rep-def*)
finally show *?thesis* **.**
qed

lemma *rep-minus*: *rep* (*r* − *s*) = *rep* *r* − *rep* *s*
proof −
from *finite-keys finite-keys* **have** *fin*: *finite* (*keys* *r* ∪ *keys* *s*) **by** (*rule finite-UnI*)
from *fin* **have** *eq1*: (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *r* *v* ∗*s* *v*) = (∑ *v*∈*keys* *r*. *lookup* *r* *v* ∗*s* *v*)
proof (*rule sum.mono-neutral-right*)
show ∀ *v*∈*keys* *r* ∪ *keys* *s* − *keys* *r*. *lookup* *r* *v* ∗*s* *v* = *0* **by** (*simp add*: *in-keys-iff*)
qed *simp*
from *fin* **have** *eq2*: (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *s* *v* ∗*s* *v*) = (∑ *v*∈*keys* *s*. *lookup* *s* *v* ∗*s* *v*)
proof (*rule sum.mono-neutral-right*)
show ∀ *v*∈*keys* *r* ∪ *keys* *s* − *keys* *s*. *lookup* *s* *v* ∗*s* *v* = *0* **by** (*simp add*: *in-keys-iff*)
qed *simp*
have *rep* (*r* − *s*) = (∑ *v*∈*keys* (*r* − *s*). *lookup* (*r* − *s*) *v* ∗*s* *v*) **by** (*simp only*: *rep-def*)
also from *fin keys-minus* **have** ... = (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* (*r* − *s*) *v* ∗*s* *v*)
proof (*rule sum.mono-neutral-left*)
show ∀ *i*∈*keys* *r* ∪ *keys* *s* − *keys* (*r* − *s*). *lookup* (*r* − *s*) *i* ∗*s* *i* = *0* **by** (*simp add*: *in-keys-iff*)
qed
also have ... = (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *r* *v* ∗*s* *v*) − (∑ *v*∈*keys* *r* ∪ *keys* *s*. *lookup* *s* *v* ∗*s* *v*)
by (*simp add*: *lookup-minus scale-left-diff-distrib sum-subtractf*)
also have ... = *rep* *r* − *rep* *s* **by** (*simp only*: *eq1 eq2 rep-def*)
finally show *?thesis* **.**
qed

lemma *rep-smult*: *rep* (*monomial c 0* ∗ *r*) = *c* ∗*s* *rep* *r*
proof −
have *l*: *lookup* (*monomial c 0* ∗ *r*) *v* = *c* ∗ (*lookup* *r* *v*) **for** *v*
unfolding *mult-map-scale-conv-mult*[*symmetric*] **by** (*rule map-lookup*, *simp*)
have *sub*: *keys* (*monomial c 0* ∗ *r*) ⊆ *keys* *r*

by (*metis l lookup-not-eq-zero-eq-in-keys mult-zero-right subsetI*)

have *rep* (*monomial c 0* ∗ *r*) = (∑ *v*∈*keys* (*monomial c 0* ∗ *r*). *lookup* (*monomial c 0* ∗ *r*) *v* ∗*s v*)
by (*simp only*: *rep-def*)
also from *finite-keys sub* **have** ... = (∑ *v*∈*keys r*. *lookup* (*monomial c 0* ∗ *r*) *v* ∗*s v*)
proof (*rule sum.mono-neutral-left*)
show ∀ *v*∈*keys r* − *keys* (*monomial c 0* ∗ *r*). *lookup* (*monomial c 0* ∗ *r*) *v* ∗*s v* = *0* **by** (*simp add*: *in-keys-iff*)
qed
also have ... = *c* ∗*s* (∑ *v*∈*keys r*. *lookup r v* ∗*s v*) **by** (*simp add*: *l scale-sum-right*)
also have ... = *c* ∗*s rep r* **by** (*simp add*: *rep-def*)
finally show *?thesis* **.**
qed

lemma *rep-in-span*: *rep r* ∈ *span* (*keys r*)
unfolding *rep-def* **by** (*fact sum-in-spanI*)

lemma *spanE-rep*:
assumes *x* ∈ *span B*
obtains *r* **where** *keys r* ⊆ *B* **and** *x* = *rep r*
proof −
from *assms* **obtain** *A q* **where** *finite A* **and** *A* ⊆ *B* **and** *x*: *x* = (∑ *a*∈*A*. *q a* ∗*s a*) **by** (*rule spanE*)
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
have *x* = (∑ *a*∈*A*. *lookup r a* ∗*s a*) **unfolding** *x* **by** (*rule sum.cong*, *simp-all add*: *1*)
also from ‹*finite A*› *2* **have** ... = (∑ *a*∈*keys r*. *lookup r a* ∗*s a*)
proof (*rule sum.mono-neutral-right*)
show ∀ *a*∈*A* − *keys r*. *lookup r a* ∗*s a* = *0* **by** (*simp add*: *in-keys-iff*)
qed
finally show *x* = *rep r* **by** (*simp only*: *rep-def*)
next
from *2* ‹*A* ⊆ *B*› **show** *keys r* ⊆ *B* **by** (*rule subset-trans*)
qed
qed

lemma *representsI*:
assumes *keys r* ⊆ *B* **and** *rep r* = *x*
shows *represents B r x*
unfolding *represents-def* **using** *assms* **by** *blast*

lemma *representsD1*:

assumes *represents B r x*
shows *keys r* ⊆ *B*
using *assms* **unfolding** *represents-def* **by** *blast*

lemma *representsD2*:
assumes *represents B r x*
shows *x* = *rep r*
using *assms* **unfolding** *represents-def* **by** *blast*

lemma *represents-mono*:
assumes *represents B r x* **and** *B* ⊆ *A*
shows *represents A r x*
proof (*rule representsI*)
from *assms*(*1*) **have** *keys r* ⊆ *B* **by** (*rule representsD1*)
thus *keys r* ⊆ *A* **using** *assms*(*2*) **by** (*rule subset-trans*)
next
from *assms*(*1*) **have** *x* = *rep r* **by** (*rule representsD2*)
thus *rep r* = *x* **by** (*rule HOL.sym*)
qed

lemma *represents-self*: *represents* {*x*} (*monomial 1 x*) *x*
proof −
have *sub*: *keys* (*monomial* (*1*::′*a*) *x*) ⊆ {*x*} **by** *simp*
moreover have *rep* (*monomial* (*1*::′*a*) *x*) = *x* **by** *simp*
ultimately show *?thesis* **by** (*rule representsI*)
qed

lemma *represents-zero*: *represents B 0 0*
by (*rule representsI*, *simp-all*)

lemma *represents-plus*:
assumes *represents A r x* **and** *represents B s y*
shows *represents* (*A* ∪ *B*) (*r* + *s*) (*x* + *y*)
proof −
from *assms*(*1*) **have** *r*: *keys r* ⊆ *A* **and** *x*: *x* = *rep r* **by** (*rule representsD1*, *rule representsD2*)
from *assms*(*2*) **have** *s*: *keys s* ⊆ *B* **and** *y*: *y* = *rep s* **by** (*rule representsD1*, *rule representsD2*)
show *?thesis*
proof (*rule representsI*)
from *r s* **have** *keys r* ∪ *keys s* ⊆ *A* ∪ *B* **by** *blast*
thus *keys* (*r* + *s*) ⊆ *A* ∪ *B*
by (*meson Poly-Mapping.keys-add subset-trans*)
qed (*simp add*: *rep-plus x y*)
qed

lemma *represents-uminus*:
assumes *represents B r x*
shows *represents B* (− *r*) (− *x*)

proof −
from *assms* **have** *r*: *keys r* ⊆ *B* **and** *x*: *x* = *rep r* **by** (*rule representsD1*, *rule representsD2*)
show *?thesis*
proof (*rule representsI*)
from *r* **show** *keys* (− *r*) ⊆ *B* **by** (*simp only*: *keys-uminus*)
qed (*simp add*: *x*)
qed

lemma *represents-minus*:
assumes *represents A r x* **and** *represents B s y*
shows *represents* (*A* ∪ *B*) (*r* − *s*) (*x* − *y*)
proof −
from *assms*(*1*) **have** *r*: *keys r* ⊆ *A* **and** *x*: *x* = *rep r* **by** (*rule representsD1*, *rule representsD2*)
from *assms*(*2*) **have** *s*: *keys s* ⊆ *B* **and** *y*: *y* = *rep s* **by** (*rule representsD1*, *rule representsD2*)
show *?thesis*
proof (*rule representsI*)
from *r s* **have** *keys r* ∪ *keys s* ⊆ *A* ∪ *B* **by** *blast*
with *keys-minus* **show** *keys* (*r* − *s*) ⊆ *A* ∪ *B* **by** (*rule subset-trans*)
qed (*simp only*: *rep-minus x y*)
qed

lemma *represents-scale*:
assumes *represents B r x*
shows *represents B* (*monomial c 0* ∗ *r*) (*c* ∗*s x*)
proof −
from *assms* **have** *r*: *keys r* ⊆ *B* **and** *x*: *x* = *rep r* **by** (*rule representsD1*, *rule representsD2*)
show *?thesis*
proof (*rule representsI*)
have *l*: *lookup* (*monomial c 0* ∗ *r*) *v* = *c* ∗ (*lookup r v*) **for** *v*
unfolding *mult-map-scale-conv-mult*[*symmetric*] **by** (*rule map-lookup*, *simp*)
have *sub*: *keys* (*monomial c 0* ∗ *r*) ⊆ *keys r*
by (*metis l lookup-not-eq-zero-eq-in-keys mult-zero-right subsetI*)
thus *keys* (*monomial c 0* ∗ *r*) ⊆ *B* **using** *r* **by** (*rule subset-trans*)
qed (*simp only*: *rep-smult x*)
qed

lemma *represents-in-span*:
assumes *represents B r x*
shows *x* ∈ *span B*
proof −
from *assms* **have** *r*: *keys r* ⊆ *B* **and** *x*: *x* = *rep r* **by** (*rule representsD1*, *rule representsD2*)
have *x* ∈ *span* (*keys r*) **unfolding** *x* **by** (*fact rep-in-span*)
also from *r* **have** ... ⊆ *span B* **by** (*rule span-mono*)
finally show *?thesis* **.**

qed

lemma *syzygy-module-iff*: *s* ∈ *syzygy-module B* ⟷ *represents B s 0*
by (*simp add*: *syzygy-module-def*)

lemma *syzygy-moduleI*:
assumes *represents B s 0*
shows *s* ∈ *syzygy-module B*
unfolding *syzygy-module-iff* **using** *assms* **.**

lemma *syzygy-moduleD*:
assumes *s* ∈ *syzygy-module B*
shows *represents B s 0*
using *assms* **unfolding** *syzygy-module-iff* **.**

lemma *zero-in-syzygy-module*: *0* ∈ *syzygy-module B*
using *represents-zero* **by** (*rule syzygy-moduleI*)

lemma *syzygy-module-closed-plus*:
assumes *s1* ∈ *syzygy-module B* **and** *s2* ∈ *syzygy-module B*
shows *s1* + *s2* ∈ *syzygy-module B*
proof −
from *assms*(*1*) **have** *represents B s1 0* **by** (*rule syzygy-moduleD*)
moreover from *assms*(*2*) **have** *represents B s2 0* **by** (*rule syzygy-moduleD*)
ultimately have *represents* (*B* ∪ *B*) (*s1* + *s2*) (*0* + *0*) **by** (*rule represents-plus*)
hence *represents B* (*s1* + *s2*) *0* **by** *simp*
thus *?thesis* **by** (*rule syzygy-moduleI*)
qed

lemma *syzygy-module-closed-minus*:
assumes *s1* ∈ *syzygy-module B* **and** *s2* ∈ *syzygy-module B*
shows *s1* − *s2* ∈ *syzygy-module B*
proof −
from *assms*(*1*) **have** *represents B s1 0* **by** (*rule syzygy-moduleD*)
moreover from *assms*(*2*) **have** *represents B s2 0* **by** (*rule syzygy-moduleD*)
ultimately have *represents* (*B* ∪ *B*) (*s1* − *s2*) (*0* − *0*) **by** (*rule represents-minus*)
hence *represents B* (*s1* − *s2*) *0* **by** *simp*
thus *?thesis* **by** (*rule syzygy-moduleI*)
qed

lemma *syzygy-module-closed-times-monomial*:
assumes *s* ∈ *syzygy-module B*
shows *monomial c 0* ∗ *s* ∈ *syzygy-module B*
proof −
from *assms*(*1*) **have** *represents B s 0* **by** (*rule syzygy-moduleD*)
hence *represents B* (*monomial c 0* ∗ *s*) (*c* ∗*s* *0*) **by** (*rule represents-scale*)
hence *represents B* (*monomial c 0* ∗ *s*) *0* **by** *simp*
thus *?thesis* **by** (*rule syzygy-moduleI*)
qed

end

context *term-powerprod*
begin

lemma *keys-rep-subset*:
assumes $u \in keys\ (pmdl.rep\ r)$
obtains $t\ v$ **where** $t \in Keys\ (Poly\text{-}Mapping.range\ r)$ **and** $v \in Keys\ (keys\ r)$ **and** $u = t \oplus v$
proof −
note *assms*
also have $keys\ (pmdl.rep\ r) \subseteq (\bigcup v{\in}keys\ r.\ keys\ (lookup\ r\ v \odot v))$
by (*simp add*: *pmdl.rep-def keys-sum-subset*)
finally obtain *v0* **where** $v0 \in keys\ r$ **and** $u \in keys\ (lookup\ r\ v0 \odot v0)$ **..**
from *this*(*2*) **obtain** $t\ v$ **where** $t \in keys\ (lookup\ r\ v0)$ **and** $v \in keys\ v0$ **and** $u = t \oplus v$
by (*rule in-keys-mult-scalarE*)
show *?thesis*
proof
from ‹$v0 \in keys\ r$› **have** $lookup\ r\ v0 \in Poly\text{-}Mapping.range\ r$ **by** (*rule in-keys-lookup-in-range*)
with ‹$t \in keys\ (lookup\ r\ v0)$› **show** $t \in Keys\ (Poly\text{-}Mapping.range\ r)$ **by** (*rule in-KeysI*)
next
from ‹$v \in keys\ v0$› ‹$v0 \in keys\ r$› **show** $v \in Keys\ (keys\ r)$ **by** (*rule in-KeysI*)
qed *fact*
qed

lemma *rep-mult-scalar*: $pmdl.rep\ (punit.monom\text{-}mult\ c\ 0\ r) = c \odot pmdl.rep\ r$
unfolding *punit.mult-scalar-monomial*[*symmetric*] *punit-mult-scalar* **by** (*fact pmdl.rep-smult*)

lemma *represents-mult-scalar*:
assumes *pmdl.represents B r x*
shows *pmdl.represents B* (*punit.monom-mult c 0 r*) ($c \odot x$)
unfolding *punit.mult-scalar-monomial*[*symmetric*] *punit-mult-scalar* **using** *assms*
by (*rule pmdl.represents-scale*)

lemma *syzygy-module-closed-map-scale*: $s \in pmdl.syzygy\text{-}module\ B \Longrightarrow c \cdot s \in pmdl.syzygy\text{-}module\ B$
unfolding *map-scale-eq-times* **by** (*rule pmdl.syzygy-module-closed-times-monomial*)

lemma *phull-syzygy-module*: *phull* (*pmdl.syzygy-module B*) = *pmdl.syzygy-module B*
unfolding *phull.span-eq-iff*
apply (*rule phull.subspaceI*)
subgoal by (*fact pmdl.zero-in-syzygy-module*)
subgoal by (*fact pmdl.syzygy-module-closed-plus*)
subgoal by (*fact syzygy-module-closed-map-scale*)

done

end

18.2 Polynomial Mappings on List-Indices

definition *pm-of-idx-pm* :: (*'a list*) ⇒ (*nat* ⇒$_0$ *'b*) ⇒ *'a* ⇒$_0$ *'b*::*zero*
where *pm-of-idx-pm xs f* = *Abs-poly-mapping* (λ*x*. *lookup f* (*Min* {*i*. *i* < *length xs* ∧ *xs* ! *i* = *x*}) *when x* ∈ *set xs*)

definition *idx-pm-of-pm* :: (*'a list*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *nat* ⇒$_0$ *'b*::*zero*
where *idx-pm-of-pm xs f* = *Abs-poly-mapping* (λ*i*. *lookup f* (*xs* ! *i*) *when i* < *length xs*)

lemma *lookup-pm-of-idx-pm*:
lookup (*pm-of-idx-pm xs f*) = (λ*x*. *lookup f* (*Min* {*i*. *i* < *length xs* ∧ *xs* ! *i* = *x*}) *when x* ∈ *set xs*)
unfolding *pm-of-idx-pm-def* **by** (*rule Abs-poly-mapping-inverse*, *simp*)

lemma *lookup-pm-of-idx-pm-distinct*:
assumes *distinct xs* **and** *i* < *length xs*
shows *lookup* (*pm-of-idx-pm xs f*) (*xs* ! *i*) = *lookup f i*
proof −
from *assms* **have** {*j*. *j* < *length xs* ∧ *xs* ! *j* = *xs* ! *i*} = {*i*}
using *distinct-Ex1 nth-mem* **by** *fastforce*
moreover from *assms*(*2*) **have** *xs* ! *i* ∈ *set xs* **by** (*rule nth-mem*)
ultimately show *?thesis* **by** (*simp add*: *lookup-pm-of-idx-pm*)
qed

lemma *keys-pm-of-idx-pm-subset*: *keys* (*pm-of-idx-pm xs f*) ⊆ *set xs*
proof
fix *t*
assume *t* ∈ *keys* (*pm-of-idx-pm xs f*)
hence *lookup* (*pm-of-idx-pm xs f*) *t* ≠ *0* **by** (*simp add*: *in-keys-iff*)
thus *t* ∈ *set xs* **by** (*simp add*: *lookup-pm-of-idx-pm*)
qed

lemma *range-pm-of-idx-pm-subset*: *Poly-Mapping.range* (*pm-of-idx-pm xs f*) ⊆ *lookup f* ‘ {*0*..<*length xs*} − {*0*}
proof
fix *c*
assume *c* ∈ *Poly-Mapping.range* (*pm-of-idx-pm xs f*)
then obtain *t* **where** *t*: *t* ∈ *keys* (*pm-of-idx-pm xs f*) **and** *c*: *c* = *lookup* (*pm-of-idx-pm xs f*) *t*
by (*metis DiffE imageE insertCI not-in-keys-iff-lookup-eq-zero range.rep-eq*)
from *t keys-pm-of-idx-pm-subset* **have** *t* ∈ *set xs* **..**
hence *c1*: *c* = *lookup f* (*Min* {*i*. *i* < *length xs* ∧ *xs* ! *i* = *t*}) **by** (*simp add*: *lookup-pm-of-idx-pm c*)
show *c* ∈ *lookup f* ‘ {*0*..<*length xs*} − {*0*}

