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*: $\mathit{distinct}\ (xs\ @\ (y\ \#\ ys)) = \mathit{distinct}\ (y\ \#\ (xs\ @\ ys))$ $\langle \mathit{proof} \rangle$

lemma *set-reorder*: $\mathit{set}\ (xs\ @\ (y\ \#\ ys)) = \mathit{set}\ (y\ \#\ (xs\ @\ ys))$ $\langle \mathit{proof} \rangle$

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*
  $\langle \mathit{proof} \rangle$

lemma *filter-nth-pairE*:
  assumes $i < j$ **and** $i < \mathit{length}\ (\mathit{filter}\ P\ xs)$ **and** $j < \mathit{length}\ (\mathit{filter}\ P\ xs)$
  obtains $i'\ j'$ **where** $i' < j'$ **and** $i' < \mathit{length}\ xs$ **and** $j' < \mathit{length}\ xs$
    and $(\mathit{filter}\ P\ xs)\ !\ i = xs\ !\ i'$ **and** $(\mathit{filter}\ P\ xs)\ !\ j = xs\ !\ j'$
  $\langle \mathit{proof} \rangle$

lemma *distinct-filterI*:
  assumes $\bigwedge i\ j.\ i < j \Longrightarrow i < \mathit{length}\ xs \Longrightarrow j < \mathit{length}\ xs \Longrightarrow P\ (xs\ !\ i) \Longrightarrow P\ (xs\ !\ j) \Longrightarrow xs\ !\ i \neq xs\ !\ j$
  shows $\mathit{distinct}\ (\mathit{filter}\ P\ xs)$
$\langle \mathit{proof} \rangle$

lemma *set-zip-map*: $set\ (zip\ (map\ f\ xs)\ (map\ g\ xs)) = (\lambda x.\ (f\ x,\ g\ x))\ `\ (set\ xs)$
⟨*proof*⟩

lemma *set-zip-map1*: $set\ (zip\ (map\ f\ xs)\ xs) = (\lambda x.\ (f\ x,\ x))\ `\ (set\ xs)$
⟨*proof*⟩

lemma *set-zip-map2*: $set\ (zip\ xs\ (map\ f\ xs)) = (\lambda x.\ (x,\ f\ x))\ `\ (set\ xs)$
⟨*proof*⟩

lemma *UN-upt*: $(\bigcup i \in \{0..<length\ xs\}.\ f\ (xs\ !\ i)) = (\bigcup x \in set\ xs.\ f\ x)$
⟨*proof*⟩

lemma *sum-list-zeroI'*:
assumes $\bigwedge i.\ i < length\ xs \Longrightarrow xs\ !\ i = 0$
shows $sum\text{-}list\ xs = 0$
⟨*proof*⟩

lemma *sum-list-map2-plus*:
assumes $length\ xs = length\ ys$
shows $sum\text{-}list\ (map2\ (+)\ xs\ ys) = sum\text{-}list\ xs + sum\text{-}list\ (ys::'a::comm\text{-}monoid\text{-}add\ list)$
⟨*proof*⟩

lemma *sum-list-eq-nthI*:
assumes $i < length\ xs$ **and** $\bigwedge j.\ j < length\ xs \Longrightarrow j \neq i \Longrightarrow xs\ !\ j = 0$
shows $sum\text{-}list\ xs = xs\ !\ i$
⟨*proof*⟩

2.1.1 *max-list*

fun (**in** *ord*) $max\text{-}list :: 'a\ list \Rightarrow 'a$ **where**
$max\text{-}list\ (x\ \#\ xs) = (case\ xs\ of\ [] \Rightarrow x\ |\ \text{-} \Rightarrow max\ x\ (max\text{-}list\ xs))$

context *linorder*
begin

lemma *max-list-Max*: $xs \neq [] \Longrightarrow max\text{-}list\ xs = Max\ (set\ xs)$
⟨*proof*⟩

lemma *max-list-ge*:
assumes $x \in set\ xs$
shows $x \leq max\text{-}list\ xs$
⟨*proof*⟩

lemma *max-list-boundedI*:
assumes $xs \neq []$ **and** $\bigwedge x.\ x \in set\ xs \Longrightarrow x \leq a$
shows $max\text{-}list\ xs \leq a$
⟨*proof*⟩

end

2.1.2 *insort-wrt*

primrec *insort-wrt* :: $('c \Rightarrow 'c \Rightarrow bool) \Rightarrow 'c \Rightarrow 'c\ list \Rightarrow 'c\ list$ **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* ≠ []
⟨*proof*⟩

lemma *length-insort-wrt* [*simp*]: *length* (*insort-wrt r x xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *set-insort-wrt* [*simp*]: *set* (*insort-wrt r x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

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

lemma *sorted-wrt-imp-sorted-wrt-insort-wrt*:
assumes *transp r* **and** $\bigwedge a.\ r\ a\ x \lor r\ x\ a$ **and** *sorted-wrt r xs*
shows *sorted-wrt r* (*insort-wrt r x xs*)
⟨*proof*⟩

corollary *sorted-wrt-insort-wrt*:
assumes *transp r* **and** $\bigwedge a.\ r\ a\ x \lor r\ x\ a$
shows *sorted-wrt r* (*insort-wrt r x xs*) ⟷ *sorted-wrt r xs* (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