proof (*intro DiffI image-eqI*)
from ‹$t \in$ *set xs*› **obtain** *i* **where** $i <$ *length xs* **and** $t = xs\ !\ i$ **by** (*metis in-set-conv-nth*)
have *finite* $\{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = t\}$ **by** *simp*
moreover from ‹$i <$ *length xs*› ‹$t = xs\ !\ i$› **have** $\{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = t\} \neq \{\}$ **by** *auto*
ultimately have *Min* $\{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = t\} \in \{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = t\}$
by (*rule Min-in*)
thus *Min* $\{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = t\} \in \{0..<$*length xs*$\}$ **by** *simp*
next
from *t* **show** $c \notin \{0\}$ **by** (*simp add*: *c in-keys-iff*)
qed (*fact c1*)
qed

corollary *range-pm-of-idx-pm-subset'*: *Poly-Mapping.range* (*pm-of-idx-pm xs f*) $\subseteq$ *Poly-Mapping.range f*
using *range-pm-of-idx-pm-subset*
proof (*rule subset-trans*)
show *lookup f* ‘ $\{0..<$*length xs*$\} - \{0\} \subseteq$ *Poly-Mapping.range f* **by** (*transfer*, *auto*)
qed

lemma *pm-of-idx-pm-zero* [*simp*]: *pm-of-idx-pm xs* $0 = 0$
by (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm*)

lemma *pm-of-idx-pm-plus*: *pm-of-idx-pm xs* $(f + g) =$ *pm-of-idx-pm xs f* $+$ *pm-of-idx-pm xs g*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm lookup-add when-def*)

lemma *pm-of-idx-pm-uminus*: *pm-of-idx-pm xs* $(- f) = -$ *pm-of-idx-pm xs f*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm when-def*)

lemma *pm-of-idx-pm-minus*: *pm-of-idx-pm xs* $(f - g) =$ *pm-of-idx-pm xs f* $-$ *pm-of-idx-pm xs g*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm lookup-minus when-def*)

lemma *pm-of-idx-pm-monom-mult*: *pm-of-idx-pm xs* (*punit.monom-mult c 0 f*) $=$ *punit.monom-mult c 0* (*pm-of-idx-pm xs f*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm punit.lookup-monom-mult-zero when-def*)

lemma *pm-of-idx-pm-monomial*:
assumes *distinct xs*
shows *pm-of-idx-pm xs* (*monomial c i*) $=$ (*monomial c* $(xs\ !\ i)$ *when* $i <$ *length xs*)
proof $-$
from *assms* **have** $*$: $\{i.\ i <$ *length xs* $\wedge\ xs\ !\ i = xs\ !\ j\} = \{j\}$ **if** $j <$ *length xs* **for** *j*

```
    using distinct-Ex1 nth-mem that by fastforce
  show ?thesis
  proof (cases i < length xs)
    case True
    have pm-of-idx-pm xs (monomial c i) = monomial c (xs ! i)
    proof (rule poly-mapping-eqI)
      fix k
      show lookup (pm-of-idx-pm xs (monomial c i)) k = lookup (monomial c (xs !
i)) k
      proof (cases xs ! i = k)
        case True
        with ‹i < length xs› have k ∈ set xs by auto
         thus ?thesis by (simp add: lookup-pm-of-idx-pm lookup-single *[OF ‹i <
length xs›] True[symmetric])
      next
        case False
        have lookup (pm-of-idx-pm xs (monomial c i)) k = 0
        proof (cases k ∈ set xs)
          case True
        then obtain j where j < length xs and k = xs ! j by (metis in-set-conv-nth)
          with False have i ≠ Min {i. i < length xs ∧ xs ! i = k}
            by (auto simp: ‹k = xs ! j› *[OF ‹j < length xs›])
          thus ?thesis by (simp add: lookup-pm-of-idx-pm True lookup-single)
        next
          case False
          thus ?thesis by (simp add: lookup-pm-of-idx-pm)
        qed
        with False show ?thesis by (simp add: lookup-single)
      qed
    qed
    with True show ?thesis by simp
  next
    case False
    have pm-of-idx-pm xs (monomial c i) = 0
    proof (rule poly-mapping-eqI, simp add: lookup-pm-of-idx-pm when-def, rule)
      fix k
      assume k ∈ set xs
     then obtain j where j < length xs and k = xs ! j by (metis in-set-conv-nth)
      with False have i ≠ Min {i. i < length xs ∧ xs ! i = k}
        by (auto simp: ‹k = xs ! j› *[OF ‹j < length xs›])
      thus lookup (monomial c i) (Min {i. i < length xs ∧ xs ! i = k}) = 0
        by (simp add: lookup-single)
    qed
    with False show ?thesis by simp
  qed
qed

lemma pm-of-idx-pm-take:
  assumes keys f ⊆ {0..<j}
```

shows *pm-of-idx-pm* (*take j xs*) *f* = *pm-of-idx-pm xs f*
proof (*rule poly-mapping-eqI*)
fix *i*
let *?xs* = *take j xs*
let *?A* = {*k*. *k* < *length xs* ∧ *xs* ! *k* = *i*}
let *?B* = {*k*. *k* < *length xs* ∧ *k* < *j* ∧ *xs* ! *k* = *i*}
have *A-fin*: *finite ?A* **and** *B-fin*: *finite ?B* **by** *fastforce+*
have *A-ne*: *i* ∈ *set xs* ⟹ *?A* ≠ {} **by** (*simp add*: *in-set-conv-nth*)
have *B-ne*: *i* ∈ *set ?xs* ⟹ *?B* ≠ {} **by** (*auto simp add*: *in-set-conv-nth*)
define *m1* **where** *m1* = *Min ?A*
define *m2* **where** *m2* = *Min ?B*
have *m1*: *m1* ∈ *?A* **if** *i* ∈ *set xs*
unfolding *m1-def* **by** (*rule Min-in*, *fact A-fin*, *rule A-ne*, *fact that*)
have *m2*: *m2* ∈ *?B* **if** *i* ∈ *set ?xs*
unfolding *m2-def* **by** (*rule Min-in*, *fact B-fin*, *rule B-ne*, *fact that*)
show *lookup* (*pm-of-idx-pm* (*take j xs*) *f*) *i* = *lookup* (*pm-of-idx-pm xs f*) *i*
proof (*cases i* ∈ *set ?xs*)
case *True*
hence *i* ∈ *set xs* **using** *set-take-subset* **..**
hence *m1* ∈ *?A* **by** (*rule m1*)
hence *m1* < *length xs* **and** *xs* ! *m1* = *i* **by** *simp-all*
from *True* **have** *m2* ∈ *?B* **by** (*rule m2*)
hence *m2* < *length xs* **and** *m2* < *j* **and** *xs* ! *m2* = *i* **by** *simp-all*
hence *m2* ∈ *?A* **by** *simp*
with *A-fin* **have** *m1* ≤ *m2* **unfolding** *m1-def* **by** (*rule Min-le*)
with ‹*m2* < *j*› **have** *m1* < *j* **by** *simp*
with ‹*m1* < *length xs*› ‹*xs* ! *m1* = *i*› **have** *m1* ∈ *?B* **by** *simp*
with *B-fin* **have** *m2* ≤ *m1* **unfolding** *m2-def* **by** (*rule Min-le*)
with ‹*m1* ≤ *m2*› **have** *m1* = *m2* **by** (*rule le-antisym*)
with *True* ‹*i* ∈ *set xs*› **show** *?thesis* **by** (*simp add*: *lookup-pm-of-idx-pm m1-def m2-def cong*: *conj-cong*)
next
case *False*
thus *?thesis*
proof (*simp add*: *lookup-pm-of-idx-pm when-def m1-def*[*symmetric*], *intro impI*)
assume *i* ∈ *set xs*
hence *m1* ∈ *?A* **by** (*rule m1*)
hence *m1* < *length xs* **and** *xs* ! *m1* = *i* **by** *simp-all*
have *m1* ∉ *keys f*
proof
assume *m1* ∈ *keys f*
hence *m1* ∈ {*0..<j*} **using** *assms* **..**
hence *m1* < *j* **by** *simp*
with ‹*m1* < *length xs*› **have** *m1* < *length ?xs* **by** *simp*
hence *?xs* ! *m1* ∈ *set ?xs* **by** (*rule nth-mem*)
with ‹*m1* < *j*› **have** *i* ∈ *set ?xs* **by** (*simp add*: ‹*xs* ! *m1* = *i*›)
with *False* **show** *False* **..**
qed
thus *lookup f m1* = *0* **by** (*simp add*: *in-keys-iff*)

    qed
  qed
qed

lemma *lookup-idx-pm-of-pm*: *lookup* (*idx-pm-of-pm xs f*) = (λ*i*. *lookup f* (*xs* ! *i*) *when i* < *length xs*)
  unfolding *idx-pm-of-pm-def* **by** (*rule Abs-poly-mapping-inverse*, *simp*)

lemma *keys-idx-pm-of-pm-subset*: *keys* (*idx-pm-of-pm xs f*) ⊆ {*0*..<*length xs*}
proof
  fix *i*
  assume *i* ∈ *keys* (*idx-pm-of-pm xs f*)
  hence *lookup* (*idx-pm-of-pm xs f*) *i* ≠ *0* **by** (*simp add*: *in-keys-iff*)
  thus *i* ∈ {*0*..<*length xs*} **by** (*simp add*: *lookup-idx-pm-of-pm*)
qed

lemma *idx-pm-of-pm-zero* [*simp*]: *idx-pm-of-pm xs 0* = *0*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-idx-pm-of-pm*)

lemma *idx-pm-of-pm-plus*: *idx-pm-of-pm xs* (*f* + *g*) = *idx-pm-of-pm xs f* + *idx-pm-of-pm xs g*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-idx-pm-of-pm lookup-add when-def*)

lemma *idx-pm-of-pm-minus*: *idx-pm-of-pm xs* (*f* − *g*) = *idx-pm-of-pm xs f* − *idx-pm-of-pm xs g*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-idx-pm-of-pm lookup-minus when-def*)

lemma *pm-of-idx-pm-of-pm*:
  assumes *keys f* ⊆ *set xs*
  shows *pm-of-idx-pm xs* (*idx-pm-of-pm xs f*) = *f*
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-pm-of-idx-pm when-def*, *intro conjI impI*)
  fix *k*
  assume *k* ∈ *set xs*
  define *i* **where** *i* = *Min* {*i*. *i* < *length xs* ∧ *xs* ! *i* = *k*}
  have *finite* {*i*. *i* < *length xs* ∧ *xs* ! *i* = *k*} **by** *simp*
  moreover from ‹*k* ∈ *set xs*› **have** {*i*. *i* < *length xs* ∧ *xs* ! *i* = *k*} ≠ {}
    by (*simp add*: *in-set-conv-nth*)
  ultimately have *i* ∈ {*i*. *i* < *length xs* ∧ *xs* ! *i* = *k*} **unfolding** *i-def* **by** (*rule Min-in*)
  hence *i* < *length xs* **and** *xs* ! *i* = *k* **by** *simp-all*
  thus *lookup* (*idx-pm-of-pm xs f*) *i* = *lookup f k* **by** (*simp add*: *lookup-idx-pm-of-pm*)
next
  fix *k*
  assume *k* ∉ *set xs*
  with *assms* **show** *lookup f k* = *0* **by** (*auto simp*: *in-keys-iff*)
qed

lemma *idx-pm-of-pm-of-idx-pm*:

assumes *distinct xs* **and** *keys f* ⊆ {*0..<length xs*}
shows *idx-pm-of-pm xs* (*pm-of-idx-pm xs f*) = *f*
proof (*rule poly-mapping-eqI*)
fix *i*
show *lookup* (*idx-pm-of-pm xs* (*pm-of-idx-pm xs f*)) *i* = *lookup f i*
proof (*cases i* < *length xs*)
case *True*
with *assms*(*1*) **show** *?thesis* **by** (*simp add*: *lookup-idx-pm-of-pm lookup-pm-of-idx-pm-distinct*)
next
case *False*
hence *i* ∉ {*0..<length xs*} **by** *simp*
with *assms*(*2*) **have** *i* ∉ *keys f* **by** *blast*
with *False* **show** *?thesis* **by** (*simp add*: *in-keys-iff lookup-idx-pm-of-pm*)
qed
qed

18.3 POT Orders

context *ordered-term*
begin

definition *is-pot-ord* :: *bool*
where *is-pot-ord* ⟷ (∀ *u v*. *component-of-term u* < *component-of-term v* ⟶ $u \prec_t v$)

lemma *is-pot-ordI*:
assumes ⋀*u v*. *component-of-term u* < *component-of-term v* ⟹ $u \prec_t v$
shows *is-pot-ord*
unfolding *is-pot-ord-def* **using** *assms* **by** *blast*

lemma *is-pot-ordD*:
assumes *is-pot-ord* **and** *component-of-term u* < *component-of-term v*
shows $u \prec_t v$
using *assms* **unfolding** *is-pot-ord-def* **by** *blast*

lemma *is-pot-ordD2*:
assumes *is-pot-ord* **and** $u \preceq_t v$
shows *component-of-term u* ≤ *component-of-term v*
proof (*rule ccontr*)
assume ¬ *component-of-term u* ≤ *component-of-term v*
hence *component-of-term v* < *component-of-term u* **by** *simp*
with *assms*(*1*) **have** $v \prec_t u$ **by** (*rule is-pot-ordD*)
with *assms*(*2*) **show** *False* **by** *simp*
qed

lemma *is-pot-ord*:
assumes *is-pot-ord*
shows $u \preceq_t v$ ⟷ (*component-of-term u* < *component-of-term v* ∨
(*component-of-term u* = *component-of-term v* ∧ *pp-of-term u* ⪯

pp-of-term v)) (**is** *?l* ⟷ *?r*)
proof
assume *?l*
with *assms* **have** *component-of-term u* ≤ *component-of-term v* **by** (*rule is-pot-ordD2*)
hence *component-of-term u* < *component-of-term v* ∨ *component-of-term u* = *component-of-term v*
by (*simp add*: *order-class.le-less*)
thus *?r*
proof
assume *component-of-term u* < *component-of-term v*
thus *?r* **..**
next
assume *1*: *component-of-term u* = *component-of-term v*
moreover have *pp-of-term u* ⪯ *pp-of-term v*
proof (*rule ccontr*)
assume ¬ *pp-of-term u* ⪯ *pp-of-term v*
hence *2*: *pp-of-term v* ⪯ *pp-of-term u* **and** *3*: *pp-of-term u* ≠ *pp-of-term v* **by** *simp-all*
from *1* **have** *component-of-term v* ≤ *component-of-term u* **by** *simp*
with *2* **have** $v \preceq_t u$ **by** (*rule ord-termI*)
with ‹*?l*› **have** *u* = *v* **by** *simp*
with *3* **show** *False* **by** *simp*
qed
ultimately show *?r* **by** *simp*
qed
next
assume *?r*
thus *?l*
proof
assume *component-of-term u* < *component-of-term v*
with *assms* **have** $u \prec_t v$ **by** (*rule is-pot-ordD*)
thus *?l* **by** *simp*
next
assume *component-of-term u* = *component-of-term v* ∧ *pp-of-term u* ⪯ *pp-of-term v*
hence *pp-of-term u* ⪯ *pp-of-term v* **and** *component-of-term u* ≤ *component-of-term v* **by** *simp-all*
thus *?l* **by** (*rule ord-termI*)
qed
qed

definition *map-component* :: (*'k* ⇒ *'k*) ⇒ *'t* ⇒ *'t*
where *map-component f v* = *term-of-pair* (*pp-of-term v*, *f* (*component-of-term v*))

lemma *pair-of-map-component* [*term-simps*]:
pair-of-term (*map-component f v*) = (*pp-of-term v*, *f* (*component-of-term v*))
by (*simp add*: *map-component-def pair-term*)

lemma *pp-of-map-component* [*term-simps*]: *pp-of-term* (*map-component f v*) = *pp-of-term v*
  by (*simp add*: *pp-of-term-def pair-of-map-component*)

lemma *component-of-map-component* [*term-simps*]:
  component-of-term (*map-component f v*) = *f* (*component-of-term v*)
  by (*simp add*: *component-of-term-def pair-of-map-component*)

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

lemma *map-component-comp*: *map-component f* (*map-component g x*) = *map-component* (*λk. f* (*g k*)) *x*
  by (*simp add*: *map-component-def term-simps*)

lemma *map-component-id* [*term-simps*]: *map-component* (*λk. k*) *x* = *x*
  by (*simp add*: *map-component-def term-simps*)

lemma *map-component-inj*:
  assumes *inj f* **and** *map-component f u* = *map-component f v*
  shows *u* = *v*
proof −
  from *assms*(*2*) **have** *term-of-pair* (*pp-of-term u*, *f* (*component-of-term u*)) =
                         term-of-pair (*pp-of-term v*, *f* (*component-of-term v*))
    by (*simp only*: *map-component-def*)
  hence (*pp-of-term u*, *f* (*component-of-term u*)) = (*pp-of-term v*, *f* (*component-of-term v*))
    by (*rule term-of-pair-injective*)
  hence *1*: *pp-of-term u* = *pp-of-term v* **and** *f* (*component-of-term u*) = *f* (*component-of-term v*) **by** *simp-all*
  from *assms*(*1*) *this*(*2*) **have** *component-of-term u* = *component-of-term v* **by** (*rule injD*)
  with *1* **show** *?thesis* **by** (*metis term-of-pair-pair*)
qed

end

18.4 Gröbner Bases of Syzygy Modules

locale *gd-inf-term* =
    gd-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
      for *pair-of-term*::*'t* ⇒ (*'a*::*graded-dickson-powerprod* × *nat*)
      and *term-of-pair*::(*'a* × *nat*) ⇒ *'t*
      and *ord*::*'a* ⇒ *'a* ⇒ *bool* (**infixl** ‹$\preceq$› *50*)
      and *ord-strict* (**infixl** ‹$\prec$› *50*)
      and *ord-term*::*'t* ⇒ *'t* ⇒ *bool* (**infixl** ‹$\preceq_t$› *50*)
      and *ord-term-strict*::*'t* ⇒ *'t* ⇒ *bool* (**infixl** ‹$\prec_t$› *50*)
begin

In order to compute a Gröbner basis of the syzygy module of a list *bs* of polynomials, one first needs to "lift" *bs* to a new list *bs'* by adding further components, compute a Gröbner basis *gs* of *bs'*, and then filter out those elements of *gs* whose only non-zero components are those that were newly added to *bs*. Function *init-syzygy-list* takes care of constructing *bs'*, and function *filter-syzygy-basis* does the filtering. Function *proj-orig-basis*, finally, projects the Gröbner basis *gs* of *bs'* to a Gröbner basis of the original list *bs*.

definition *lift-poly-syz* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*) ⇒ *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*::*semiring-1*)
  where *lift-poly-syz n b i* = *Abs-poly-mapping*
    (*λx. if pair-of-term x* = (*0*, *i*) *then 1*
      else if n ≤ *component-of-term x then lookup b* (*map-component* (*λk.*
k − *n*) *x*)
      else 0)

definition *proj-poly-syz* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*) ⇒ (*'t* $\Rightarrow_0$ *'b*::*semiring-1*)
  where *proj-poly-syz n b* = *Poly-Mapping.map-key* (*λx. map-component* (*λk. k* +
n) *x*) *b*

definition *cofactor-list-syz* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*) ⇒ (*'a* $\Rightarrow_0$ *'b*::*semiring-1*) *list*
  where *cofactor-list-syz n b* = *map* (*λi. proj-poly i b*) [*0*..<*n*]

definition *init-syzygy-list* :: (*'t* $\Rightarrow_0$ *'b*) *list* ⇒ (*'t* $\Rightarrow_0$ *'b*::*semiring-1*) *list*
  where *init-syzygy-list bs* = *map-idx* (*lift-poly-syz* (*length bs*)) *bs 0*

definition *proj-orig-basis* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*) *list* ⇒ (*'t* $\Rightarrow_0$ *'b*::*semiring-1*) *list*
  where *proj-orig-basis n bs* = *map* (*proj-poly-syz n*) *bs*

definition *filter-syzygy-basis* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*) *list* ⇒ (*'t* $\Rightarrow_0$ *'b*::*semiring-1*) *list*
  where *filter-syzygy-basis n bs* = [*b*←*bs. component-of-term* ` *keys b* ⊆ {*0*..<*n*}]

definition *syzygy-module-list* :: (*'t* $\Rightarrow_0$ *'b*) *list* ⇒ (*'t* $\Rightarrow_0$ *'b*::*comm-ring-1*) *set*
  where *syzygy-module-list bs* = *atomize-poly* ` *idx-pm-of-pm bs* ` *pmdl.syzygy-module*
(*set bs*)

18.4.1 *lift-poly-syz*

lemma *keys-lift-poly-syz-aux*:
  {*x.* (*if pair-of-term x* = (*0*, *i*) *then 1*
    else if n ≤ *component-of-term x then lookup b* (*map-component* (*λk. k* − *n*)
x)
    else 0) ≠ *0*} ⊆ *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (*λk. k* + *n*) `
keys b)
  (**is** *?l* ⊆ *?r*) **for** *b*::*'t* $\Rightarrow_0$ *'b*::*semiring-1*
proof
  fix *x*::*'t*
  assume *x* ∈ *?l*
  hence (*if pair-of-term x* = (*0*, *i*) *then 1 else if n* ≤ *component-of-term x then*

lookup b (*map-component* (λ*k*. *k* − *n*) *x*) *else 0*) ≠ *0*
  by *simp*
 hence *pair-of-term x* = (*0*, *i*) ∨ (*n* ≤ *component-of-term x* ∧ *lookup b* (*map-component* (λ*k*. *k* − *n*) *x*) ≠ *0*)
  by (*simp split*: *if-split-asm*)
 thus *x* ∈ *?r*
 proof
  assume *pair-of-term x* = (*0*, *i*)
  hence (*0*, *i*) = *pair-of-term x* **by** (*rule sym*)
  hence *x* = *term-of-pair* (*0*, *i*) **by** (*simp add*: *term-pair*)
  thus *?thesis* **by** *simp*
 next
  assume *n* ≤ *component-of-term x* ∧ *lookup b* (*map-component* (λ*k*. *k* − *n*) *x*) ≠ *0*
  hence *n* ≤ *component-of-term x* **and** *2*: *map-component* (λ*k*. *k* − *n*) *x* ∈ *keys b*
   by (*auto simp*: *in-keys-iff*)
  from *this*(*1*) **have** *3*: *map-component* (λ*k*. *k* − *n* + *n*) *x* = *x* **by** (*simp add*: *map-component-def term-simps*)
  from *2* **have** *map-component* (λ*k*. *k* + *n*) (*map-component* (λ*k*. *k* − *n*) *x*) ∈ *map-component* (λ*k*. *k* + *n*) ‘ *keys b*
   by (*rule imageI*)
  with *3* **have** *x* ∈ *map-component* (λ*k*. *k* + *n*) ‘ *keys b* **by** (*simp add*: *map-component-comp*)
  thus *?thesis* **by** *simp*
 qed
qed

lemma *lookup-lift-poly-syz*:
 lookup (*lift-poly-syz n b i*) =
  (λ*x*. *if pair-of-term x* = (*0*, *i*) *then 1 else if n* ≤ *component-of-term x then lookup b* (*map-component* (λ*k*. *k* − *n*) *x*) *else 0*)
 unfolding *lift-poly-syz-def*
proof (*rule Abs-poly-mapping-inverse*)
 from *finite-keys* **have** *finite* (*map-component* (λ*k*. *k* + *n*) ‘ *keys b*) **..**
 hence *finite* (*insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λ*k*. *k* + *n*) ‘ *keys b*)) **by** (*rule finite.insertI*)
 with *keys-lift-poly-syz-aux*
 have *finite* {*x*. (*if pair-of-term x* = (*0*, *i*) *then 1*
           else if n ≤ *component-of-term x then lookup b* (*map-component* (λ*k*. *k* − *n*) *x*)
          else 0) ≠ *0*}
  by (*rule finite-subset*)
 thus (λ*x*. *if pair-of-term x* = (*0*, *i*) *then 1*
      else if n ≤ *component-of-term x then lookup b* (*map-component* (λ*k*. *k* − *n*) *x*)
      else 0) ∈
    {*f*. *finite* {*x*. *f x* ≠ *0*}} **by** *simp*
qed

corollary *lookup-lift-poly-syz-alt*:
lookup (*lift-poly-syz n b i*) (*term-of-pair* (*t*, *j*)) =
(*if* (*t*, *j*) = (*0*, *i*) *then 1 else if n* ≤ *j then lookup b* (*term-of-pair* (*t*, *j* − *n*)) *else 0*)
by (*simp only*: *lookup-lift-poly-syz term-simps*)

lemma *keys-lift-poly-syz*:
keys (*lift-poly-syz n b i*) = *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λ*k*. *k* + *n*) ' *keys b*)
proof
have *keys* (*lift-poly-syz n b i*) ⊆
{*x*. (*if pair-of-term x* = (*0*, *i*) *then 1*
else if n ≤ *component-of-term x then lookup b* (*map-component* (λ*k*. *k* − *n*) *x*)
else 0) ≠ *0*}
(**is** - ⊆ *?A*)
proof
fix *x*
assume *x* ∈ *keys* (*lift-poly-syz n b i*)
hence *lookup* (*lift-poly-syz n b i*) *x* ≠ *0* **by** (*simp add*: *in-keys-iff*)
thus *x* ∈ *?A* **by** (*simp add*: *lookup-lift-poly-syz*)
qed
also note *keys-lift-poly-syz-aux*
finally show *keys* (*lift-poly-syz n b i*) ⊆ *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λ*k*. *k* + *n*) ' *keys b*) **.**
next
show *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λ*k*. *k* + *n*) ' *keys b*) ⊆ *keys* (*lift-poly-syz n b i*)
proof (*simp*, *rule*)
have *lookup* (*lift-poly-syz n b i*) (*term-of-pair* (*0*, *i*)) ≠ *0* **by** (*simp add*: *lookup-lift-poly-syz-alt*)
thus *term-of-pair* (*0*, *i*) ∈ *keys* (*lift-poly-syz n b i*) **by** (*simp add*: *in-keys-iff*)
next
show *map-component* (λ*k*. *k* + *n*) ' *keys b* ⊆ *keys* (*lift-poly-syz n b i*)
proof (*rule*, *elim imageE*, *simp*)
fix *x*
assume *x* ∈ *keys b*
hence *lookup* (*lift-poly-syz n b i*) (*map-component* (λ*k*. *k* + *n*) *x*) ≠ *0*
by (*simp add*: *in-keys-iff lookup-lift-poly-syz-alt map-component-def term-simps*)
thus *map-component* (λ*k*. *k* + *n*) *x* ∈ *keys* (*lift-poly-syz n b i*) **by** (*simp add*: *in-keys-iff*)
qed
qed
qed

18.4.2 *proj-poly-syz*

lemma *inj-map-component-plus*: *inj* (*map-component* (λ*k*. *k* + *n*))
proof (*rule injI*)

fix *x y*
have *inj* (λ*k*::*nat*. *k* + *n*) **by** (*simp add*: *inj-def*)
moreover assume *map-component* (λ*k*. *k* + *n*) *x* = *map-component* (λ*k*. *k* + *n*) *y*
ultimately show *x* = *y* **by** (*rule map-component-inj*)
qed

lemma *lookup-proj-poly-syz*: *lookup* (*proj-poly-syz n p*) *x* = *lookup p* (*map-component* (λ*k*. *k* + *n*) *x*)
by (*simp add*: *proj-poly-syz-def map-key.rep-eq*[*OF inj-map-component-plus*])

lemma *lookup-proj-poly-syz-alt*:
lookup (*proj-poly-syz n p*) (*term-of-pair* (*t*, *i*)) = *lookup p* (*term-of-pair* (*t*, *i* + *n*))
by (*simp add*: *lookup-proj-poly-syz map-component-term-of-pair*)

lemma *keys-proj-poly-syz*: *keys* (*proj-poly-syz n p*) = *map-component* (λ*k*. *k* + *n*) −' *keys p*
by (*simp add*: *proj-poly-syz-def keys-map-key*[*OF inj-map-component-plus*])

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

lemma *proj-poly-syz-plus*: *proj-poly-syz n* (*p* + *q*) = *proj-poly-syz n p* + *proj-poly-syz n q*
by (*simp add*: *proj-poly-syz-def map-key-plus*[*OF inj-map-component-plus*])

lemma *proj-poly-syz-sum*: *proj-poly-syz n* (*sum f A*) = (∑ *a*∈*A*. *proj-poly-syz n* (*f a*))
by (*rule fun-sum-commute*, *simp-all add*: *proj-poly-syz-plus*)

lemma *proj-poly-syz-sum-list*: *proj-poly-syz n* (*sum-list xs*) = *sum-list* (*map* (*proj-poly-syz n*) *xs*)
by (*rule fun-sum-list-commute*, *simp-all add*: *proj-poly-syz-plus*)

lemma *proj-poly-syz-monom-mult*:
proj-poly-syz n (*monom-mult c t p*) = *monom-mult c t* (*proj-poly-syz n p*)
by (*rule poly-mapping-eqI*,
simp add: *lookup-proj-poly-syz lookup-monom-mult term-simps adds-pp-def sminus-def*)

lemma *proj-poly-syz-mult-scalar*:
proj-poly-syz n (*mult-scalar q p*) = *mult-scalar q* (*proj-poly-syz n p*)
by (*rule fun-mult-scalar-commute*, *simp-all add*: *proj-poly-syz-plus proj-poly-syz-monom-mult*)

lemma *proj-poly-syz-lift-poly-syz*:
assumes *i* < *n*
shows *proj-poly-syz n* (*lift-poly-syz n p i*) = *p*
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly-syz lookup-lift-poly-syz*

term-simps map-component-comp,
rule, *elim conjE*)
fix *x*::'*t*
assume *component-of-term x* + *n* = *i*
hence *n* ≤ *i* **by** *simp*
with *assms* **show** *lookup p x* = *1* **by** *simp*
qed

lemma *proj-poly-syz-eq-zero-iff*: *proj-poly-syz n p* = *0* ⟷ (*component-of-term* ' *keys p* ⊆ {*0*..<*n*})
unfolding *keys-eq-empty*[*symmetric*] *keys-proj-poly-syz*
proof
assume *map-component* (λ*k*. *k* + *n*) −' *keys p* = {} (**is** *?A* = {})
show *component-of-term* ' *keys p* ⊆ {*0*..<*n*}
proof (*rule*, *rule ccontr*)
fix *i*
assume *i* ∈ *component-of-term* ' *keys p*
then obtain *x* **where** *x*: *x* ∈ *keys p* **and** *i*: *i* = *component-of-term x* **..**
assume *i* ∉ {*0*..<*n*}
hence *i* − *n* + *n* = *i* **by** *simp*
hence *1*: *map-component* (λ*k*. *k* − *n* + *n*) *x* = *x* **by** (*simp add*: *map-component-def i term-simps*)
have *map-component* (λ*k*. *k* − *n*) *x* ∈ *?A* **by** (*rule vimageI2*, *simp add*: *map-component-comp x 1*)
thus *False* **by** (*simp add*: ‹*?A* = {}›)
qed
next
assume *a*: *component-of-term* ' *keys p* ⊆ {*0*..<*n*}
show *map-component* (λ*k*. *k* + *n*) −' *keys p* = {} (**is** *?A* = {})
proof (*rule ccontr*)
assume *?A* ≠ {}
then obtain *x* **where** *x* ∈ *?A* **by** *blast*
hence *map-component* (λ*k*. *k* + *n*) *x* ∈ *keys p* **by** (*rule vimageD*)
with *a* **have** *component-of-term* (*map-component* (λ*k*. *k* + *n*) *x*) ∈ {*0*..<*n*} **by** *blast*
thus *False* **by** (*simp add*: *term-simps*)
qed
qed

lemma *component-of-lt-ge*:
assumes *is-pot-ord* **and** *proj-poly-syz n p* ≠ *0*
shows *n* ≤ *component-of-term* (*lt p*)
proof −
from *assms*(*2*) **have** ¬ *component-of-term* ' *keys p* ⊆ {*0*..<*n*} **by** (*simp add*: *proj-poly-syz-eq-zero-iff*)
then obtain *i* **where** *i* ∈ *component-of-term* ' *keys p* **and** *i* ∉ {*0*..<*n*} **by** *fastforce*
from *this*(*1*) **obtain** *x* **where** *x* ∈ *keys p* **and** *i*: *i* = *component-of-term x* **..**
from *this*(*1*) **have** $x \preceq_t$ *lt p* **by** (*rule lt-max-keys*)

with *assms*(*1*) **have** *component-of-term x* ≤ *component-of-term* (*lt p*) **by** (*rule is-pot-ordD2*)
with ‹*i* ∉ {*0*..<*n*}› **show** *?thesis* **by** (*simp add*: *i*)
qed

lemma *lt-proj-poly-syz*:
assumes *is-pot-ord* **and** *proj-poly-syz n p* ≠ *0*
shows *lt* (*proj-poly-syz n p*) = *map-component* (λ*k*. *k* − *n*) (*lt p*) (**is** - = *?l*)
proof −
from *component-of-lt-ge*[*OF assms*]
have *component-of-term* (*lt p*) − *n* + *n* = *component-of-term* (*lt p*) **by** *simp*
hence *eq*: *map-component* (λ*k*. *k* − *n* + *n*) (*lt p*) = *lt p* **by** (*simp add*: *map-component-def term-simps*)
show *?thesis*
proof (*rule lt-eqI*)
have *lookup* (*proj-poly-syz n p*) *?l* = *lc p*
by (*simp add*: *lc-def lookup-proj-poly-syz term-simps map-component-comp eq*)
also have ... ≠ *0*
proof (*rule lc-not-0*, *rule*)
assume *p* = *0*
hence *proj-poly-syz n p* = *0* **by** *simp*
with *assms*(*2*) **show** *False* **..**
qed
finally show *lookup* (*proj-poly-syz n p*) *?l* ≠ *0* **.**
next
fix *x*
assume *lookup* (*proj-poly-syz n p*) *x* ≠ *0*
hence *map-component* (λ*k*. *k* + *n*) *x* ∈ *keys p* **by** (*simp add*: *in-keys-iff lookup-proj-poly-syz*)
hence *map-component* (λ*k*. *k* + *n*) *x* $\preceq_t$ *lt p* **by** (*rule lt-max-keys*)
with *assms*(*1*) **show** *x* $\preceq_t$ *?l* **by** (*auto simp add*: *is-pot-ord term-simps*)
qed
qed

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

lemma *poly-mapping-eqI-proj-syz*:
assumes *proj-poly-syz n p* = *proj-poly-syz n q*
and ⋀*k*. *k* < *n* ⟹ *proj-poly k p* = *proj-poly k q*
shows *p* = *q*
proof (*rule poly-mapping-eqI-proj*)
fix *k*
show *proj-poly k p* = *proj-poly k q*
proof (*cases k* < *n*)
case *True*
thus *?thesis* **by** (*rule assms*(*2*))
next
case *False*

have *proj-poly* (*k* − *n* + *n*) *p* = *proj-poly* (*k* − *n* + *n*) *q*
by (*simp only*: *proj-proj-poly-syz*[*symmetric*] *assms*(*1*))
with *False* **show** *?thesis* **by** *simp*
qed
qed

18.4.3 *cofactor-list-syz*

lemma *length-cofactor-list-syz* [*simp*]: *length* (*cofactor-list-syz n p*) = *n*
by (*simp add*: *cofactor-list-syz-def*)

lemma *cofactor-list-syz-nth*:
assumes *i* < *n*
shows (*cofactor-list-syz n p*) ! *i* = *proj-poly i p*
by (*simp add*: *cofactor-list-syz-def map-idx-nth assms*)

lemma *cofactor-list-syz-zero* [*simp*]: *cofactor-list-syz n 0* = *replicate n 0*
by (*rule nth-equalityI*, *simp-all add*: *cofactor-list-syz-nth proj-zero*)

lemma *cofactor-list-syz-plus*:
cofactor-list-syz n (*p* + *q*) = *map2* (+) (*cofactor-list-syz n p*) (*cofactor-list-syz n q*)
by (*rule nth-equalityI*, *simp-all add*: *cofactor-list-syz-nth proj-plus*)