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

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

lemma *set-diff-list*: *set* (*xs* −− *ys*) = *set xs* − *set ys*
⟨*proof*⟩

lemma *diff-list-disjoint*: *set ys* ∩ *set* (*xs* −− *ys*) = {}
⟨*proof*⟩

lemma *subset-append-diff-cancel*:
assumes *set ys* ⊆ *set xs*
shows *set* (*ys* @ (*xs* −− *ys*)) = *set xs*
⟨*proof*⟩

definition *insert-list* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$

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*)
⟨*proof*⟩

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*⟩

lemma *subset-remdups-wrt*: *set* (*remdups-wrt f xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *remdups-wrt-distinct-wrt*:
assumes *x* ∈ *set* (*remdups-wrt f xs*) **and** *y* ∈ *set* (*remdups-wrt f xs*) **and** *x* ≠ *y*
shows *f x* ≠ *f y*
⟨*proof*⟩

lemma *distinct-remdups-wrt*: *distinct* (*remdups-wrt f xs*)
⟨*proof*⟩

lemma *map-remdups-wrt*: *map f* (*remdups-wrt f xs*) = *remdups* (*map f xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *length-map-idx* [*simp*]: *length* (*map-idx f xs n*) = *length xs*
⟨*proof*⟩

lemma *map-idx-append*: *map-idx f* (*xs* @ *ys*) *n* = (*map-idx f xs n*) @ (*map-idx f ys* (*n* + *length xs*))
⟨*proof*⟩

lemma *map-idx-nth*:
  assumes $i < length\ xs$
  shows $(map\text{-}idx\ f\ xs\ n)\ !\ i = f\ (xs\ !\ i)\ (n + i)$
  ⟨*proof*⟩

lemma *map-map-idx*: $map\ f\ (map\text{-}idx\ g\ xs\ n) = map\text{-}idx\ (\lambda x\ i.\ f\ (g\ x\ i))\ xs\ n$
  ⟨*proof*⟩

lemma *map-idx-map*: $map\text{-}idx\ f\ (map\ g\ xs)\ n = map\text{-}idx\ (f \circ g)\ xs\ n$
  ⟨*proof*⟩

lemma *map-idx-no-idx*: $map\text{-}idx\ (\lambda x\ \text{-}.\ f\ x)\ xs\ n = map\ f\ xs$
  ⟨*proof*⟩

lemma *map-idx-no-elem*: $map\text{-}idx\ (\lambda\text{-}.\ f)\ xs\ n = map\ f\ [n..<n + length\ xs]$
⟨*proof*⟩

lemma *map-idx-eq-map*: $map\text{-}idx\ f\ xs\ n = map\ (\lambda i.\ f\ (xs\ !\ i)\ (i + n))\ [0..<length\ xs]$
⟨*proof*⟩

lemma *set-map-idx*: $set\ (map\text{-}idx\ f\ xs\ n) = (\lambda i.\ f\ (xs\ !\ i)\ (i + n))\ \text{`}\ \{0..<length\ xs\}$
  ⟨*proof*⟩

2.1.6 *map-dup*

primrec *map-dup* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
  $map\text{-}dup\ \text{-}\ \text{-}\ [] = []\ |$
  $map\text{-}dup\ f\ g\ (x\ \#\ xs) = (if\ x \in set\ xs\ then\ g\ x\ else\ f\ x)\ \#\ (map\text{-}dup\ f\ g\ xs)$

lemma *length-map-dup*[*simp*]: $length\ (map\text{-}dup\ f\ g\ xs) = length\ xs$
  ⟨*proof*⟩

lemma *map-dup-distinct*:
  assumes *distinct xs*
  shows $map\text{-}dup\ f\ g\ xs = map\ f\ xs$
  ⟨*proof*⟩

lemma *filter-map-dup-const*:
  $filter\ (\lambda x.\ x \neq c)\ (map\text{-}dup\ f\ (\lambda\text{-}.\ c)\ xs) = filter\ (\lambda x.\ x \neq c)\ (map\ f\ (remdups\ xs))$
  ⟨*proof*⟩

lemma *filter-zip-map-dup-const*:
  $filter\ (\lambda(a, b).\ a \neq c)\ (zip\ (map\text{-}dup\ f\ (\lambda\text{-}.\ c)\ xs)\ xs) =$
    $filter\ (\lambda(a, b).\ a \neq c)\ (zip\ (map\ f\ (remdups\ xs))\ (remdups\ xs))$
  ⟨*proof*⟩

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*⟩

lemma *filter-min-aux-subset*: *set* (*filter-min-aux xs ys*) ⊆ *set xs* ∪ *set ys*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *filter-min-aux-distinct*:
  assumes *reflp rel* **and** *distinct ys*
  shows *distinct* (*filter-min-aux xs ys*)
  ⟨*proof*⟩

lemma *filter-min-subset*: *set* (*filter-min xs*) ⊆ *set xs*
  ⟨*proof*⟩

lemma *filter-min-cases*:
  assumes *transp rel* **and** *x* ∈ *set xs*
  assumes *x* ∈ *set* (*filter-min xs*) ⟹ *thesis*

assumes $\bigwedge y.\ y \in set\ (\textit{filter-min}\ xs) \Longrightarrow x \notin set\ (\textit{filter-min}\ xs) \Longrightarrow rel\ y\ x \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

corollary *filter-min-relE*:
assumes *transp rel* **and** *reflp rel* **and** $x \in set\ xs$
obtains y **where** $y \in set\ (\textit{filter-min}\ xs)$ **and** $rel\ y\ x$
⟨*proof*⟩

lemma *filter-min-minimal*:
assumes *transp rel* **and** $x \in set\ (\textit{filter-min}\ xs)$ **and** $y \in set\ (\textit{filter-min}\ xs)$ **and** $rel\ x\ y$
shows $x = y$
⟨*proof*⟩