18.4.4 *init-syzygy-list*

lemma *length-init-syzygy-list* [*simp*]: *length* (*init-syzygy-list bs*) = *length bs*
by (*simp add*: *init-syzygy-list-def*)

lemma *init-syzygy-list-nth*:
assumes *i* < *length bs*
shows (*init-syzygy-list bs*) ! *i* = *lift-poly-syz* (*length bs*) (*bs* ! *i*) *i*
by (*simp add*: *init-syzygy-list-def map-idx-nth*[*OF assms*])

lemma *Keys-init-syzygy-list*:
Keys (*set* (*init-syzygy-list bs*)) =
map-component (λ*k*. *k* + *length bs*) ‘ *Keys* (*set bs*) ∪ (λ*i*. *term-of-pair* (*0*, *i*)) ‘ {*0*..<*length bs*}
proof −
have *eq1*: (⋃*b*∈*set bs*. *map-component* (λ*k*. *k* + *length bs*) ‘ *keys b*) =
(⋃*i*∈{*0*..<*length bs*}. *map-component* (λ*k*. *k* + *length bs*) ‘ *keys* (*bs* ! *i*))
by (*fact UN-upt*[*symmetric*])
have *eq2*: (λ*i*. *term-of-pair* (*0*, *i*)) ‘ {*0*..<*length bs*} = (⋃*i*∈{*0*..<*length bs*}. {*term-of-pair* (*0*, *i*)})
by *auto*
show *?thesis*
by (*simp add*: *init-syzygy-list-def set-map-idx Keys-def keys-lift-poly-syz image-UN*
eq1 eq2 UN-Un-distrib[*symmetric*])

qed

lemma *pp-of-Keys-init-syzygy-list-subset*:
pp-of-term ‘ Keys (set (init-syzygy-list bs)) ⊆ insert 0 (pp-of-term ‘ Keys (set bs))
by (*auto simp add*: *Keys-init-syzygy-list image-Un rev-image-eqI term-simps*)

lemma *pp-of-Keys-init-syzygy-list-superset*:
pp-of-term ‘ Keys (set bs) ⊆ pp-of-term ‘ Keys (set (init-syzygy-list bs))
by (*simp add*: *Keys-init-syzygy-list image-Un term-simps image-image*)

lemma *pp-of-Keys-init-syzygy-list*:
assumes *bs ≠ []*
shows *pp-of-term ‘ Keys (set (init-syzygy-list bs)) = insert 0 (pp-of-term ‘ Keys (set bs))*
proof
show *insert 0 (pp-of-term ‘ Keys (set bs)) ⊆ pp-of-term ‘ Keys (set (init-syzygy-list bs))*
proof (*simp add*: *pp-of-Keys-init-syzygy-list-superset*)
from *assms* **have** *{0..<length bs} ≠ {}* **by** *auto*
hence *Pair 0 ‘ {0..<length bs} ≠ {}* **by** *blast*
then obtain *x::'t* **where** *x*: *x ∈ (λi. term-of-pair (0, i)) ‘ {0..<length bs}* **by** *blast*
hence *pp-of-term ‘ (λi. term-of-pair (0, i)) ‘ {0..<length bs} = {pp-of-term x}*
using *image-subset-iff* **by** (*auto simp*: *term-simps*)
also from *x* **have** ... *= {0}* **using** *pp-of-term-of-pair* **by** *auto*
finally show *0 ∈ pp-of-term ‘ Keys (set (init-syzygy-list bs))*
by (*simp add*: *Keys-init-syzygy-list image-Un*)
qed
qed (*fact pp-of-Keys-init-syzygy-list-subset*)

lemma *component-of-Keys-init-syzygy-list*:
component-of-term ‘ Keys (set (init-syzygy-list bs)) =
(+) (length bs) ‘ component-of-term ‘ Keys (set bs) ∪ {0..<length bs}
by (*simp add*: *Keys-init-syzygy-list image-Un image-comp o-def ac-simps term-simps*)

lemma *proj-lift-poly-syz*:
assumes *j < n*
shows *proj-poly j (lift-poly-syz n p i) = (1 when j = i)*
proof (*simp add*: *when-def*, *intro conjI impI*)
assume *j = i*
with *assms* **have** *¬ n ≤ i* **by** *simp*
show *proj-poly i (lift-poly-syz n p i) = 1*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-lift-poly-syz-alt ‹¬ n ≤ i› lookup-one*)
next
assume *j ≠ i*
from *assms* **have** *¬ n ≤ j* **by** *simp*
show *proj-poly j (lift-poly-syz n p i) = 0*

by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-lift-poly-syz-alt* ‹¬ $n \leq j$› ‹$j \neq i$›)
qed

18.4.5 *proj-orig-basis*

lemma *length-proj-orig-basis* [*simp*]: *length* (*proj-orig-basis n bs*) = *length bs*
by (*simp add*: *proj-orig-basis-def*)

lemma *proj-orig-basis-nth*:
assumes $i <$ *length bs*
shows (*proj-orig-basis n bs*) ! i = *proj-poly-syz n* (*bs* ! i)
by (*simp add*: *proj-orig-basis-def assms*)

lemma *proj-orig-basis-init-syzygy-list* [*simp*]:
proj-orig-basis (*length bs*) (*init-syzygy-list bs*) = *bs*
by (*rule nth-equalityI*, *simp-all add*: *init-syzygy-list-nth proj-orig-basis-nth proj-poly-syz-lift-poly-syz*)

lemma *set-proj-orig-basis*: *set* (*proj-orig-basis n bs*) = *proj-poly-syz n* ‘ *set bs*
by (*simp add*: *proj-orig-basis-def*)

The following lemma could be generalized from *proj-poly-syz* to arbitrary module homomorphisms, i. e. functions respecting *0*, addition and scalar multiplication.

lemma *pmdl-proj-orig-basis'*:
pmdl (*set* (*proj-orig-basis n bs*)) = *proj-poly-syz n* ‘ *pmdl* (*set bs*) (**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix *p*
assume *p* ∈ *pmdl* (*set* (*proj-orig-basis n bs*))
thus *p* ∈ *proj-poly-syz n* ‘ *pmdl* (*set bs*)
proof (*induct rule*: *pmdl-induct*)
case *module-0*
have *0* = *proj-poly-syz n 0* **by** *simp*
also from *pmdl.span-zero* **have** ... ∈ *proj-poly-syz n* ‘ *pmdl* (*set bs*) **by** (*rule imageI*)
finally show *?case* **.**
next
case (*module-plus p b c t*)
from *module-plus*(*2*) **obtain** *q* **where** *q* ∈ *pmdl* (*set bs*) **and** *p*: *p* = *proj-poly-syz n q* **..**
from *module-plus*(*3*) **obtain** *a* **where** *a* ∈ *set bs* **and** *b*: *b* = *proj-poly-syz n a*
unfolding *set-proj-orig-basis* **..**
have *p* + *monom-mult c t b* = *proj-poly-syz n* (*q* + *monom-mult c t a*)
by (*simp add*: *p b proj-poly-syz-monom-mult proj-poly-syz-plus*)
also have ... ∈ *proj-poly-syz n* ‘ *pmdl* (*set bs*)
proof (*rule imageI*, *rule pmdl.span-add*)

```
      show monom-mult c t a ∈ pmdl (set bs)
        by (rule pmdl-closed-monom-mult, rule pmdl.span-base, fact)
    qed fact
    finally show ?case .
  qed
 qed
next
 show ?B ⊆ ?A
 proof
  fix p
  assume p ∈ proj-poly-syz n ` pmdl (set bs)
  then obtain q where q ∈ pmdl (set bs) and p: p = proj-poly-syz n q ..
  from this(1) show p ∈ pmdl (set (proj-orig-basis n bs)) unfolding p
  proof (induct rule: pmdl-induct)
    case module-0
    have proj-poly-syz n 0 = 0 by simp
    also have ... ∈ pmdl (set (proj-orig-basis n bs)) by (fact pmdl.span-zero)
    finally show ?case .
  next
    case (module-plus q b c t)
    have proj-poly-syz n (q + monom-mult c t b) =
          proj-poly-syz n q + monom-mult c t (proj-poly-syz n b)
      by (simp add: proj-poly-syz-plus proj-poly-syz-monom-mult)
    also have ... ∈ pmdl (set (proj-orig-basis n bs))
    proof (rule pmdl.span-add)
      show monom-mult c t (proj-poly-syz n b) ∈ pmdl (set (proj-orig-basis n bs))
      proof (rule pmdl-closed-monom-mult, rule pmdl.span-base)
        show proj-poly-syz n b ∈ set (proj-orig-basis n bs)
          by (simp add: set-proj-orig-basis, rule imageI, fact)
      qed
    qed fact
    finally show ?case .
  qed
 qed
qed
```

18.4.6 *filter-syzygy-basis*

```
lemma filter-syzygy-basis-alt: filter-syzygy-basis n bs = [b←bs. proj-poly-syz n b =
0]
 by (simp add: filter-syzygy-basis-def proj-poly-syz-eq-zero-iff)

lemma set-filter-syzygy-basis:
 set (filter-syzygy-basis n bs) = {b∈set bs. proj-poly-syz n b = 0}
 by (simp add: filter-syzygy-basis-alt)
```

18.4.7 *syzygy-module-list*

```
lemma syzygy-module-listI:
```

assumes *s*′ ∈ *pmdl.syzygy-module* (*set bs*) **and** *s* = *atomize-poly* (*idx-pm-of-pm bs s*′)
shows *s* ∈ *syzygy-module-list bs*
unfolding *assms*(*2*) *syzygy-module-list-def* **by** (*intro imageI*, *fact assms*(*1*))

lemma *syzygy-module-listE*:
assumes *s* ∈ *syzygy-module-list bs*
obtains *s*′ **where** *s*′ ∈ *pmdl.syzygy-module* (*set bs*) **and** *s* = *atomize-poly* (*idx-pm-of-pm bs s*′)
using *assms* **unfolding** *syzygy-module-list-def* **by** (*elim imageE*, *simp*)

lemma *monom-mult-atomize*:
monom-mult c t (*atomize-poly p*) = *atomize-poly* (*MPoly-Type-Class.punit.monom-mult* (*monomial c t*) *0 p*)
by (*rule poly-mapping-eqI-proj*, *simp add*: *proj-monom-mult proj-atomize-poly MPoly-Type-Class.punit.lookup-monom-mult times-monomial-left*)

lemma *punit-monom-mult-monomial-idx-pm-of-pm*:
MPoly-Type-Class.punit.monom-mult (*monomial c t*) (*0*::*nat*) (*idx-pm-of-pm bs s*) =
idx-pm-of-pm bs (*MPoly-Type-Class.punit.monom-mult* (*monomial c t*) (*0*::′*t* $\Rightarrow_0$ ′*b*::*ring-1*) *s*)
by (*rule poly-mapping-eqI*, *simp add*: *MPoly-Type-Class.punit.lookup-monom-mult lookup-idx-pm-of-pm when-def*)

lemma *syzygy-module-list-closed-monom-mult*:
assumes *s* ∈ *syzygy-module-list bs*
shows *monom-mult c t s* ∈ *syzygy-module-list bs*
proof −
from *assms* **obtain** *s*′ **where** *s*′: *s*′ ∈ *pmdl.syzygy-module* (*set bs*)
and *s*: *s* = *atomize-poly* (*idx-pm-of-pm bs s*′) **by** (*rule syzygy-module-listE*)
show *?thesis* **unfolding** *s*
proof (*rule syzygy-module-listI*)
from *s*′ **show** (*monomial c t*) · *s*′ ∈ *pmdl.syzygy-module* (*set bs*)
by (*rule syzygy-module-closed-map-scale*)
next
show *monom-mult c t* (*atomize-poly* (*idx-pm-of-pm bs s*′)) =
atomize-poly (*idx-pm-of-pm bs* ((*monomial c t*) · *s*′))
by (*simp add*: *monom-mult-atomize punit-monom-mult-monomial-idx-pm-of-pm MPoly-Type-Class.punit.map-scale-eq-monom-mult*)
qed
qed

lemma *pmdl-syzygy-module-list* [*simp*]: *pmdl* (*syzygy-module-list bs*) = *syzygy-module-list bs*
proof (*rule pmdl-idI*)
show *0* ∈ *syzygy-module-list bs*
by (*rule syzygy-module-listI*, *fact pmdl.zero-in-syzygy-module*, *simp add*: *atomize-zero*)

next
  fix *s1 s2*
  assume *s1* ∈ *syzygy-module-list bs*
  then obtain *s1′* **where** *s1′*: *s1′* ∈ *pmdl.syzygy-module* (*set bs*)
   and *s1*: *s1* = *atomize-poly* (*idx-pm-of-pm bs s1′*) **by** (*rule syzygy-module-listE*)
  assume *s2* ∈ *syzygy-module-list bs*
  then obtain *s2′* **where** *s2′*: *s2′* ∈ *pmdl.syzygy-module* (*set bs*)
   and *s2*: *s2* = *atomize-poly* (*idx-pm-of-pm bs s2′*) **by** (*rule syzygy-module-listE*)
  show *s1* + *s2* ∈ *syzygy-module-list bs*
  proof (*rule syzygy-module-listI*)
   from *s1′ s2′* **show** *s1′* + *s2′* ∈ *pmdl.syzygy-module* (*set bs*)
    by (*rule pmdl.syzygy-module-closed-plus*)
  next
   show *s1* + *s2* = *atomize-poly* (*idx-pm-of-pm bs* (*s1′* + *s2′*))
    by (*simp add*: *idx-pm-of-pm-plus atomize-plus s1 s2*)
  qed
qed (*fact syzygy-module-list-closed-monom-mult*)

The following lemma also holds without the distinctness constraint on *bs*, but then the proof becomes more difficult.

lemma *syzygy-module-listI′*:
 assumes *distinct bs* **and** *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *s*) *bs*) = *0*
   and *component-of-term* ‘ *keys s* ⊆ {*0*..<*length bs*}
 shows *s* ∈ *syzygy-module-list bs*
proof (*rule syzygy-module-listI*)
 show *pm-of-idx-pm bs* (*vectorize-poly s*) ∈ *pmdl.syzygy-module* (*set bs*)
 proof (*rule pmdl.syzygy-moduleI*, *rule pmdl.representsI*)
  have (∑ *v*∈*keys* (*pm-of-idx-pm bs* (*vectorize-poly s*)).
     mult-scalar (*lookup* (*pm-of-idx-pm bs* (*vectorize-poly s*)) *v*) *v*) =
    (∑ *b*∈*set bs*. *mult-scalar* (*lookup* (*pm-of-idx-pm bs* (*vectorize-poly s*)) *b*) *b*)
   by (*rule sum.mono-neutral-left*, *fact finite-set*, *fact keys-pm-of-idx-pm-subset*, *simp add*: *in-keys-iff*)
   also have ... = *sum-list* (*map* (λ*b*. *mult-scalar* (*lookup* (*pm-of-idx-pm bs* (*vectorize-poly s*)) *b*) *b*) *bs*)
   by (*simp only*: *sum-code distinct-remdups-id*[*OF assms*(*1*)])
  also have ... = *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *s*) *bs*)
  proof (*rule arg-cong*[*of - - sum-list*], *rule nth-equalityI*, *simp-all*)
   fix *i*
   assume *i* < *length bs*
   with *assms*(*1*) **have** *lookup* (*pm-of-idx-pm bs* (*vectorize-poly s*)) (*bs* ! *i*) =
      cofactor-list-syz (*length bs*) *s* ! *i*
   by (*simp add*: *lookup-pm-of-idx-pm-distinct*[*OF assms*(*1*)] *cofactor-list-syz-nth lookup-vectorize-poly*)
   thus *mult-scalar* (*lookup* (*pm-of-idx-pm bs* (*vectorize-poly s*)) (*bs* ! *i*)) (*bs* ! *i*) =
     mult-scalar (*cofactor-list-syz* (*length bs*) *s* ! *i*) (*bs* ! *i*) **by** (*simp only*:)
  qed
  also have ... = *0* **by** (*fact assms*(*2*))

finally show *pmdl.rep* (*pm-of-idx-pm bs* (*vectorize-poly s*)) = *0* **by** (*simp only*: *pmdl.rep-def*)
qed (*fact keys-pm-of-idx-pm-subset*)
next
from *assms*(*3*) **have** *keys* (*vectorize-poly s*) ⊆ {*0*..<*length bs*} **by** (*simp add*: *keys-vectorize-poly*)
with *assms*(*1*) **have** *idx-pm-of-pm bs* (*pm-of-idx-pm bs* (*vectorize-poly s*)) = *vectorize-poly s*
by (*rule idx-pm-of-pm-of-idx-pm*)
thus *s* = *atomize-poly* (*idx-pm-of-pm bs* (*pm-of-idx-pm bs* (*vectorize-poly s*)))
by (*simp add*: *atomize-vectorize-poly*)
qed

lemma *component-of-syzygy-module-list*:
assumes *s* ∈ *syzygy-module-list bs*
shows *component-of-term* ‘ *keys s* ⊆ {*0*..<*length bs*}
proof −
from *assms* **obtain** *s′* **where** *s*: *s* = *atomize-poly* (*idx-pm-of-pm bs s′*)
by (*rule syzygy-module-listE*)
have *component-of-term* ‘ *keys s* ⊆ (⋃*x*∈{*0*..<*length bs*}. {*x*})
by (*simp only*: *s keys-atomize-poly image-UN*, *rule UN-mono*, *fact keys-idx-pm-of-pm-subset*, *auto simp*: *term-simps*)
also have ... = {*0*..<*length bs*} **by** *simp*
finally show *?thesis* **.**
qed

lemma *map2-mult-scalar-proj-poly-syz*:
map2 mult-scalar xs (*map* (*proj-poly-syz n*) *ys*) =
map (*proj-poly-syz n* ∘ (λ(*x*, *y*). *mult-scalar x y*)) (*zip xs ys*)
by (*rule nth-equalityI*, *simp-all add*: *proj-poly-syz-mult-scalar*)

lemma *map2-times-proj*:
map2 (∗) *xs* (*map* (*proj-poly k*) *ys*) = *map* (*proj-poly k* ∘ (λ(*x*, *y*). *x* ⊙ *y*)) (*zip xs ys*)
by (*rule nth-equalityI*, *simp-all add*: *proj-mult-scalar*)

Probably the following lemma also holds without the distinctness constraint on *bs*.