lemma *filter-min-distinct*:
assumes *reflp rel*
shows *distinct* (*filter-min xs*)
⟨*proof*⟩

lemma *filter-min-append-subset*: $set\ (\textit{filter-min-append}\ xs\ ys) \subseteq set\ xs \cup set\ ys$
⟨*proof*⟩

lemma *filter-min-append-cases*:
assumes *transp rel* **and** $x \in set\ xs \cup set\ ys$
assumes $x \in set\ (\textit{filter-min-append}\ xs\ ys) \Longrightarrow thesis$
assumes $\bigwedge y.\ y \in set\ (\textit{filter-min-append}\ xs\ ys) \Longrightarrow x \notin set\ (\textit{filter-min-append}\ xs\ ys) \Longrightarrow rel\ y\ x \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

corollary *filter-min-append-relE*:
assumes *transp rel* **and** *reflp rel* **and** $x \in set\ xs \cup set\ ys$
obtains y **where** $y \in set\ (\textit{filter-min-append}\ xs\ ys)$ **and** $rel\ y\ x$
⟨*proof*⟩

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 $x \in set\ (\textit{filter-min-append}\ xs\ ys)$ **and** $y \in set\ (\textit{filter-min-append}\ xs\ ys)$ **and** $rel\ x\ y$
shows $x = y$
⟨*proof*⟩

lemma *filter-min-append-distinct*:
assumes *reflp rel* **and** *distinct xs* **and** *distinct ys*
shows *distinct* (*filter-min-append xs ys*)
⟨*proof*⟩

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 $wfp\text{-}on\ r\ A$
shows $wfP\ (\lambda x\ y.\ r\ x\ y \wedge x \in A \wedge y \in A)$ (**is** $wfP\ ?r$)
⟨*proof*⟩

lemma *wfp-onI-min*:
assumes $\bigwedge x\ Q.\ x \in Q \Longrightarrow Q \subseteq A \Longrightarrow \exists z{\in}Q.\ \forall y{\in}A.\ r\ y\ z \longrightarrow y \notin Q$
shows $wfp\text{-}on\ r\ A$
⟨*proof*⟩

lemma *wfp-onE-min*:
assumes $wfp\text{-}on\ r\ A$ **and** $x \in Q$ **and** $Q \subseteq A$
obtains z **where** $z \in Q$ **and** $\bigwedge y.\ r\ y\ z \Longrightarrow y \notin Q$
⟨*proof*⟩

lemma *wfp-onI-chain*: $\neg\ (\exists f.\ \forall i.\ f\ i \in A \wedge r\ (f\ (Suc\ i))\ (f\ i)) \Longrightarrow wfp\text{-}on\ r\ A$
⟨*proof*⟩

lemma *finite-minimalE*:
assumes $finite\ A$ **and** $A \neq \{\}$ **and** $irreflp\ rel$ **and** $transp\ rel$
obtains a **where** $a \in A$ **and** $\bigwedge b.\ rel\ b\ a \Longrightarrow b \notin A$
⟨*proof*⟩

lemma *wfp-on-finite*:
assumes $irreflp\ rel$ **and** $transp\ rel$ **and** $finite\ A$
shows $wfp\text{-}on\ rel\ A$
⟨*proof*⟩

3.2 Relations

locale *relation* = **fixes** $r{::}'a \Rightarrow 'a \Rightarrow bool$ (**infixl** ‹→› *50*)
begin

abbreviation *rtc*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\to^*$› *50*)
  where *rtc a b* $\equiv r^{**}$ *a b*

abbreviation *sc*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\leftrightarrow$› *50*)
  where *sc a b* $\equiv a \to b \vee b \to a$

definition *is-final*::$'a \Rightarrow$ *bool* **where**
  is-final a $\equiv \neg (\exists\, b.\ r\ a\ b)$

definition *srtc*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\leftrightarrow^*$› *50*) **where**
  srtc a b $\equiv sc^{**}$ *a b*
definition *cs*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\downarrow^*$› *50*) **where**
  cs a b $\equiv (\exists\, s.\ (a \to^* s) \wedge (b \to^* s))$

definition *is-confluent-on* :: $'a$ *set* $\Rightarrow$ *bool*
  where *is-confluent-on A* $\longleftrightarrow (\forall\, a{\in}A.\ \forall\, b1\ b2.\ (a \to^* b1 \wedge a \to^* b2) \longrightarrow b1 \downarrow^*$ $b2)$

definition *is-confluent* :: *bool*
  where *is-confluent* $\equiv$ *is-confluent-on UNIV*

definition *is-loc-confluent* :: *bool*
  where *is-loc-confluent* $\equiv (\forall\, a\ b1\ b2.\ (a \to b1 \wedge a \to b2) \longrightarrow b1 \downarrow^* b2)$

definition *is-ChurchRosser* :: *bool*
  where *is-ChurchRosser* $\equiv (\forall\, a\ b.\ a \leftrightarrow^* b \longrightarrow a \downarrow^* b)$

definition *dw-closed* :: $'a$ *set* $\Rightarrow$ *bool*
  where *dw-closed A* $\longleftrightarrow (\forall\, a{\in}A.\ \forall\, b.\ a \to b \longrightarrow b \in A)$

lemma *dw-closedI* [*intro*]:
  assumes $\bigwedge a\ b.\ a \in A \Longrightarrow a \to b \Longrightarrow b \in A$
  shows *dw-closed A*
  ⟨*proof*⟩

lemma *dw-closedD*:
  assumes *dw-closed A* **and** $a \in A$ **and** $a \to b$
  shows $b \in A$
  ⟨*proof*⟩