lemma *syzygy-module-list-subset*:
assumes *distinct bs*
shows *syzygy-module-list bs* ⊆ *pmdl* (*set* (*init-syzygy-list bs*))
proof
let *?as* = *init-syzygy-list bs*
fix *s*
assume *s* ∈ *syzygy-module-list bs*
then obtain *s′* **where** *s′*: *s′* ∈ *pmdl.syzygy-module* (*set bs*)
and *s*: *s* = *atomize-poly* (*idx-pm-of-pm bs s′*) **by** (*rule syzygy-module-listE*)
from *s′* **have** *pmdl.represents* (*set bs*) *s′ 0* **by** (*rule pmdl.syzygy-moduleD*)
hence *keys s′* ⊆ *set bs* **and** *1*: *0* = *pmdl.rep s′*

```
    by (rule pmdl.representsD1, rule pmdl.representsD2)
  have s = sum-list (map2 mult-scalar (cofactor-list-syz (length bs) s) (init-syzygy-list
bs))
    (is - = ?r)
  proof (rule poly-mapping-eqI-proj-syz)
    have proj-poly-syz (length bs) ?r =
            sum-list (map2 mult-scalar (cofactor-list-syz (length bs) s)
                                        (map (proj-poly-syz (length bs)) (init-syzygy-list
bs)))
      by (simp add: proj-poly-syz-sum-list map2-mult-scalar-proj-poly-syz)
    also have ... = sum-list (map2 mult-scalar (cofactor-list-syz (length bs) s) bs)
      by (simp add: proj-orig-basis-def[symmetric])
    also have ... = sum-list (map (λb. mult-scalar (lookup s′ b) b) bs)
    proof (rule arg-cong[of - - sum-list], rule nth-equalityI, simp-all)
      fix i
      assume i < length bs
      with assms(1) have lookup s′ (bs ! i) = cofactor-list-syz (length bs) s ! i
        by (simp add: s cofactor-list-syz-nth lookup-idx-pm-of-pm proj-atomize-poly)
      thus mult-scalar (cofactor-list-syz (length bs) s ! i) (bs ! i) =
            mult-scalar (lookup s′ (bs ! i)) (bs ! i) by (simp only:)
    qed
    also have ... = (∑b∈set bs. mult-scalar (lookup s′ b) b)
      by (simp only: sum-code distinct-remdups-id[OF assms])
    also have ... = (∑v∈keys s′. mult-scalar (lookup s′ v) v)
      by (rule sum.mono-neutral-right, fact finite-set, fact, simp add: in-keys-iff)
    also have ... = 0 by (simp add: 1 pmdl.rep-def)
    finally have eq: proj-poly-syz (length bs) ?r = 0 .
    show proj-poly-syz (length bs) s = proj-poly-syz (length bs) ?r
      by (simp add: eq ‹s ∈ syzygy-module-list bs› proj-poly-syz-eq-zero-iff compo-
nent-of-syzygy-module-list)
  next
    fix k
    assume k < length bs
    have proj-poly k s = map2 (∗) (cofactor-list-syz (length bs) s) (map (proj-poly
k)
                                        (init-syzygy-list bs)) ! k
      by (simp add: ‹k < length bs› init-syzygy-list-nth proj-lift-poly-syz cofac-
tor-list-syz-nth)
    also have ... = sum-list (map2 (∗) (cofactor-list-syz (length bs) s)
                                        (map (proj-poly k) (init-syzygy-list bs)))
      by (rule sum-list-eq-nthI[symmetric],
          simp-all add: ‹k < length bs› init-syzygy-list-nth proj-lift-poly-syz)
    also have ... = proj-poly k ?r
      by (simp add: proj-sum-list map2-times-proj)
    finally show proj-poly k s = proj-poly k ?r .
  qed
  also have ... ∈ pmdl (set (init-syzygy-list bs)) by (fact pmdl.span-listI)
  finally show s ∈ pmdl (set (init-syzygy-list bs)) .
qed
```

18.4.8 Cofactors

lemma *map2-mult-scalar-plus*:
map2 (⊙) (*map2* (+) *xs ys*) *zs* = *map2* (+) (*map2* (⊙) *xs zs*) (*map2* (⊙) *ys zs*)
by (*rule nth-equalityI*, *simp-all add*: *mult-scalar-distrib-right*)

lemma *syz-cofactors*:
assumes *p* ∈ *pmdl* (*set* (*init-syzygy-list bs*))
shows *proj-poly-syz* (*length bs*) *p* = *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *p*) *bs*)
using *assms*
proof (*induct rule*: *pmdl-induct*)
case *module-0*
show *?case* **by** (*simp*, *rule sum-list-zeroI'*, *simp*)
next
case (*module-plus p b c t*)
from *this*(*3*) **obtain** *i* **where** *i*: *i* < *length bs* **and** *b*: *b* = (*init-syzygy-list bs*) ! *i*
unfolding *length-init-syzygy-list*[*symmetric*, *of bs*] **by** (*metis in-set-conv-nth*)
have *proj-poly-syz* (*length bs*) (*p* + *monom-mult c t b*) =
proj-poly-syz (*length bs*) *p* + *monom-mult c t* (*bs* ! *i*)
by (*simp only*: *proj-poly-syz-plus proj-poly-syz-monom-mult b init-syzygy-list-nth*[*OF i*]
proj-poly-syz-lift-poly-syz[*OF i*])
also have ... = *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *p*) *bs*) +
monom-mult c t (*bs* ! *i*) **by** (*simp only*: *module-plus*(*2*))
also have ... = *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) (*p* + *monom-mult c t b*)) *bs*)
proof (*simp add*: *cofactor-list-syz-plus map2-mult-scalar-plus sum-list-map2-plus*)
have *proj-b*: *j* < *length bs* ⟹ *proj-poly j b* = (*1 when j* = *i*) **for** *j*
by (*simp add*: *b init-syzygy-list-nth i proj-lift-poly-syz*)
have *eq*: *j* < *length bs* ⟹ (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) (*monom-mult c t b*)) *bs*) ! *j* =
(*monom-mult c t* (*bs* ! *i*) *when j* = *i*) **for** *j*
by (*simp add*: *cofactor-list-syz-nth proj-monom-mult proj-b mult-scalar-monom-mult when-def*)
have *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) (*monom-mult c t b*)) *bs*) =
(*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) (*monom-mult c t b*)) *bs*) ! *i*
by (*rule sum-list-eq-nthI*, *simp add*: *i*, *simp add*: *eq del*: *nth-zip nth-map*)
also have ... = *mult-scalar* (*punit.monom-mult c t* (*proj-poly i b*)) (*bs* ! *i*)
by (*simp add*: *i cofactor-list-syz-nth proj-monom-mult*)
also have ... = *monom-mult c t* (*bs* ! *i*)
by (*simp add*: *proj-b i mult-scalar-monomial times-monomial-left*[*symmetric*])
finally show *monom-mult c t* (*bs* ! *i*) =
sum-list (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) (*monom-mult c t b*)) *bs*)
by (*simp only*:)
qed
finally show *?case* **.**
qed

18.4.9 Modules

lemma *pmdl-proj-orig-basis*:
  assumes *pmdl* (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
  shows *pmdl* (*set* (*proj-orig-basis* (*length bs*) *gs*)) = *pmdl* (*set bs*)
  by (*simp add*: *pmdl-proj-orig-basis′ assms*,
      simp only: *pmdl-proj-orig-basis′*[*symmetric*] *proj-orig-basis-init-syzygy-list*)

lemma *pmdl-filter-syzygy-basis-subset*:
  assumes *distinct bs* **and** *pmdl* (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
  shows *pmdl* (*set* (*filter-syzygy-basis* (*length bs*) *gs*)) ⊆ *pmdl* (*syzygy-module-list bs*)
proof (*rule pmdl.span-mono*, *rule*)
  fix *s*
  assume *s* ∈ *set* (*filter-syzygy-basis* (*length bs*) *gs*)
  hence *s* ∈ *set gs* **and** *eq*: *proj-poly-syz* (*length bs*) *s* = *0*
    by (*simp-all add*: *set-filter-syzygy-basis*)
  from *this*(*1*) **have** *s* ∈ *pmdl* (*set gs*) **by** (*rule pmdl.span-base*)
  hence *s* ∈ *pmdl* (*set* (*init-syzygy-list bs*)) **by** (*simp only*: *assms*)
  hence *proj-poly-syz* (*length bs*) *s* =
        sum-list (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *s*) *bs*)
    by (*rule syz-cofactors*)
  hence *distinct bs* **and** *sum-list* (*map2 mult-scalar* (*cofactor-list-syz* (*length bs*) *s*) *bs*) = *0*
    by (*simp-all only*: *eq assms*(*1*))
  moreover from *eq* **have** *component-of-term* ‘ *keys s* ⊆ {*0*..<*length bs*} **by** (*simp only*: *proj-poly-syz-eq-zero-iff*)
  ultimately show *s* ∈ *syzygy-module-list bs* **by** (*rule syzygy-module-listI′*)
qed

lemma *ex-filter-syzygy-basis-adds-lt*:
  assumes *is-pot-ord* **and** *distinct bs* **and** *is-Groebner-basis* (*set gs*)
    and *pmdl* (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
    and *f* ∈ *pmdl* (*syzygy-module-list bs*) **and** *f* ≠ *0*
  shows ∃ *g*∈*set* (*filter-syzygy-basis* (*length bs*) *gs*). *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
proof −
  from *assms*(*5*) **have** *f* ∈ *syzygy-module-list bs* **by** *simp*
  also from *assms*(*2*) **have** ... ⊆ *pmdl* (*set* (*init-syzygy-list bs*))
    by (*rule syzygy-module-list-subset*)
  also have ... = *pmdl* (*set gs*) **by** (*simp only*: *assms*(*4*))
  finally have *f* ∈ *pmdl* (*set gs*) **.**
  with *assms*(*3*, *6*) **obtain** *g* **where** *g* ∈ *set gs* **and** *g* ≠ *0*
    and *adds*: *lt g* $adds_t$ *lt f* **unfolding** *GB-alt-3-finite*[*OF finite-set*] **by** *blast*
  show *?thesis*
  proof (*intro bexI conjI*)
    show *g* ∈ *set* (*filter-syzygy-basis* (*length bs*) *gs*)
    proof (*simp add*: *set-filter-syzygy-basis*, *rule*)
      show *proj-poly-syz* (*length bs*) *g* = *0*
      proof (*rule ccontr*)
        assume *proj-poly-syz* (*length bs*) *g* ≠ *0*

with *assms*(*1*) **have** *length bs* ≤ *component-of-term* (*lt g*) **by** (*rule component-of-lt-ge*)
also from *adds* **have** ... = *component-of-term* (*lt f*) **by** (*simp add*: *adds-term-def*)
also have ... < *length bs*
proof −
from ‹*f* ≠ *0*› **have** *lt f* ∈ *keys f* **by** (*rule lt-in-keys*)
hence *component-of-term* (*lt f*) ∈ *component-of-term* ‘ *keys f* **by** (*rule imageI*)
also from ‹*f* ∈ *syzygy-module-list bs*› **have** ... ⊆ {*0*..<*length bs*}
by (*rule component-of-syzygy-module-list*)
finally show *component-of-term* (*lt f*) < *length bs* **by** *simp*
qed
finally show *False* **..**
qed
qed *fact*
qed *fact+*
qed

lemma *pmdl-filter-syzygy-basis*:
fixes *bs*::(*'t* $\Rightarrow_0$ *'b*::*field*) *list*
assumes *is-pot-ord* **and** *distinct bs* **and** *is-Groebner-basis* (*set gs*) **and**
pmdl (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
shows *pmdl* (*set* (*filter-syzygy-basis* (*length bs*) *gs*)) = *syzygy-module-list bs*
proof −
from *finite-set*
have *pmdl* (*set* (*filter-syzygy-basis* (*length bs*) *gs*)) = *pmdl* (*syzygy-module-list bs*)
proof (*rule pmdl-eqI-adds-lt-finite*)
from *assms*(*2*, *4*)
show *pmdl* (*set* (*filter-syzygy-basis* (*length bs*) *gs*)) ⊆ *pmdl* (*syzygy-module-list bs*)
by (*rule pmdl-filter-syzygy-basis-subset*)
next
fix *f*
assume *f* ∈ *pmdl* (*syzygy-module-list bs*) **and** *f* ≠ *0*
with *assms* **show** ∃ *g*∈*set* (*filter-syzygy-basis* (*length bs*) *gs*). *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
by (*rule ex-filter-syzygy-basis-adds-lt*)
qed
thus *?thesis* **by** *simp*
qed

18.4.10 Gröbner Bases

lemma *proj-orig-basis-isGB*:
assumes *is-pot-ord* **and** *is-Groebner-basis* (*set gs*) **and** *pmdl* (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
shows *is-Groebner-basis* (*set* (*proj-orig-basis* (*length bs*) *gs*))

unfolding *GB-alt-3-finite*[*OF finite-set*]
proof (*intro ballI impI*)
fix *f*
assume *f* ∈ *pmdl* (*set* (*proj-orig-basis* (*length bs*) *gs*))
also have ... = *proj-poly-syz* (*length bs*) *‘ pmdl* (*set gs*) **by** (*fact pmdl-proj-orig-basis′*)
finally obtain *h* **where** *h* ∈ *pmdl* (*set gs*) **and** *f*: *f* = *proj-poly-syz* (*length bs*) *h* **..**
assume *f* ≠ *0*
with *assms*(*1*) **have** *ltf*: *lt f* = *map-component* (λ*k*. *k* − *length bs*) (*lt h*) **unfolding** *f*
by (*rule lt-proj-poly-syz*)
from ‹*f* ≠ *0*› **have** *h* ≠ *0* **by** (*auto simp add*: *f*)
with *assms*(*2*) ‹*h* ∈ *pmdl* (*set gs*)› **obtain** *g* **where** *g* ∈ *set gs* **and** *g* ≠ *0*
and *lt g* $adds_t$ *lt h* **unfolding** *GB-alt-3-finite*[*OF finite-set*] **by** *blast*
from *this*(*3*) **have** *1*: *component-of-term* (*lt g*) = *component-of-term* (*lt h*)
and *2*: *pp-of-term* (*lt g*) *adds pp-of-term* (*lt h*) **by** (*simp-all add*: *adds-term-def*)
let *?g* = *proj-poly-syz* (*length bs*) *g*
have *?g* ≠ *0*
proof (*simp add*: *proj-poly-syz-eq-zero-iff*, *rule*)
assume *component-of-term ‘ keys g* ⊆ {*0*..<*length bs*}
from *assms*(*1*) ‹*f* ≠ *0*› **have** *length bs* ≤ *component-of-term* (*lt h*)
unfolding *f* **by** (*rule component-of-lt-ge*)
hence *component-of-term* (*lt g*) ∉ {*0*..<*length bs*} **by** (*simp add*: *1*)
moreover from ‹*g* ≠ *0*› **have** *lt g* ∈ *keys g* **by** (*rule lt-in-keys*)
ultimately show *False* **using** ‹*component-of-term ‘ keys g* ⊆ {*0*..<*length bs*}› **by** *blast*
qed
with *assms*(*1*) **have** *ltg*: *lt ?g* = *map-component* (λ*k*. *k* − *length bs*) (*lt g*) **by** (*rule lt-proj-poly-syz*)
show ∃ *g*∈*set* (*proj-orig-basis* (*length bs*) *gs*). *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
proof (*intro bexI conjI*)
show *lt ?g* $adds_t$ *lt f* **by** (*simp add*: *ltf ltg adds-term-def 1 2 term-simps*)
next
show *?g* ∈ *set* (*proj-orig-basis* (*length bs*) *gs*)
unfolding *set-proj-orig-basis* **using** ‹*g* ∈ *set gs*› **by** (*rule imageI*)
qed *fact*
qed

lemma *filter-syzygy-basis-isGB*:
assumes *is-pot-ord* **and** *distinct bs* **and** *is-Groebner-basis* (*set gs*)
and *pmdl* (*set gs*) = *pmdl* (*set* (*init-syzygy-list bs*))
shows *is-Groebner-basis* (*set* (*filter-syzygy-basis* (*length bs*) *gs*))
unfolding *GB-alt-3-finite*[*OF finite-set*]
proof (*intro ballI impI*)
fix *f*::′*t* ⇒$_0$ ′*b*
assume *f* ≠ *0*
assume *f* ∈ *pmdl* (*set* (*filter-syzygy-basis* (*length bs*) *gs*))
also from *assms* **have** ... = *syzygy-module-list bs* **by** (*rule pmdl-filter-syzygy-basis*)
finally have *f* ∈ *pmdl* (*syzygy-module-list bs*) **by** *simp*

from *assms this* ‹*f* ≠ *0*›
show ∃ *g*∈*set* (*filter-syzygy-basis* (*length bs*) *gs*). *g* ≠ *0* ∧ *lt g* $adds_t$ *lt f*
by (*rule ex-filter-syzygy-basis-adds-lt*)
qed

end

end

19 Sample Computations of Syzygies

theory *Syzygy-Examples*
imports *Buchberger Algorithm-Schema-Impl Syzygy Code-Target-Rat*
begin

19.1 Preparations

We must define the following four constants outside the global interpretation, since otherwise their types are too general.

definition *splus-pprod* :: (*'a*::*nat*, *'b*::*nat*) *pp* ⇒ -
where *splus-pprod* = *pprod.splus*

definition *monom-mult-pprod* :: *'c*::*semiring-0* ⇒ (*'a*::*nat*, *'b*::*nat*) *pp* ⇒ (((*'a*, *'b*) *pp* × *nat*) $\Rightarrow_0$ *'c*) ⇒ -
where *monom-mult-pprod* = *pprod.monom-mult*

definition *mult-scalar-pprod* :: ((*'a*::*nat*, *'b*::*nat*) *pp* $\Rightarrow_0$ *'c*::*semiring-0*) ⇒ (((*'a*, *'b*) *pp* × *nat*) $\Rightarrow_0$ *'c*) ⇒ -
where *mult-scalar-pprod* = *pprod.mult-scalar*

definition *adds-term-pprod* :: ((*'a*::*nat*, *'b*::*nat*) *pp* × -) ⇒ -
where *adds-term-pprod* = *pprod.adds-term*

lemma (**in** *gd-term*) *compute-trd-aux* [*code*]:
trd-aux fs p r =
(*if is-zero p then*
r
else
case find-adds fs (*lt p*) *of*
None ⇒ *trd-aux fs* (*tail p*) (*plus-monomial-less r* (*lc p*) (*lt p*))
| *Some f* ⇒ *trd-aux fs* (*tail p* − *monom-mult* (*lc p* / *lc f*) (*lp p* − *lp f*) (*tail f*)) *r*
)
by (*simp only*: *trd-aux.simps*[*of fs p r*] *plus-monomial-less-def is-zero-def*)

locale *gd-nat-inf-term* = *gd-nat-term pair-of-term term-of-pair cmp-term*
for *pair-of-term*::*'t*::*nat-term* ⇒ (*'a*::{*nat-term*,*graded-dickson-powerprod*} × *nat*)

and *term-of-pair*::('*a* × *nat*) ⇒ '*t*
and *cmp-term*
begin

sublocale *aux*: *gd-inf-term pair-of-term term-of-pair*
λ*s t*. *le-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))
λ*s t*. *lt-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))
le-of-nat-term-order cmp-term
lt-of-nat-term-order cmp-term **..**

definition *lift-keys* :: *nat* ⇒ ('*t*, '*b*) *oalist-ntm* ⇒ ('*t*, '*b*::*semiring-0*) *oalist-ntm*
where *lift-keys i xs* = *oalist-of-list-ntm* (*map-raw* (λ*kv*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)) (*list-of-oalist-ntm xs*))

lemma *list-of-oalist-lift-keys*:
list-of-oalist-ntm (*lift-keys i xs*) = (*map-raw* (λ*kv*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)) (*list-of-oalist-ntm xs*))
unfolding *lift-keys-def* **oops**