lemma *dw-closed-rtrancl*:
  assumes *dw-closed A* **and** $a \in A$ **and** $a \to^* b$
  shows $b \in A$
  ⟨*proof*⟩

lemma *dw-closed-empty*: *dw-closed* {}
  ⟨*proof*⟩

lemma *dw-closed-UNIV*: *dw-closed UNIV*

$\langle proof \rangle$

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* →* *b*
shows (*a*, *b*) ∈ *relset**
$\langle proof \rangle$

lemma *final-NF*: (*is-final a*) = (*a* ∈ *NF relset*)
$\langle proof \rangle$

lemma *sc-symcl*: (*a* ↔ *b*) = ((*a*, *b*) ∈ *relset*$^{\leftrightarrow}$)
$\langle proof \rangle$

lemma *srtc-conversion*: (*a* ↔* *b*) = ((*a*, *b*) ∈ *relset*$^{\leftrightarrow *}$)
$\langle proof \rangle$

lemma *cs-join*: (*a* ↓* *b*) = ((*a*, *b*) ∈ *relset*$^{\downarrow}$)
$\langle proof \rangle$

lemma *confluent-CR*: *is-confluent* = *CR relset*
$\langle proof \rangle$

lemma *ChurchRosser-conversion*: *is-ChurchRosser* = (*relset*$^{\leftrightarrow *}$ ⊆ *relset*$^{\downarrow}$)
$\langle proof \rangle$

lemma *loc-confluent-WCR*:
shows *is-loc-confluent* = *WCR relset*
$\langle proof \rangle$

lemma *wf-converse*:
shows (*wfP r*$\widehat{}$*−−1*) = (*wf* (*relset*$^{-1}$))
$\langle proof \rangle$

lemma *wf-SN*:
shows (*wfP r*$\widehat{}$*−−1*) = (*SN relset*)
$\langle proof \rangle$

3.4 Simple Lemmas

lemma *rtrancl-is-final*:
assumes *a* →* *b* **and** *is-final a*
shows *a* = *b*
$\langle proof \rangle$

lemma *cs-refl*:

shows $x \downarrow^* x$
⟨*proof*⟩

lemma *cs-sym*:
assumes $x \downarrow^* y$
shows $y \downarrow^* x$
⟨*proof*⟩

lemma *rtc-implies-cs*:
assumes $x \rightarrow^* y$
shows $x \downarrow^* y$
⟨*proof*⟩

lemma *rtc-implies-srtc*:
assumes $a \rightarrow^* b$
shows $a \leftrightarrow^* b$
⟨*proof*⟩

lemma *srtc-symmetric*:
assumes $a \leftrightarrow^* b$
shows $b \leftrightarrow^* a$
⟨*proof*⟩

lemma *srtc-transitive*:
assumes $a \leftrightarrow^* b$ **and** $b \leftrightarrow^* c$
shows $a \leftrightarrow^* c$
⟨*proof*⟩

lemma *cs-implies-srtc*:
assumes $a \downarrow^* b$
shows $a \leftrightarrow^* b$
⟨*proof*⟩

lemma *confluence-equiv-ChurchRosser*: *is-confluent* = *is-ChurchRosser*
⟨*proof*⟩

corollary *confluence-implies-ChurchRosser*:
assumes *is-confluent*
shows *is-ChurchRosser*
⟨*proof*⟩

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*⟩

lemma *wf-on-imp-nf-ex*:
assumes *wfp-on* $((\rightarrow)^{-1-1})$ A **and** *dw-closed* A **and** $a \in A$

obtains b **where** $a \to^* b$ **and** *is-final* b
⟨*proof*⟩

lemma *unique-nf-imp-confluence-on*:
assumes *major*: $\bigwedge a\ b1\ b2.\ a \in A \Longrightarrow (a \to^* b1) \Longrightarrow (a \to^* b2) \Longrightarrow \textit{is-final}\ b1 \Longrightarrow \textit{is-final}\ b2 \Longrightarrow b1 = b2$
and *wf*: *wfp-on* $((\to)^{-1-1})$ A **and** *dw*: *dw-closed* A
shows *is-confluent-on* A
⟨*proof*⟩

corollary *wf-imp-nf-ex*:
assumes *wfP* $((\to)^{-1-1})$
obtains b **where** $a \to^* b$ **and** *is-final* b
⟨*proof*⟩

corollary *unique-nf-imp-confluence*:
assumes $\bigwedge a\ b1\ b2.\ (a \to^* b1) \Longrightarrow (a \to^* b2) \Longrightarrow \textit{is-final}\ b1 \Longrightarrow \textit{is-final}\ b2 \Longrightarrow b1 = b2$
and *wfP* $((\to)^{-1-1})$
shows *is-confluent*
⟨*proof*⟩

end

3.5 Advanced Results and the Generalized Newman Lemma

definition *relbelow-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
where *relbelow-on* A *ord* z *rel* a $b \equiv (a \in A \land b \in A \land rel\ a\ b \land ord\ a\ z \land ord\ b\ z)$