Regardless of whether the above lemma holds (which might be the case) or not, we can use *lift-keys* in computations. Now, however, it is implemented rather inefficiently, because the list resulting from the application of *map-raw* is sorted again. That should not be a big problem though, since *lift-keys* is applied only once to every input polynomial before computing syzygies.

lemma *lookup-lift-keys-plus*:
lookup (*MP-oalist* (*lift-keys i xs*)) (*term-of-pair* (*t*, *i* + *k*)) = *lookup* (*MP-oalist xs*) (*term-of-pair* (*t*, *k*))
(**is** *?l* = *?r*)
proof −
let *?f* = λ*kv*::'*t* × '*b*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)
obtain *xs*′ *ox* **where** *xs*: *list-of-oalist-ntm xs* = (*xs*′, *ox*) **by** *fastforce*
from *oalist-inv-list-of-oalist-ntm*[*of xs*] **have** *inv*: *ko-ntm.oalist-inv-raw ox xs*′
by (*simp add*: *xs ko-ntm.oalist-inv-def nat-term-compare-inv-conv*)
let *?rel* = *ko.lt* (*key-order-of-nat-term-order-inv ox*)
have *irreflp ?rel* **by** (*simp add*: *irreflp-def*)
moreover have *transp ?rel* **by** (*simp add*: *lt-of-nat-term-order-alt*)
moreover from *oa-ntm.list-of-oalist-sorted*[*of xs*]
have *sorted-wrt* (*ko.lt* (*key-order-of-nat-term-order-inv ox*)) (*map fst xs*′) **by** (*simp add*: *xs*)
ultimately have *dist1*: *distinct* (*map fst xs*′) **by** (*rule distinct-sorted-wrt-irrefl*)
have *1*: *u* = *v* **if** *map-component* ((+) *i*) *u* = *map-component* ((+) *i*) *v* **for** *u v*
proof −
have *inj* ((+) *i*) **by** (*simp add*: *inj-def*)
thus *?thesis* **using** *that* **by** (*rule map-component-inj*)
qed
have *dist2*: *distinct* (*map fst* (*map-pair* (λ*kv*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)) *xs*′))

by (*rule ko-ntm.distinct-map-pair*, *fact dist1*, *simp add*: *1*)
have *?l* = *lookup-dflt* (*map-pair ?f xs'*) (*term-of-pair* (*t*, *i* + *k*))
by (*simp add*: *oa-ntm.lookup-def lift-keys-def xs oalist-of-list-ntm-def list-of-oalist-OAlist-ntm ko-ntm.lookup-pair-sort-oalist'*[*OF dist2*])
also have ... = *lookup-dflt* (*map-pair ?f xs'*) (*fst* (*?f* (*term-of-pair* (*t*, *k*), *b*)))
by (*simp add*: *map-component-term-of-pair*)
also have ... = *snd* (*?f* (*term-of-pair* (*t*, *k*), *lookup-dflt xs'* (*term-of-pair* (*t*, *k*))))
by (*rule ko-ntm.lookup-dflt-map-pair*, *fact dist1*, *auto intro*: *1*)
also have ... = *?r* **by** (*simp add*: *oa-ntm.lookup-def xs ko-ntm.lookup-dflt-eq-lookup-pair*[*OF inv*])
finally show *?thesis* **.**
qed

lemma *keys-lift-keys-subset*:
keys (*MP-oalist* (*lift-keys i xs*)) ⊆ (*map-component* ((+) *i*)) ' *keys* (*MP-oalist xs*)
(**is** *?l* ⊆ *?r*)
proof −
let *?f* = *λkv*::*'t* × *'b*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)
obtain *xs' ox* **where** *xs*: *list-of-oalist-ntm xs* = (*xs'*, *ox*) **by** *fastforce*
let *?rel* = *ko.lt* (*key-order-of-nat-term-order-inv ox*)
have *irreflp ?rel* **by** (*simp add*: *irreflp-def*)
moreover have *transp ?rel* **by** (*simp add*: *lt-of-nat-term-order-alt*)
moreover from *oa-ntm.list-of-oalist-sorted*[*of xs*]
have *sorted-wrt* (*ko.lt* (*key-order-of-nat-term-order-inv ox*)) (*map fst xs'*) **by** (*simp add*: *xs*)
ultimately have *dist1*: *distinct* (*map fst xs'*) **by** (*rule distinct-sorted-wrt-irrefl*)
have *1*: *u* = *v* **if** *map-component* ((+) *i*) *u* = *map-component* ((+) *i*) *v* **for** *u v*
proof −
have *inj* ((+) *i*) **by** (*simp add*: *inj-def*)
thus *?thesis* **using** *that* **by** (*rule map-component-inj*)
qed
have *dist2*: *distinct* (*map fst* (*map-pair* (*λkv*. (*map-component* ((+) *i*) (*fst kv*), *snd kv*)) *xs'*))
by (*rule ko-ntm.distinct-map-pair*, *fact dist1*, *simp add*: *1*)
have *?l* ⊆ *fst* ' *set* (*fst* (*map-raw ?f* (*list-of-oalist-ntm xs*)))
by (*auto simp*: *keys-MP-oalist lift-keys-def oalist-of-list-ntm-def list-of-oalist-OAlist-ntm xs ko-ntm.set-sort-oalist*[*OF dist2*])
also from *ko-ntm.map-raw-subset* **have** ... ⊆ *fst* ' *?f* ' *set* (*fst* (*list-of-oalist-ntm xs*))
by (*rule image-mono*)
also have ... ⊆ *?r* **by** (*simp add*: *keys-MP-oalist image-image*)
finally show *?thesis* **.**
qed

end

global-interpretation *pprod'*: *gd-nat-inf-term λx*::(*'a*, *'b*) *pp* × *nat*. *x λx*. *x cmp-term*
rewrites *pprod.pp-of-term* = *fst*

and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.monom-mult* = *monom-mult-pprod*
and *pprod.mult-scalar* = *mult-scalar-pprod*
and *pprod.adds-term* = *adds-term-pprod*
for *cmp-term* :: (('*a*::*nat*, '*b*::*nat*) *pp* × *nat*) *nat-term-order*
defines *shift-map-keys-pprod* = *pprod'.shift-map-keys*
and *lift-keys-pprod* = *pprod'.lift-keys*
and *min-term-pprod* = *pprod'.min-term*
and *lt-pprod* = *pprod'.lt*
and *lc-pprod* = *pprod'.lc*
and *tail-pprod* = *pprod'.tail*
and *comp-opt-p-pprod* = *pprod'.comp-opt-p*
and *ord-p-pprod* = *pprod'.ord-p*
and *ord-strict-p-pprod* = *pprod'.ord-strict-p*
and *find-adds-pprod* = *pprod'.find-adds*
and *trd-aux-pprod*= *pprod'.trd-aux*
and *trd-pprod* = *pprod'.trd*
and *spoly-pprod* = *pprod'.spoly*
and *count-const-lt-components-pprod* = *pprod'.count-const-lt-components*
and *count-rem-components-pprod* = *pprod'.count-rem-components*
and *const-lt-component-pprod* = *pprod'.const-lt-component*
and *full-gb-pprod* = *pprod'.full-gb*
and *keys-to-list-pprod* = *pprod'.keys-to-list*
and *Keys-to-list-pprod* = *pprod'.Keys-to-list*
and *add-pairs-single-sorted-pprod* = *pprod'.add-pairs-single-sorted*
and *add-pairs-pprod* = *pprod'.add-pairs*
and *canon-pair-order-aux-pprod* = *pprod'.canon-pair-order-aux*
and *canon-basis-order-pprod* = *pprod'.canon-basis-order*
and *new-pairs-sorted-pprod* = *pprod'.new-pairs-sorted*
and *component-crit-pprod* = *pprod'.component-crit*
and *chain-ncrit-pprod* = *pprod'.chain-ncrit*
and *chain-ocrit-pprod* = *pprod'.chain-ocrit*
and *apply-icrit-pprod* = *pprod'.apply-icrit*
and *apply-ncrit-pprod* = *pprod'.apply-ncrit*
and *apply-ocrit-pprod* = *pprod'.apply-ocrit*
and *trdsp-pprod* = *pprod'.trdsp*
and *gb-sel-pprod* = *pprod'.gb-sel*
and *gb-red-aux-pprod* = *pprod'.gb-red-aux*
and *gb-red-pprod* = *pprod'.gb-red*
and *gb-aux-pprod* = *pprod'.gb-aux*
and *gb-pprod* = *pprod'.gb*
and *filter-syzygy-basis-pprod* = *pprod'.aux.filter-syzygy-basis*
and *init-syzygy-list-pprod* = *pprod'.aux.init-syzygy-list*
and *lift-poly-syz-pprod* = *pprod'.aux.lift-poly-syz*
and *map-component-pprod* = *pprod'.map-component*
subgoal by (*rule gd-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*: *monom-mult-pprod-def*)
subgoal by (*simp only*: *mult-scalar-pprod-def*)
subgoal by (*simp only*: *adds-term-pprod-def*)
done

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-shift-map-keys-pprod* [*code abstract*]:
list-of-oalist-ntm (*shift-map-keys-pprod t f xs*) = *map-raw* (λ(*k*, *v*). (*splus-pprod t k*, *f v*)) (*list-of-oalist-ntm xs*)
by (*simp add*: *pprod′.list-of-oalist-shift-keys case-prod-beta′*)

lemma *compute-trd-pprod* [*code*]: *trd-pprod to fs p* = *trd-aux-pprod to fs p* (*change-ord to 0*)
by (*simp only*: *pprod′.trd-def change-ord-def*)

lemmas [*code*] = *conversep-iff*

lemma *POT-is-pot-ord*: *pprod′.is-pot-ord* (*TYPE*(*′a*::*nat*)) (*TYPE*(*′b*::*nat*)) (*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*)

definition Vec_0 :: *nat* ⇒ ((*′a*, *nat*) *pp* $\Rightarrow_0$ *′b*) ⇒ ((*′a*::*nat*, *nat*) *pp* × *nat*) $\Rightarrow_0$ *′b*::*semiring-1* **where**
Vec_0 *i p* = *mult-scalar-pprod p* (*Poly-Mapping.single* (*0*, *i*) *1*)

definition *syzygy-basis to bs* =
filter-syzygy-basis-pprod (*length bs*) (*map fst* (*gb-pprod* (*POT to*) (*map* (λ*p*. (*p*, ())) (*init-syzygy-list-pprod bs*)) ()))

thm *pprod′.aux.filter-syzygy-basis-isGB*[*OF POT-is-pot-ord*]

lemma *lift-poly-syz-MP-oalist* [*code*]:
lift-poly-syz-pprod n (*MP-oalist xs*) *i* = *MP-oalist* (*OAlist-insert-ntm* ((*0*, *i*), *1*) (*lift-keys-pprod n xs*))
proof (*rule poly-mapping-eqI*, *simp add*: *pprod′.aux.lookup-lift-poly-syz del*: *MP-oalist.rep-eq*, *intro conjI impI*)
fix *v*::(*′a*, *′b*) *pp* × *nat*
assume *n* ≤ *snd v*
moreover obtain *t k* **where** *v* = (*t*, *k*) **by** *fastforce*
ultimately have *k*: *n* + (*k* − *n*) = *k* **by** *simp*

hence *v*: *v* = (*t*, *n* + (*k* − *n*)) **by** (*simp only*: ‹*v* = (*t*, *k*)›)
assume *v* ≠ (*0*, *i*)
hence *lookup* (*MP-oalist* (*OAlist-insert-ntm* ((*0*, *i*), *1*) (*lift-keys-pprod n xs*))) *v* =
lookup (*MP-oalist* (*lift-keys-pprod n xs*)) *v* **by** (*simp add*: *oa-ntm.lookup-insert*)
also have ... = *lookup* (*MP-oalist xs*) (*t*, *k* − *n*) **by** (*simp only*: *v pprod'.lookup-lift-keys-plus*)
also have ... = *lookup* (*MP-oalist xs*) (*map-component-pprod* (λ*k*. *k* − *n*) *v*)
by (*simp add*: *v pprod'.map-component-term-of-pair*)
finally show *lookup* (*MP-oalist xs*) (*map-component-pprod* (λ*k*. *k* − *n*) *v*) =
lookup (*MP-oalist* (*OAlist-insert-ntm* ((*0*, *i*), *1*) (*lift-keys-pprod n xs*))) *v* **by** (*rule HOL.sym*)
next
fix *v*::('*a*, '*b*) *pp* × *nat*
assume ¬ *n* ≤ *snd v*
assume *v* ≠ (*0*, *i*)
hence *lookup* (*MP-oalist* (*OAlist-insert-ntm* ((*0*, *i*), *1*) (*lift-keys-pprod n xs*))) *v* =
lookup (*MP-oalist* (*lift-keys-pprod n xs*)) *v* **by** (*simp add*: *add*: *oa-ntm.lookup-insert*)
also have ... = *0*
proof (*rule ccontr*)
assume *lookup* (*MP-oalist* (*lift-keys-pprod n xs*)) *v* ≠ *0*
hence *v* ∈ *keys* (*MP-oalist* (*lift-keys-pprod n xs*)) **by** (*simp add*: *in-keys-iff del*: *MP-oalist.rep-eq*)
also have ... ⊆ *map-component-pprod* ((+) *n*) ‘ *keys* (*MP-oalist xs*)
by (*fact pprod'.keys-lift-keys-subset*)
finally obtain *u* **where** *v* = *map-component-pprod* ((+) *n*) *u* **..**
hence *snd v* = *n* + *snd u* **by** (*simp add*: *pprod'.component-of-map-component*)
with ‹¬ *n* ≤ *snd v*› **show** *False* **by** *simp*
qed
finally show *lookup* (*MP-oalist* (*OAlist-insert-ntm* ((*0*, *i*), *1*) (*lift-keys-pprod n xs*))) *v* = *0* **.**
qed (*simp-all add*: *oa-ntm.lookup-insert*)

19.2 Computations

experiment begin interpretation *trivariate*$_0$*-rat* **.**

lemma
syzygy-basis DRLEX [*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*), *Vec*$_0$ *0* (*X* ∗ *Y* ∗ *Z* + *2* ∗ Y^2)] =
[*Vec*$_0$ *0* (C_0 (*1* / *3*) ∗ *X* ∗ *Y* ∗ *Z* + C_0 (*2* / *3*) ∗ Y^2) + *Vec*$_0$ *1* (C_0 (− *1* / *3*) ∗ X^2 ∗ *Z* ^ *3* − X^2 ∗ *Y*)]
by *eval*

value [*code*] *syzygy-basis DRLEX* [*Vec*$_0$ *0* (X^2 ∗ *Z* ^ *3* + *3* ∗ X^2 ∗ *Y*), *Vec*$_0$ *0* (*X* ∗ *Y* ∗ *Z* + *2* ∗ Y^2), *Vec*$_0$ *0* (*X* − *Y* + *3* ∗ *Z*)]

lemma
map fst (*gb-pprod* (*POT DRLEX*) (*map* (λ*p*. (*p*, ()))) (*init-syzygy-list-pprod*