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

definition *cbelow-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
where *cbelow-on* A *ord* z *rel* a $b \equiv (a = b \land b \in A \land ord\ b\ z) \lor$ *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* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *is-loc-connective-on* A *ord* $r \longleftrightarrow (\forall a{\in}A.\ \forall b1\ b2.\ r\ a\ b1 \land r\ a\ b2 \longrightarrow$ *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*
  ⟨*proof*⟩

lemma *cbelow-on-second-below*:
  assumes *cbelow-on A ord z rel a b*
  shows *ord b z*
  ⟨*proof*⟩

lemma *cbelow-on-first-in*:
  assumes *cbelow-on A ord z rel a b*
  shows $a \in A$
  ⟨*proof*⟩

lemma *cbelow-on-second-in*:
  assumes *cbelow-on A ord z rel a b*
  shows $b \in A$
  ⟨*proof*⟩

lemma *cbelow-on-intro* [*intro*]:
  assumes *main*: *cbelow-on A ord z rel a b* **and** $c \in A$ **and** *rel b c* **and** *ord c z*
  shows *cbelow-on A ord z rel a c*
⟨*proof*⟩

lemma *cbelow-on-induct* [*consumes 1*, *case-names base step*]:
  assumes *a*: *cbelow-on A ord z rel a b*
    and *base*: $a \in A \Longrightarrow$ *ord a z* $\Longrightarrow$ *P a*
    and *ind*: $\bigwedge b\ c.$ [| *cbelow-on A ord z rel a b*; *rel b c*; $c \in A$; *ord c z*; *P b* |] ==>
P c
  shows *P b*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *cbelow-on-mono*:
  assumes *cbelow-on A ord z rel a b* **and** $A \subseteq B$
  shows *cbelow-on B ord z rel a b*
  ⟨*proof*⟩

locale *relation-order* = *relation* +
  fixes *ord*::$'a \Rightarrow 'a \Rightarrow$ *bool*
  fixes *A*::$'a$ *set*
  assumes *trans*: *ord x y* $\Longrightarrow$ *ord y z* $\Longrightarrow$ *ord x z*
  assumes *wf*: *wfp-on ord A*
  assumes *refines*: $(\rightarrow) \leq ord^{-1-1}$
begin

lemma *relation-refines*:
  assumes $a \rightarrow b$
  shows *ord b a*
  ⟨*proof*⟩

lemma *relation-wf*: *wfp-on* $(\rightarrow)^{-1-1}$ *A*
  ⟨*proof*⟩

lemma *rtc-implies-cbelow-on*:
  assumes *dw-closed A* **and** *main*: $a \rightarrow^* b$ **and** $a \in A$ **and** *ord a c*
  shows *cbelow-on A ord c* $(\leftrightarrow)$ *a b*
  ⟨*proof*⟩

lemma *cs-implies-cbelow-on*:
  assumes *dw-closed A* **and** $a \downarrow^* b$ **and** $a \in A$ **and** $b \in A$ **and** *ord a c* **and** *ord b c*
  shows *cbelow-on A ord c* $(\leftrightarrow)$ *a b*
⟨*proof*⟩

The generalized Newman lemma, taken from [17]:

lemma *loc-connectivity-implies-confluence*:
  assumes *is-loc-connective-on A ord* $(\rightarrow)$ **and** *dw-closed A*
  shows *is-confluent-on A*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *red-setE*:
  assumes *red F p q*
  obtains *f* **and** *t* **where** $f \in F$ **and** *red-single p q f t*
⟨*proof*⟩

lemma *red-empty*: $\neg$ *red* {} *p q*
  ⟨*proof*⟩

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

lemma *red-union*: *red* $(F \cup G)$ *p q* = (*red F p q* $\vee$ *red G p q*)
⟨*proof*⟩

lemma *red-unionI1*:
  assumes *red F p q*
  shows *red* $(F \cup G)$ *p q*
  ⟨*proof*⟩

lemma *red-unionI2*:
  assumes *red G p q*
  shows *red* $(F \cup G)$ *p q*
  ⟨*proof*⟩

lemma *red-subset*:
  assumes *red G p q* **and** $G \subseteq F$
  shows *red F p q*
⟨*proof*⟩

lemma *red-union-singleton-zero*: $red\ (F \cup \{0\}) = red\ F$
⟨*proof*⟩

lemma *red-minus-singleton-zero*: $red\ (F - \{0\}) = red\ F$
⟨*proof*⟩

lemma *red-rtrancl-subset*:
assumes *major*: $(red\ G)^{**}\ p\ q$ **and** $G \subseteq F$
shows $(red\ F)^{**}\ p\ q$
⟨*proof*⟩

lemma *red-singleton*: $red\ \{f\}\ p\ q \longleftrightarrow (\exists\, t.\ red\text{-}single\ p\ q\ f\ t)$
⟨*proof*⟩

lemma *red-single-lookup*:
assumes $red\text{-}single\ p\ q\ f\ t$
shows $lookup\ q\ (t \oplus lt\ f) = 0$
⟨*proof*⟩

lemma *red-single-higher*:
assumes $red\text{-}single\ p\ q\ f\ t$
shows $higher\ q\ (t \oplus lt\ f) = higher\ p\ (t \oplus lt\ f)$
⟨*proof*⟩

lemma *red-single-ord*:
assumes $red\text{-}single\ p\ q\ f\ t$
shows $q \prec_p p$
⟨*proof*⟩

lemma *red-single-nonzero1*:
assumes $red\text{-}single\ p\ q\ f\ t$
shows $p \neq 0$
⟨*proof*⟩

lemma *red-single-nonzero2*:
assumes $red\text{-}single\ p\ q\ f\ t$
shows $f \neq 0$
⟨*proof*⟩

lemma *red-single-self*:
assumes $p \neq 0$
shows $red\text{-}single\ p\ 0\ p\ 0$
⟨*proof*⟩

lemma *red-single-trans*:
assumes $red\text{-}single\ p\ p0\ f\ t$ **and** $lt\ g\ adds_t\ lt\ f$ **and** $g \neq 0$
obtains $p1$ **where** $red\text{-}single\ p\ p1\ g\ (t + (lp\ f - lp\ g))$
⟨*proof*⟩