[*Vec*$_0$ *0* (X ^ *4* + *3* ∗ X^2 ∗ Y), *Vec*$_0$ *0* (Y ^ *3* + *2* ∗ X ∗ Z), *Vec*$_0$ *0* (Z^2 − X − Y)])) ()) =
[
Vec$_0$ *0 1* + *Vec*$_0$ *3* (X ^ *4* + *3* ∗ X^2 ∗ Y),
Vec$_0$ *1 1* + *Vec*$_0$ *3* (Y ^ *3* + *2* ∗ X ∗ Z),
Vec$_0$ *0* (Y ^ *3* + *2* ∗ X ∗ Z) − *Vec*$_0$ *1* (X ^ *4* + *3* ∗ X^2 ∗ Y),
Vec$_0$ *2 1* + *Vec*$_0$ *3* (Z^2 − X − Y),
Vec$_0$ *1* (Z^2 − X − Y) − *Vec*$_0$ *2* (Y ^ *3* + *2* ∗ X ∗ Z),
Vec$_0$ *0* (Z^2 − X − Y) − *Vec*$_0$ *2* (X ^ *4* + *3* ∗ X^2 ∗ Y),
Vec$_0$ *0* (− (Y ^ *3* ∗ Z^2) + Y ^ *4* + X ∗ Y ^ *3* + *2* ∗ X^2 ∗ Z + *2* ∗ X ∗ Y ∗ Z − *2* ∗ X ∗ Z ^ *3*) +
Vec$_0$ *1* (X ^ *4* ∗ Z^2 − X ^ *5* − X ^ *4* ∗ Y − *3* ∗ X ^ *3* ∗ Y − *3* ∗ X^2 ∗ Y^2 + *3* ∗ X^2 ∗ Y ∗ Z^2)
]
by *eval*

lemma
syzygy-basis DRLEX [*Vec*$_0$ *0* (X ^ *4* + *3* ∗ X^2 ∗ Y), *Vec*$_0$ *0* (Y ^ *3* + *2* ∗ X ∗ Z), *Vec*$_0$ *0* (Z^2 − X − Y)] =
[
Vec$_0$ *0* (Y ^ *3* + *2* ∗ X ∗ Z) − *Vec*$_0$ *1* (X ^ *4* + *3* ∗ X^2 ∗ Y),
Vec$_0$ *1* (Z^2 − X − Y) − *Vec*$_0$ *2* (Y ^ *3* + *2* ∗ X ∗ Z),
Vec$_0$ *0* (Z^2 − X − Y) − *Vec*$_0$ *2* (X ^ *4* + *3* ∗ X^2 ∗ Y),
Vec$_0$ *0* (− (Y ^ *3* ∗ Z^2) + Y ^ *4* + X ∗ Y ^ *3* + *2* ∗ X^2 ∗ Z + *2* ∗ X ∗ Y ∗ Z − *2* ∗ X ∗ Z ^ *3*) +
Vec$_0$ *1* (X ^ *4* ∗ Z^2 − X ^ *5* − X ^ *4* ∗ Y − *3* ∗ X ^ *3* ∗ Y − *3* ∗ X^2 ∗ Y^2 + *3* ∗ X^2 ∗ Y ∗ Z^2)
]
by *eval*

value [*code*] *syzygy-basis DRLEX* [*Vec*$_0$ *0* (X ∗ Y − Z), *Vec*$_0$ *0* (X ∗ Z − Y), *Vec*$_0$ *0* (Y ∗ Z − X)]

lemma
map fst (*gb-pprod* (*POT DRLEX*) (*map* (λ*p*. (*p*, ())) (*init-syzygy-list-pprod*
[*Vec*$_0$ *0* (X ∗ Y − Z), *Vec*$_0$ *0* (X ∗ Z − Y), *Vec*$_0$ *0* (Y ∗ Z − X)])) ()) =
[
Vec$_0$ *0 1* + *Vec*$_0$ *3* (X ∗ Y − Z),
Vec$_0$ *1 1* + *Vec*$_0$ *3* (X ∗ Z − Y),
Vec$_0$ *2 1* + *Vec*$_0$ *3* (Y ∗ Z − X),
Vec$_0$ *0* (− X ∗ Z + Y) + *Vec*$_0$ *1* (X ∗ Y − Z),
Vec$_0$ *0* (− Y ∗ Z + X) + *Vec*$_0$ *2* (X ∗ Y − Z),
Vec$_0$ *1* (− Y ∗ Z + X) + *Vec*$_0$ *2* (X ∗ Z − Y),
Vec$_0$ *1* (−Y) + *Vec*$_0$ *2* (X) + *Vec*$_0$ *3* (Y ^ *2* − X ^ *2*),
Vec$_0$ *0* (Z) + *Vec*$_0$ *2* (−X) + *Vec*$_0$ *3* (X ^ *2* − Z ^ *2*),
Vec$_0$ *0* (Y − Y ∗ Z ^ *2*) + *Vec*$_0$ *1* (Y ^ *2* ∗ Z − Z) + *Vec*$_0$ *2* (Y ^ *2* − Z ^ *2*),
Vec$_0$ *0* (− Y) + *Vec*$_0$ *1* (− (X ∗ Y)) + *Vec*$_0$ *2* (X ^ *2* − *1*) + *Vec*$_0$ *3* (X − X ^ *3*)

]
by *eval*

lemma
syzygy-basis DRLEX [Vec_0 *0* ($X * Y - Z$), Vec_0 *0* ($X * Z - Y$), Vec_0 *0* ($Y * Z - X$)] =
[
Vec_0 *0* ($- X * Z + Y$) + Vec_0 *1* ($X * Y - Z$),
Vec_0 *0* ($- Y * Z + X$) + Vec_0 *2* ($X * Y - Z$),
Vec_0 *1* ($- Y * Z + X$) + Vec_0 *2* ($X * Z - Y$),
Vec_0 *0* ($Y - Y * Z$ ^ *2*) + Vec_0 *1* (Y ^ *2* $* Z - Z$) + Vec_0 *2* (Y ^ *2* $- Z$ ^ *2*)
]
by *eval*

end

end

theory *Groebner-PM*
imports *Polynomials.MPoly-PM Reduced-GB*
begin

We prove results that hold specifically for Gröbner bases in polynomial rings, where the polynomials really have *indeterminates*.

context *pm-powerprod*
begin

lemmas *finite-reduced-GB-Polys* =
punit.finite-reduced-GB-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-is-reduced-GB-Polys* =
punit.reduced-GB-is-reduced-GB-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-is-GB-Polys* =
punit.reduced-GB-is-GB-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-is-auto-reduced-Polys* =
punit.reduced-GB-is-auto-reduced-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-is-monic-set-Polys* =
punit.reduced-GB-is-monic-set-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-nonzero-Polys* =
punit.reduced-GB-nonzero-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-ideal-Polys* =
punit.reduced-GB-pmdl-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where**

m=0, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-unique-Polys* =
punit.reduced-GB-unique-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *reduced-GB-Polys* =
punit.reduced-GB-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]
lemmas *ideal-eq-UNIV-iff-reduced-GB-eq-one-Polys* =
ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set[*simplified*, *OF dickson-grading-varnum*, **where** *m=0*, *simplified dgrad-p-set-varnum*]

19.3 Univariate Polynomials

lemma (**in** −) *adds-univariate-linear*:
assumes *finite X* **and** *card X* ≤ *1* **and** *s* ∈ .[*X*] **and** *t* ∈ .[*X*]
obtains *s adds t* | *t adds s*
proof (*cases s adds t*)
case *True*
thus *?thesis* **..**
next
case *False*
then obtain *x* **where** *1*: *lookup t x* < *lookup s x* **by** (*auto simp*: *adds-poly-mapping le-fun-def not-le*)
hence *x* ∈ *keys s* **by** (*simp add*: *in-keys-iff*)
also from *assms*(*3*) **have** ... ⊆ *X* **by** (*rule PPsD*)
finally have *x* ∈ *X* **.**
have *t adds s* **unfolding** *adds-poly-mapping le-fun-def*
proof
fix *y*
show *lookup t y* ≤ *lookup s y*
proof (*cases y* ∈ *keys t*)
case *True*
also from *assms*(*4*) **have** *keys t* ⊆ *X* **by** (*rule PPsD*)
finally have *y* ∈ *X* **.**
with *assms*(*1*, *2*) ‹*x* ∈ *X*› **have** *x* = *y* **by** (*simp add*: *card-le-Suc0-iff-eq*)
with *1* **show** *?thesis* **by** *simp*
next
case *False*
thus *?thesis* **by** (*simp add*: *in-keys-iff*)
qed
qed
thus *?thesis* **..**
qed

context
fixes *X* :: *'x set*
assumes *fin-X*: *finite X* **and** *card-X*: *card X* ≤ *1*
begin

lemma *ord-iff-adds-univariate*:
  assumes $s \in .[X]$ **and** $t \in .[X]$
  shows $s \preceq t \longleftrightarrow s$ *adds* t
proof
  assume $s \preceq t$
  from *fin-X card-X assms* **show** s *adds* t
  proof (*rule adds-univariate-linear*)
    assume t *adds* s
    hence $t \preceq s$ **by** (*rule ord-adds*)
    with ‹$s \preceq t$› **have** $s = t$
      by *simp*
    thus *?thesis* **by** *simp*
  qed
qed (*rule ord-adds*)

lemma *adds-iff-deg-le-univariate*:
  assumes $s \in .[X]$ **and** $t \in .[X]$
  shows s *adds* $t \longleftrightarrow$ *deg-pm* $s \le$ *deg-pm* t
proof
  assume $*$: *deg-pm* $s \le$ *deg-pm* t
  from *fin-X card-X assms* **show** s *adds* t
  proof (*rule adds-univariate-linear*)
    assume t *adds* s
    hence $t = s$ **using** $*$ **by** (*rule adds-deg-pm-antisym*)
    thus *?thesis* **by** *simp*
  qed
qed (*rule deg-pm-mono*)

corollary *ord-iff-deg-le-univariate*: $s \in .[X] \Longrightarrow t \in .[X] \Longrightarrow s \preceq t \longleftrightarrow$ *deg-pm* s $\le$ *deg-pm* t
  by (*simp only*: *ord-iff-adds-univariate adds-iff-deg-le-univariate*)

lemma *poly-deg-univariate*:
  assumes $p \in P[X]$
  shows *poly-deg* p = *deg-pm* (*lpp* p)
proof (*cases* $p = 0$)
  case *True*
  thus *?thesis* **by** *simp*
next
  case *False*
  hence *lp-in*: *lpp* $p \in$ *keys* p **by** (*rule punit.lt-in-keys*)
  also from *assms* **have** $\ldots \subseteq .[X]$ **by** (*rule PolysD*)
  finally have *lpp* $p \in .[X]$ **.**
  show *?thesis*
  proof (*intro antisym poly-deg-leI*)
    fix t
    assume $t \in$ *keys* p
    hence $t \preceq$ *lpp* p **by** (*rule punit.lt-max-keys*)
    moreover from ‹$t \in$ *keys* p› ‹*keys* $p \subseteq .[X]$› **have** $t \in .[X]$ **..**

ultimately show *deg-pm t* ≤ *deg-pm* (*lpp p*) **using** ‹*lpp p* ∈ .[*X*]›
by (*simp only*: *ord-iff-deg-le-univariate*)
next
from *lp-in* **show** *deg-pm* (*lpp p*) ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
qed
qed

lemma *reduced-GB-univariate-cases*:
assumes *F* ⊆ *P*[*X*]
obtains *g* **where** *g* ∈ *P*[*X*] **and** *g* ≠ *0* **and** *lcf g* = *1* **and** *punit.reduced-GB F* = {*g*} |
punit.reduced-GB F = {}
proof (*cases punit.reduced-GB F* = {})
case *True*
thus *?thesis* **..**
next
case *False*
let *?G* = *punit.reduced-GB F*
from *fin-X assms* **have** *ar*: *punit.is-auto-reduced ?G* **and** *0* ∉ *?G* **and** *?G* ⊆ *P*[*X*]
and *m*: *punit.is-monic-set ?G*
by (*rule reduced-GB-is-auto-reduced-Polys*, *rule reduced-GB-nonzero-Polys*, *rule reduced-GB-Polys*,
rule reduced-GB-is-monic-set-Polys)
from *False* **obtain** *g* **where** *g* ∈ *?G* **by** *blast*
with ‹*0* ∉ *?G*› ‹*?G* ⊆ *P*[*X*]› **have** *g* ≠ *0* **and** *g* ∈ *P*[*X*] **by** *blast+*
from *this*(*1*) **have** *lp-g*: *lpp g* ∈ *keys g* **by** (*rule punit.lt-in-keys*)
also from ‹*g* ∈ *P*[*X*]› **have** ... ⊆ .[*X*] **by** (*rule PolysD*)
finally have *lpp g* ∈ .[*X*] **.**
note ‹*g* ∈ *P*[*X*]› ‹*g* ≠ *0*›
moreover from *m* ‹*g* ∈ *?G*› ‹*g* ≠ *0*› **have** *lcf g* = *1* **by** (*rule punit.is-monic-setD*)
moreover have *?G* = {*g*}
proof
show *?G* ⊆ {*g*}
proof
fix *g'*
assume *g'* ∈ *?G*
with ‹*0* ∉ *?G*› ‹*?G* ⊆ *P*[*X*]› **have** *g'* ≠ *0* **and** *g'* ∈ *P*[*X*] **by** *blast+*
from *this*(*1*) **have** *lp-g'*: *lpp g'* ∈ *keys g'* **by** (*rule punit.lt-in-keys*)
also from ‹*g'* ∈ *P*[*X*]› **have** ... ⊆ .[*X*] **by** (*rule PolysD*)
finally have *lpp g'* ∈ .[*X*] **.**
have *g'* = *g*
proof (*rule ccontr*)
assume *g'* ≠ *g*
with ‹*g* ∈ *?G*› ‹*g'* ∈ *?G*› **have** *g*: *g* ∈ *?G* − {*g'*} **and** *g'*: *g'* ∈ *?G* − {*g*} **by** *blast+*
from *fin-X card-X* ‹*lpp g* ∈ .[*X*]› ‹*lpp g'* ∈ .[*X*]› **show** *False*
proof (*rule adds-univariate-linear*)
assume ∗: *lpp g adds lpp g'*

from *ar* ‹$g' \in$ *?G*› **have** ¬ *punit.is-red* (*?G* − {g'}) g' **by** (*rule punit.is-auto-reducedD*)
moreover from *g* ‹$g \neq 0$› *lp-g'* ∗ **have** *punit.is-red* (*?G* − {g'}) g'
by (*rule punit.is-red-addsI*[*simplified*])
ultimately show *?thesis* **..**
next
assume ∗: *lpp* g' *adds lpp g*
from *ar* ‹$g \in$ *?G*› **have** ¬ *punit.is-red* (*?G* − {*g*}) *g* **by** (*rule punit.is-auto-reducedD*)
moreover from g' ‹$g' \neq 0$› *lp-g* ∗ **have** *punit.is-red* (*?G* − {*g*}) *g*
by (*rule punit.is-red-addsI*[*simplified*])
ultimately show *?thesis* **..**
qed
qed
thus $g' \in \{g\}$ **by** *simp*
qed
next
from ‹$g \in$ *?G*› **show** {*g*} ⊆ *?G* **by** *simp*
qed
ultimately show *?thesis* **..**
qed

corollary *deg-reduced-GB-univariate-le*:
assumes $F \subseteq P[X]$ **and** $f \in$ *ideal F* **and** $f \neq 0$ **and** $g \in$ *punit.reduced-GB F*
shows *poly-deg g* ≤ *poly-deg f*
using *assms*(*1*)
proof (*rule reduced-GB-univariate-cases*)
let *?G* = *punit.reduced-GB F*
fix g'
assume $g' \in P[X]$ **and** $g' \neq 0$ **and** *G*: *?G* = {g'}
from *fin-X assms*(*1*) **have** *gb*: *punit.is-Groebner-basis ?G* **and** *ideal ?G* = *ideal F*
and *?G* ⊆ $P[X]$
by (*rule reduced-GB-is-GB-Polys*, *rule reduced-GB-ideal-Polys*, *rule reduced-GB-Polys*)
from *assms*(*2*) *this*(*2*) **have** $f \in$ *ideal ?G* **by** *simp*
with *gb* **obtain** g'' **where** $g'' \in$ *?G* **and** *lpp* g'' *adds lpp f*
using *assms*(*3*) **by** (*rule punit.GB-adds-lt*[*simplified*])
with *assms*(*4*) **have** *lpp g adds lpp f* **by** (*simp add*: *G*)
hence *deg-pm* (*lpp g*) ≤ *deg-pm* (*lpp f*) **by** (*rule deg-pm-mono*)
moreover from *assms*(*4*) ‹*?G* ⊆ $P[X]$› **have** $g \in P[X]$ **..**
ultimately have *poly-deg g* ≤ *deg-pm* (*lpp f*) **by** (*simp only*: *poly-deg-univariate*)
also from *punit.lt-in-keys* **have** ... ≤ *poly-deg f* **by** (*rule poly-deg-max-keys*) *fact*
finally show *?thesis* **.**
next
assume *punit.reduced-GB F* = {}
with *assms*(*4*) **show** *?thesis* **by** *simp*
qed

end

19.4 Homogeneity

lemma *is-reduced-GB-homogeneous*:
assumes $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** *punit.is-reduced-GB G* **and** *ideal G = ideal F*
and $g \in G$
shows *homogeneous g*
proof (*rule homogeneousI*)
fix *s t*
have *1*: *deg-pm u = deg-pm* (*lpp g*) **if** $u \in$ *keys g* **for** *u*
proof −
from *assms*(*4*) **have** $g \in$ *ideal G* **by** (*rule ideal.span-base*)
hence $g \in$ *ideal F* **by** (*simp only*: *assms*(*3*))
from *that* **have** $u \in$ *Keys* (*hom-components g*) **by** (*simp only*: *Keys-hom-components*)
then obtain *q* **where** *q*: $q \in$ *hom-components g* **and** $u \in$ *keys q* **by** (*rule in-KeysE*)
from *assms*(*1*) ‹$g \in$ *ideal F*› *q* **have** $q \in$ *ideal F* **by** (*rule homogeneous-ideal′*)
from *assms*(*2*) **have** *punit.is-Groebner-basis G* **by** (*rule punit.reduced-GB-D1*)
moreover from ‹$q \in$ *ideal F*› **have** $q \in$ *ideal G* **by** (*simp only*: *assms*(*3*))
moreover from *q* **have** $q \neq 0$ **by** (*rule hom-components-nonzero*)
ultimately obtain g' **where** $g' \in G$ **and** $g' \neq 0$ **and** *adds*: *lpp g′ adds lpp q*
by (*rule punit.GB-adds-lt*[*simplified*])
from ‹$q \neq 0$› **have** *lpp q* $\in$ *keys q* **by** (*rule punit.lt-in-keys*)
also from *q* **have** ... $\subseteq$ *Keys* (*hom-components g*) **by** (*rule keys-subset-Keys*)
finally have *lpp q* $\in$ *keys g* **by** (*simp only*: *Keys-hom-components*)
with - ‹$g' \neq 0$› **have** *red*: *punit.is-red* $\{g'\}$ *g*
using *adds* **by** (*rule punit.is-red-addsI*[*simplified*]) *simp*
from *assms*(*2*) **have** *punit.is-auto-reduced G* **by** (*rule punit.reduced-GB-D2*)
hence ¬ *punit.is-red* $(G - \{g\})$ *g* **using** *assms*(*4*) **by** (*rule punit.is-auto-reducedD*)
with *red* **have** $\neg \{g'\} \subseteq G - \{g\}$ **using** *punit.is-red-subset* **by** *blast*
with ‹$g' \in G$› **have** $g' = g$ **by** *simp*
from ‹*lpp q* $\in$ *keys g*› **have** *lpp q* $\preceq$ *lpp g* **by** (*rule punit.lt-max-keys*)
moreover from *adds* **have** *lpp g* $\preceq$ *lpp q*
unfolding ‹$g' = g$› **by** (*rule punit.ord-adds-term*[*simplified*])
ultimately have *eq*: *lpp q = lpp g*
by *simp*
from *q* **have** *homogeneous q* **by** (*rule hom-components-homogeneous*)
hence *deg-pm u = deg-pm* (*lpp q*)
using ‹$u \in$ *keys q*› ‹*lpp q* $\in$ *keys q*› **by** (*rule homogeneousD*)
thus *?thesis* **by** (*simp only*: *eq*)
qed
assume $s \in$ *keys g*
hence *2*: *deg-pm s = deg-pm* (*lpp g*) **by** (*rule 1*)
assume $t \in$ *keys g*
hence *deg-pm t = deg-pm* (*lpp g*) **by** (*rule 1*)
with *2* **show** *deg-pm s = deg-pm t* **by** *simp*
qed