lemma *red-nonzero*:
  assumes *red F p q*
  shows $p \neq 0$
⟨*proof*⟩

lemma *red-self*:
  assumes $p \neq 0$
  shows *red* $\{p\}$ *p 0*
⟨*proof*⟩

lemma *red-ord*:
  assumes *red F p q*
  shows $q \prec_p p$
⟨*proof*⟩

lemma *red-indI1*:
  assumes $f \in F$ **and** $f \neq 0$ **and** $p \neq 0$ **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*⟩

lemma *red-indI2*:
  assumes $p \neq 0$ **and** *r*: *red F* (*tail p*) *q*
  shows *red F p* (*q* + *monomial* (*lc p*) (*lt p*))
⟨*proof*⟩

lemma *red-indE*:
  assumes *red F p q*
  shows (∃ *f*∈*F*. $f \neq 0$ ∧ *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*⟩

lemma *is-redI*:
  assumes *red F a b*
  shows *is-red F a*
  ⟨*proof*⟩

lemma *is-redE*:
  assumes *is-red F a*
  obtains *b* **where** *red F a b*
  ⟨*proof*⟩

lemma *is-red-alt*:
  shows *is-red F a* ⟷ (∃ *b*. *red F a b*)
⟨*proof*⟩

lemma *is-red-singletonI*:
  assumes *is-red F q*

obtains p **where** $p \in F$ **and** *is-red* $\{p\}$ q
⟨*proof*⟩

lemma *is-red-singletonD*:
assumes *is-red* $\{p\}$ q **and** $p \in F$
shows *is-red* F q
⟨*proof*⟩

lemma *is-red-singleton-trans*:
assumes *is-red* $\{f\}$ p **and** *lt* g $adds_t$ *lt* f **and** $g \neq 0$
shows *is-red* $\{g\}$ p
⟨*proof*⟩

lemma *is-red-singleton-not-0*:
assumes *is-red* $\{f\}$ p
shows $f \neq 0$
⟨*proof*⟩

lemma *irred-0*:
shows $\neg$ *is-red* F *0*
⟨*proof*⟩

lemma *is-red-indI1*:
assumes $f \in F$ **and** $f \neq 0$ **and** $p \neq 0$ **and** *lt* f $adds_t$ *lt* p
shows *is-red* F p
⟨*proof*⟩

lemma *is-red-indI2*:
assumes $p \neq 0$ **and** *is-red* F (*tail* p)
shows *is-red* F p
⟨*proof*⟩

lemma *is-red-indE*:
assumes *is-red* F p
shows ($\exists f \in F$. $f \neq 0 \wedge$ *lt* f $adds_t$ *lt* p) $\vee$ *is-red* F (*tail* p)
⟨*proof*⟩

lemma *rtrancl-0*:
assumes (*red* F)** *0* x
shows $x = 0$
⟨*proof*⟩

lemma *red-rtrancl-ord*:
assumes (*red* F)** p q
shows $q \preceq_p p$
⟨*proof*⟩

lemma *components-red-subset*:
assumes *red* F p q

shows *component-of-term* ‘ *keys* $q \subseteq$ *component-of-term* ‘ *keys* $p \cup$ *component-of-term* ‘ *Keys* F
⟨*proof*⟩

corollary *components-red-rtrancl-subset*:
assumes (*red* F)** p q
shows *component-of-term* ‘ *keys* $q \subseteq$ *component-of-term* ‘ *keys* $p \cup$ *component-of-term* ‘ *Keys* F
⟨*proof*⟩

4.2 Reducibility and Addition & Multiplication

lemma *red-single-monom-mult*:
assumes *red-single* p q f t **and** $c \neq 0$
shows *red-single* (*monom-mult* c s p) (*monom-mult* c s q) f $(s + t)$
⟨*proof*⟩

lemma *red-single-plus-1*:
assumes *red-single* p q f t **and** $t \oplus$ *lt* $f \notin$ *keys* $(p + r)$
shows *red-single* $(q + r)$ $(p + r)$ f t
⟨*proof*⟩

lemma *red-single-plus-2*:
assumes *red-single* p q f t **and** $t \oplus$ *lt* $f \notin$ *keys* $(q + r)$
shows *red-single* $(p + r)$ $(q + r)$ f t
⟨*proof*⟩

lemma *red-single-plus-3*:
assumes *red-single* p q f t **and** $t \oplus$ *lt* $f \in$ *keys* $(p + r)$ **and** $t \oplus$ *lt* $f \in$ *keys* $(q + r)$
shows $\exists\, s.$ *red-single* $(p + r)$ s f t $\wedge$ *red-single* $(q + r)$ s f t
⟨*proof*⟩

lemma *red-single-plus*:
assumes *red-single* p q f t
shows *red-single* $(p + r)$ $(q + r)$ f t $\vee$
red-single $(q + r)$ $(p + r)$ f t $\vee$
($\exists\, s.$ *red-single* $(p + r)$ s f t $\wedge$ *red-single* $(q + r)$ s f t) (**is** $?A \vee ?B \vee ?C$)
⟨*proof*⟩

lemma *red-single-diff*:
assumes *red-single* $(p - q)$ r f t
shows *red-single* p $(r + q)$ f t $\vee$ *red-single* q $(p - r)$ f t $\vee$
($\exists\, p'\ q'.$ *red-single* p p' f t $\wedge$ *red-single* q q' f t $\wedge$ $r = p' - q'$) (**is** $?A \vee ?B \vee ?C$)
⟨*proof*⟩

lemma *red-monom-mult*:
assumes a: *red* F p q **and** $c \neq 0$

shows *red F* (*monom-mult c s p*) (*monom-mult c s q*)
⟨*proof*⟩

lemma *red-plus-keys-disjoint*:
assumes *red F p q* **and** *keys p* ∩ *keys r* = {}
shows *red F* (*p* + *r*) (*q* + *r*)
⟨*proof*⟩

lemma *red-plus*:
assumes *red F p q*
obtains *s* **where** (*red F*)** (*p* + *r*) *s* **and** (*red F*)** (*q* + *r*) *s*
⟨*proof*⟩

corollary *red-plus-cs*:
assumes *red F p q*
shows *relation.cs* (*red F*) (*p* + *r*) (*q* + *r*)
⟨*proof*⟩

lemma *red-uminus*:
assumes *red F p q*
shows *red F* (−*p*) (−*q*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

lemma *red-diff-rtrancl*:
assumes (*red F*)** (*p* − *q*) *0*
obtains *s* **where** (*red F*)** *p s* **and** (*red F*)** *q s*
⟨*proof*⟩

corollary *red-diff-rtrancl-cs*:
assumes (*red F*)** (*p* − *q*) *0*
shows *relation.cs* (*red F*) *p q*
⟨*proof*⟩

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* ⊕ *lt f1* $\prec_t$ *t2* ⊕ *lt f2* **and** *f1* ∈ *F* **and** *f2* ∈ *F*
obtains *s* **where** (*red F*)** *q1 s* **and** (*red F*)** *q2 s*

⟨*proof*⟩

lemma *confluent-distinct*:
assumes *r1*: *red-single p q1 f1 t1* **and** *r2*: *red-single p q2 f2 t2*
and *ne*: *t1* ⊕ *lt f1* ≠ *t2* ⊕ *lt f2* **and** *f1* ∈ *F* **and** *f2* ∈ *F*
obtains *s* **where** (*red F*)** *q1 s* **and** (*red F*)** *q2 s*
⟨*proof*⟩

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*⟩

4.4 Reducibility and Module Membership

lemma *srtc-in-pmdl*:
assumes *relation.srtc* (*red F*) *p q*
shows *p* − *q* ∈ *pmdl F*
⟨*proof*⟩

lemma *in-pmdl-srtc*:
assumes *p* ∈ *pmdl F*
shows *relation.srtc* (*red F*) *p 0*
⟨*proof*⟩

lemma *red-rtranclp-diff-in-pmdl*:
assumes (*red F*)** *p q*
shows *p* − *q* ∈ *pmdl F*
⟨*proof*⟩

corollary *red-diff-in-pmdl*:
assumes *red F p q*
shows *p* − *q* ∈ *pmdl F*
⟨*proof*⟩

corollary *red-rtranclp-0-in-pmdl*:
assumes (*red F*)** *p 0*
shows *p* ∈ *pmdl F*
⟨*proof*⟩

lemma *pmdl-closed-red*:
assumes *pmdl B* ⊆ *pmdl A* **and** *p* ∈ *pmdl A* **and** *red B p q*
shows *q* ∈ *pmdl A*
⟨*proof*⟩

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*⟩

corollary *red-rtrancl-uminus*:
assumes $(red\ F)^{**}\ p\ q$
shows $(red\ F)^{**}\ (-p)\ (-q)$
⟨*proof*⟩

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 $P\ p\ (q + r)$
⟨*proof*⟩

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*: $\bigwedge y\ z.$ [| $(red\ F)^{**}\ (p - q)\ y$; $red\ F\ y\ z$; $P\ p\ (y + q)$|] ==> $P\ p\ (z + q)$
shows $P\ p\ q$
⟨*proof*⟩

lemma *is-red-union*: $is\text{-}red\ (A \cup B)\ p \longleftrightarrow (is\text{-}red\ A\ p \lor is\text{-}red\ B\ p)$
⟨*proof*⟩

lemma *red-single-0-lt*:
assumes $red\text{-}single\ f\ 0\ h\ t$
shows $lt\ f = t \oplus lt\ h$
⟨*proof*⟩

lemma *red-single-lt-distinct-lt*:
assumes *rs*: $red\text{-}single\ f\ g\ h\ t$ **and** $g \neq 0$ **and** $lt\ g \neq lt\ f$
shows $lt\ f = t \oplus lt\ h$
⟨*proof*⟩

lemma *zero-reducibility-implies-lt-divisibility'*:
assumes $(red\ F)^{**}\ f\ 0$ **and** $f \neq 0$
shows $\exists h \in F.\ h \neq 0 \land (lt\ h\ adds_t\ lt\ f)$
⟨*proof*⟩

lemma *zero-reducibility-implies-lt-divisibility*:
assumes $(red\ F)^{**}\ f\ 0$ **and** $f \neq 0$
obtains h **where** $h \in F$ **and** $h \neq 0$ **and** $lt\ h\ adds_t\ lt\ f$
⟨*proof*⟩

lemma *is-red-addsI*:
assumes $f \in F$ **and** $f \neq 0$ **and** $v \in keys\ p$ **and** $lt\ f\ adds_t\ v$
shows $is\text{-}red\ F\ p$
⟨*proof*⟩

lemma *is-red-addsE′*:
  assumes *is-red F p*
  shows $\exists f \in F.\ \exists v \in keys\ p.\ f \neq 0 \wedge lt\ f\ adds_t\ v$
  ⟨*proof*⟩