lemma *lp-dehomogenize*:
assumes *is-hom-ord x* **and** *homogeneous p*

shows *lpp* (*dehomogenize x p*) = *except* (*lpp p*) {*x*}
proof (*cases p* = *0*)
  case *True*
  thus *?thesis* **by** *simp*
next
  case *False*
  hence *lpp p* ∈ *keys p* **by** (*rule punit.lt-in-keys*)
  with *assms*(*2*) **have** *except* (*lpp p*) {*x*} ∈ *keys* (*dehomogenize x p*)
    by (*rule keys-dehomogenizeI*)
  thus *?thesis*
  proof (*rule punit.lt-eqI-keys*)
    fix *t*
    assume *t* ∈ *keys* (*dehomogenize x p*)
    then obtain *s* **where** *s* ∈ *keys p* **and** *t*: *t* = *except s* {*x*} **by** (*rule keys-dehomogenizeE*)
    from *this*(*1*) **have** *s* ⪯ *lpp p* **by** (*rule punit.lt-max-keys*)
    moreover from *assms*(*2*) ‹*s* ∈ *keys p*› ‹*lpp p* ∈ *keys p*› **have** *deg-pm s* = *deg-pm* (*lpp p*)
      by (*rule homogeneousD*)
    ultimately show *t* ⪯ *except* (*lpp p*) {*x*} **using** *assms*(*1*) **by** (*simp add*: *t is-hom-ordD*)
  qed
qed

lemma *isGB-dehomogenize*:
  assumes *is-hom-ord x* **and** *finite X* **and** *G* ⊆ *P*[*X*] **and** *punit.is-Groebner-basis G*
    and ⋀*g*. *g* ∈ *G* ⟹ *homogeneous g*
  shows *punit.is-Groebner-basis* (*dehomogenize x* ‘ *G*)
  using *dickson-grading-varnum*
proof (*rule punit.isGB-I-adds-lt*[*simplified*])
  from *assms*(*2*) **show** *finite* (*X* − {*x*}) **by** *simp*
next
  have *dehomogenize x* ‘ *G* ⊆ *P*[*X* − {*x*}]
  proof
    fix *g*
    assume *g* ∈ *dehomogenize x* ‘ *G*
    then obtain *g'* **where** *g'* ∈ *G* **and** *g*: *g* = *dehomogenize x g'* **..**
    from *this*(*1*) *assms*(*3*) **have** *g'* ∈ *P*[*X*] **..**
    hence *indets g'* ⊆ *X* **by** (*rule PolysD*)
    have *indets g* ⊆ *indets g'* − {*x*} **by** (*simp only*: *g indets-dehomogenize*)
    also from ‹*indets g'* ⊆ *X*› *subset-refl* **have** ... ⊆ *X* − {*x*} **by** (*rule Diff-mono*)
    finally show *g* ∈ *P*[*X* − {*x*}] **by** (*rule PolysI-alt*)
  qed
  thus *dehomogenize x* ‘ *G* ⊆ *punit.dgrad-p-set* (*varnum* (*X* − {*x*})) *0*
    by (*simp only*: *dgrad-p-set-varnum*)
next
  fix *p*
  assume *p* ∈ *ideal* (*dehomogenize x* ‘ *G*)
  then obtain *G0 q* **where** *G0* ⊆ *dehomogenize x* ‘ *G* **and** *finite G0* **and** *p*: *p* =

$(\sum g \in G0.\ q\ g * g)$
by (*rule ideal.spanE*)
from *this*(*1*) **obtain** G' **where** $G' \subseteq G$ **and** *G0*: $G0 = dehomogenize\ x\ `\ G'$
and *inj*: *inj-on* (*dehomogenize x*) G' **by** (*rule subset-imageE-inj*)
define p' **where** $p' = (\sum g \in G'.\ q\ (dehomogenize\ x\ g) * g)$
have $p' \in ideal\ G'$ **unfolding** p'*-def* **by** (*rule ideal.sum-in-spanI*)
also from ‹$G' \subseteq G$› **have** ... $\subseteq$ *ideal G* **by** (*rule ideal.span-mono*)
finally have $p' \in ideal\ G$ **.**
with *assms*(*5*) **have** *homogenize x* $p' \in ideal\ G$ (**is** *?p* $\in$ -) **by** (*rule homogeneous-ideal-homogenize*)

assume $p \in punit.dgrad\text{-}p\text{-}set\ (varnum\ (X - \{x\}))\ 0$
hence $p \in P[X - \{x\}]$ **by** (*simp only*: *dgrad-p-set-varnum*)
hence *indets p* $\subseteq X - \{x\}$ **by** (*rule PolysD*)
hence $x \notin$ *indets p* **by** *blast*
have $p =$ *dehomogenize x p* **by** (*rule sym*) (*simp add*: ‹$x \notin$ *indets p*›)
also from *inj* **have** ... = *dehomogenize x* $(\sum g \in G'.\ q\ (dehomogenize\ x\ g) * dehomogenize\ x\ g)$
by (*simp add*: *p G0 sum.reindex*)
also have ... = *dehomogenize x ?p*
by (*simp add*: *dehomogenize-sum dehomogenize-times* p'*-def*)
finally have *p*: $p =$ *dehomogenize x ?p* **.**
moreover assume $p \neq 0$
ultimately have *?p* $\neq 0$ **by** (*auto simp del*: *dehomogenize-homogenize*)
with *assms*(*4*) ‹*?p* $\in$ *ideal G*› **obtain** *g* **where** $g \in G$ **and** $g \neq 0$ **and** *adds*: *lpp g adds lpp ?p*
by (*rule punit.GB-adds-lt*[*simplified*])
from *this*(*1*) **have** *homogeneous g* **by** (*rule assms*(*5*))
show $\exists g \in dehomogenize\ x\ `\ G.\ g \neq 0 \wedge$ *lpp g adds lpp p*
proof (*intro bexI conjI notI*)
assume *dehomogenize x g* $= 0$
hence $g = 0$ **using** ‹*homogeneous g*› **by** (*rule dehomogenize-zeroD*)
with ‹$g \neq 0$› **show** *False* **..**
next
from *assms*(*1*) ‹*homogeneous g*› **have** *lpp* (*dehomogenize x g*) = *except* (*lpp g*) $\{x\}$
by (*rule lp-dehomogenize*)
also from *adds* **have** ... *adds except* (*lpp ?p*) $\{x\}$
by (*simp add*: *adds-poly-mapping le-fun-def lookup-except*)
also from *assms*(*1*) *homogeneous-homogenize* **have** ... = *lpp* (*dehomogenize x ?p*)
by (*rule lp-dehomogenize*[*symmetric*])
finally show *lpp* (*dehomogenize x g*) *adds lpp p* **by** (*simp only*: *p*)
next
from ‹$g \in G$› **show** *dehomogenize x g* $\in$ *dehomogenize x* ‘ *G* **by** (*rule imageI*)
qed
qed

end

context *extended-ord-pm-powerprod*
begin

lemma *extended-ord-lp*:
  assumes *None* ∉ *indets p*
  shows *restrict-indets-pp* (*extended-ord.lpp p*) = *lpp* (*restrict-indets p*)
proof (*cases p* = *0*)
  case *True*
  thus *?thesis* **by** *simp*
next
  case *False*
  hence *extended-ord.lpp p* ∈ *keys p* **by** (*rule extended-ord.punit.lt-in-keys*)
  hence *restrict-indets-pp* (*extended-ord.lpp p*) ∈ *restrict-indets-pp* ' *keys p* **by** (*rule imageI*)
  also from *assms* **have** *eq*: ... = *keys* (*restrict-indets p*) **by** (*rule keys-restrict-indets*[*symmetric*])
  finally show *?thesis*
  proof (*rule punit.lt-eqI-keys*[*symmetric*])
    fix *t*
    assume *t* ∈ *keys* (*restrict-indets p*)
    then obtain *s* **where** *s* ∈ *keys p* **and** *t*: *t* = *restrict-indets-pp s* **unfolding** *eq*[*symmetric*] **..**
    from *this*(*1*) **have** *extended-ord s* (*extended-ord.lpp p*) **by** (*rule extended-ord.punit.lt-max-keys*)
    thus *t* ≼ *restrict-indets-pp* (*extended-ord.lpp p*) **by** (*auto simp*: *t extended-ord-def*)
  qed
qed

lemma *restrict-indets-reduced-GB*:
  assumes *finite X* **and** *F* ⊆ *P*[*X*]
  shows *punit.is-Groebner-basis* (*restrict-indets* ' *extended-ord.punit.reduced-GB* (*homogenize None* ' *extend-indets* ' *F*))
      (**is** *?thesis1*)
    and *ideal* (*restrict-indets* ' *extended-ord.punit.reduced-GB* (*homogenize None* ' *extend-indets* ' *F*)) = *ideal F*
      (**is** *?thesis2*)
    and *restrict-indets* ' *extended-ord.punit.reduced-GB* (*homogenize None* ' *extend-indets* ' *F*) ⊆ *P*[*X*]
      (**is** *?thesis3*)
proof −
  let *?F* = *homogenize None* ' *extend-indets* ' *F*
  let *?G* = *extended-ord.punit.reduced-GB ?F*
  from *assms*(*1*) **have** *finite* (*insert None* (*Some* ' *X*)) **by** *simp*
  moreover have *?F* ⊆ *P*[*insert None* (*Some* ' *X*)]
  proof
    fix *hf*
    assume *hf* ∈ *?F*
    then obtain *f* **where** *f* ∈ *F* **and** *hf*: *hf* = *homogenize None* (*extend-indets f*) **by** *auto*
    from *this*(*1*) *assms*(*2*) **have** *f* ∈ *P*[*X*] **..**

hence *indets f* ⊆ *X* **by** (*rule PolysD*)
hence *Some* ' *indets f* ⊆ *Some* ' *X* **by** (*rule image-mono*)
with *indets-extend-indets*[*of f*] **have** *indets* (*extend-indets f*) ⊆ *Some* ' *X* **by** *blast*
hence *insert None* (*indets* (*extend-indets f*)) ⊆ *insert None* (*Some* ' *X*) **by** *blast*
with *indets-homogenize-subset* **have** *indets hf* ⊆ *insert None* (*Some* ' *X*)
unfolding *hf* **by** (*rule subset-trans*)
thus *hf* ∈ *P*[*insert None* (*Some* ' *X*)] **by** (*rule PolysI-alt*)
qed
ultimately have *G-sub*: *?G* ⊆ *P*[*insert None* (*Some* ' *X*)]
and *ideal-G*: *ideal ?G* = *ideal ?F*
and *GB-G*: *extended-ord.punit.is-reduced-GB ?G*
by (*rule extended-ord.reduced-GB-Polys*, *rule extended-ord.reduced-GB-ideal-Polys*, *rule extended-ord.reduced-GB-is-reduced-GB-Polys*)

show *?thesis3*
proof
fix *g*
assume *g* ∈ *restrict-indets* ' *?G*
then obtain *g'* **where** *g'* ∈ *?G* **and** *g*: *g* = *restrict-indets g'* **..**
from *this*(*1*) *G-sub* **have** *g'* ∈ *P*[*insert None* (*Some* ' *X*)] **..**
hence *indets g'* ⊆ *insert None* (*Some* ' *X*) **by** (*rule PolysD*)
have *indets g* ⊆ *the* ' (*indets g'* − {*None*}) **by** (*simp only*: *g indets-restrict-indets-subset*)
also from ‹*indets g'* ⊆ *insert None* (*Some* ' *X*)› **have** ... ⊆ *X* **by** *auto*
finally show *g* ∈ *P*[*X*] **by** (*rule PolysI-alt*)
qed

from *dickson-grading-varnum* **show** *?thesis1*
proof (*rule punit.isGB-I-adds-lt*[*simplified*])
from ‹*?thesis3*› **show** *restrict-indets* ' *?G* ⊆ *punit.dgrad-p-set* (*varnum X*) *0*
by (*simp only*: *dgrad-p-set-varnum*)
next
fix *p* :: ('*a* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ '*b*
assume *p* ≠ *0*
assume *p* ∈ *ideal* (*restrict-indets* ' *?G*)
hence *extend-indets p* ∈ *extend-indets* ' *ideal* (*restrict-indets* ' *?G*) **by** (*rule imageI*)
also have ... ⊆ *ideal* (*extend-indets* ' *restrict-indets* ' *?G*) **by** (*fact extend-indets-ideal-subset*)
also have ... = *ideal* (*dehomogenize None* ' *?G*)
by (*simp only*: *image-comp extend-indets-comp-restrict-indets*)
finally have *p-in-ideal*: *extend-indets p* ∈ *ideal* (*dehomogenize None* ' *?G*) **.**
assume *p* ∈ *punit.dgrad-p-set* (*varnum X*) *0*
hence *p* ∈ *P*[*X*] **by** (*simp only*: *dgrad-p-set-varnum*)
have *extended-ord.punit.is-Groebner-basis* (*dehomogenize None* ' *?G*)
using *extended-ord-is-hom-ord* ‹*finite* (*insert None* (*Some* ' *X*))› *G-sub*
proof (*rule extended-ord.isGB-dehomogenize*)
from *GB-G* **show** *extended-ord.punit.is-Groebner-basis ?G*

```
        by (rule extended-ord.punit.reduced-GB-D1)
    next
      fix g
      assume g ∈ ?G
      with - GB-G ideal-G show homogeneous g
      proof (rule extended-ord.is-reduced-GB-homogeneous)
        fix hf
        assume hf ∈ ?F
        then obtain f where hf = homogenize None f ..
        thus homogeneous hf by (simp only: homogeneous-homogenize)
      qed
    qed
    moreover note p-in-ideal
    moreover from ‹p ≠ 0› have extend-indets p ≠ 0 by simp
    ultimately obtain g where g-in: g ∈ dehomogenize None ‘ ?G and g ≠ 0
      and adds: extended-ord.lpp g adds extended-ord.lpp (extend-indets p)
      by (rule extended-ord.punit.GB-adds-lt[simplified])
    have None ∉ indets g
    proof
      assume None ∈ indets g
      moreover from g-in obtain g0 where g = dehomogenize None g0 ..
      ultimately show False using indets-dehomogenize[of None g0] by blast
    qed
    show ∃ g∈restrict-indets ‘ ?G. g ≠ 0 ∧ lpp g adds lpp p
    proof (intro bexI conjI notI)
      have lpp (restrict-indets g) = restrict-indets-pp (extended-ord.lpp g)
        by (rule sym, intro extended-ord-lp ‹None ∉ indets g›)
      also from adds have ... adds restrict-indets-pp (extended-ord.lpp (extend-indets
p))
        by (simp add: adds-poly-mapping le-fun-def lookup-restrict-indets-pp)
      also have ... = lpp (restrict-indets (extend-indets p))
      proof (intro extended-ord-lp notI)
        assume None ∈ indets (extend-indets p)
        thus False by (simp add: indets-extend-indets)
      qed
      also have ... = lpp p by simp
      finally show lpp (restrict-indets g) adds lpp p .
    next
      from g-in have restrict-indets g ∈ restrict-indets ‘ dehomogenize None ‘ ?G
by (rule imageI)
      also have ... = restrict-indets ‘ ?G by (simp only: image-comp restrict-indets-comp-dehomogenize)
      finally show restrict-indets g ∈ restrict-indets ‘ ?G .
    next
      assume restrict-indets g = 0
      with ‹None ∉ indets g› extend-restrict-indets have g = 0 by fastforce
      with ‹g ≠ 0› show False ..
    qed
  qed (fact assms(1))
```

from *ideal-G* **show** *?thesis2* **by** (*rule ideal-restrict-indets*)
qed

end

end

References

[1] W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Society, July 1994.

[2] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. English translation in *Journal of Symbolic Computation* 41(3–4):475–511, Special Issue on Logic, Mathematics, and Computer Science: Interactions.

[3] B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems (An Algorithmic Criterion for the Solvability of an Algebraic System of Equations). *Aequationes Mathematicae*, pages 374–383, 1970. (English translation in *Gröbner Bases and Applications (Proceedings of the International Conference "33 Years of Gröbner Bases", 1998)*, London Mathematical Society Lecture Note Series 251, Cambridge Univerity Press, 1998, pages 535–545).

[4] B. Buchberger. A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases. In E. W. Ng, editor, *Symbolic and Algebraic Computation (Proceedings of EUROSAM'79, Marseille, June 26-28)*, volume 72 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 1979.

[5] B. Buchberger. Introduction to Gröbner Bases. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications*, number 251 in London Mathematical Society Lectures Notes Series, pages 3 – 31. Cambridge University Press, 1998.

[6] B. Buchberger. Gröbner Rings in Theorema: A Case Study in Functors and Categories. Technical Report 2003-49, Johannes Kepler University Linz, Spezialforschungsbereich F013, November 2003.

[7] A. Chaieb and M. Wenzel. Context aware Calculation and Deduction: Ring Equalities via Gröbner Bases in Isabelle. In M. Kauers, M. Ker-

ber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (Proceedings of Calculemus'2007, Hagenberg, Austria, June 27–30)*, volume 4573 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2007.

[8] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F_4). *Journal of Pure and Applied Algebra*, 139(1):61–88, 1999.

[9] 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 Press, 2002.

[10] J. S. Jorge, V. M. Guilas, and J. L. Freire. Certifying properties of an efficient functional program for computing Gröbner bases. *Journal of Symbolic Computation*, 44(5):571–582, 2009.

[11] M. Kreuzer and L. Robbiano. *Computational Commutative Algebra 1*. Springer-Verlag, 2000.

[12] A. Maletzky. *Computer-Assisted Exploration of Gröbner Bases Theory in Theorema*. PhD thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, May 2016. To appear.

[13] I. Medina-Bulo, F. Palomo-Lozano, and J.-L. Ruiz-Reina. A verified Common Lisp implementation of Buchberger's algorithm in ACL2. *Journal of Symbolic Computation*, 45(1):96–123, 2010.

[14] T. Mora. An Introduction to Commutative and Non-Commutative Gröbner Bases. *Theoretical Computer Science*, 134(1):131–173, 1994.

[15] C. Schwarzweller. Gröbner Bases – Theory Refinement in the Mizar System. In M. Kohlhase, editor, *Mathematical Knowledge Management (4th International Conference, Bremen, Germany, July 15–17)*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 299–314. Springer, 2006.

[16] L. Théry. A Machine-Checked Implementation of Buchberger's Algorithm. *Journal of Automated Reasoning*, 26(2):107–137, 2001.

[17] F. Winkler and B. Buchberger. A Criterion for Eliminating Unnecessary Reductions in the Knuth-Bendix Algorithm. In J. Demetrovics, G. Katona, and A. Salomaa, editors, *Proceedings of Algebra and Logic in Computer Science, Győr, Hungary*, volume 42 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 849–869. North Holland, 1983.