lemma *is-red-addsE*:
  assumes *is-red F p*
  obtains *f v* **where** $f \in F$ **and** $v \in keys\ p$ **and** $f \neq 0$ **and** $lt\ f\ adds_t\ v$
  ⟨*proof*⟩

lemma *is-red-adds-iff*:
  shows $(is\text{-}red\ F\ p) \longleftrightarrow (\exists f \in F.\ \exists v \in keys\ p.\ f \neq 0 \wedge lt\ f\ adds_t\ v)$
  ⟨*proof*⟩

lemma *is-red-subset*:
  assumes *red*: *is-red A p* **and** *sub*: $A \subseteq B$
  shows *is-red B p*
⟨*proof*⟩

lemma *not-is-red-empty*: $\neg$ *is-red* {} *f*
  ⟨*proof*⟩

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*⟩

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 $(red\ F)^{**}\ (p + r)\ (q + r)$
  ⟨*proof*⟩

lemma *red-mult-scalar-leading-monomial*: $(red\ \{f\})^{**}\ (p \odot monomial\ (lc\ f)\ (lt\ f))$ $(-\ p \odot tail\ f)$
⟨*proof*⟩

corollary *red-mult-scalar-lt*:
  assumes $f \neq 0$
  shows $(red\ \{f\})^{**}\ (p \odot monomial\ c\ (lt\ f))\ (monom\text{-}mult\ (-\ c\ /\ lc\ f)\ 0\ (p \odot tail\ f))$
⟨*proof*⟩

lemma *is-red-monomial-iff*: $is\text{-}red\ F\ (monomial\ c\ v) \longleftrightarrow (c \neq 0 \wedge (\exists f \in F.\ f \neq 0 \wedge lt\ f\ adds_t\ v))$
  ⟨*proof*⟩

lemma *is-red-monomialI*:
  assumes $c \neq 0$ **and** $f \in F$ **and** $f \neq 0$ **and** $lt\ f\ adds_t\ v$
  shows *is-red F* (*monomial c v*)

⟨*proof*⟩

lemma *is-red-monomialD*:
assumes *is-red* F (*monomial* c v)
shows $c \neq 0$
⟨*proof*⟩

lemma *is-red-monomialE*:
assumes *is-red* F (*monomial* c v)
obtains f **where** $f \in F$ **and** $f \neq 0$ **and** *lt* f $adds_t$ v
⟨*proof*⟩

lemma *replace-lt-adds-stable-is-red*:
assumes *red*: *is-red* F f **and** $q \neq 0$ **and** *lt* q $adds_t$ *lt* p
shows *is-red* (*insert* q ($F - \{p\}$)) f
⟨*proof*⟩

lemma *conversion-property*:
assumes *is-red* $\{p\}$ f **and** *red* $\{r\}$ p q
shows *is-red* $\{q\}$ f $\vee$ *is-red* $\{r\}$ f
⟨*proof*⟩

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*⟩

lemma *is-red-map-scale*:
assumes *is-red* F ($c \cdot p$)
shows *is-red* F p
⟨*proof*⟩

corollary *is-irred-map-scale*: $\neg$ *is-red* F p $\Longrightarrow$ $\neg$ *is-red* F ($c \cdot p$)
⟨*proof*⟩

lemma *is-red-map-scale-iff*: *is-red* F ($c \cdot p$) $\longleftrightarrow$ ($c \neq 0$ $\wedge$ *is-red* F p)
⟨*proof*⟩

lemma *is-red-uminus*: *is-red* F ($- p$) $\longleftrightarrow$ *is-red* F p
⟨*proof*⟩

lemma *is-red-plus*:
assumes *is-red* F ($p + q$)
shows *is-red* F p $\vee$ *is-red* F q
⟨*proof*⟩

lemma *is-irred-plus*: $\neg$ *is-red* F p $\Longrightarrow$ $\neg$ *is-red* F q $\Longrightarrow$ $\neg$ *is-red* F ($p + q$)
⟨*proof*⟩

lemma *is-red-minus*:
  assumes *is-red F* $(p - q)$
  shows *is-red F p* $\vee$ *is-red F q*
⟨*proof*⟩

lemma *is-irred-minus*: $\neg$ *is-red F p* $\Longrightarrow$ $\neg$ *is-red F q* $\Longrightarrow$ $\neg$ *is-red F* $(p - q)$
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

corollary *dgrad-p-set-le-red-rtrancl*:
  assumes *dickson-grading d* **and** $(\mathit{red}\ F)^{**}\ p\ q$
  shows *dgrad-p-set-le d* $\{q\}$ (*insert p F*)
  ⟨*proof*⟩

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 $d\ t \leq m$
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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 $\bigwedge g.\ lt\ (q\ g \odot g) \preceq_t lt\ p$
⟨*proof*⟩

lemma *is-relation-order-red*:
assumes *dickson-grading d*
shows *Confluence.relation-order* (*red F*) $(\prec_p)$ (*dgrad-p-set d m*)
⟨*proof*⟩

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*⟩

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*⟩

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** $d\ t \le m$ **and** $c \neq 0$
and *cbelow-on* (*dgrad-p-set d m*) $(\prec_p)$ *z* $(\lambda a\ b.\ red\ F\ a\ b \lor red\ F\ b\ a)$ *p q*
shows *cbelow-on* (*dgrad-p-set d m*) $(\prec_p)$ (*monom-mult c t z*) $(\lambda a\ b.\ red\ F\ a\ b \lor red\ F\ b\ a)$
(*monom-mult c t p*) (*monom-mult c t q*)
⟨*proof*⟩

lemma *cbelow-on-monom-mult-monomial*:
assumes $c \neq 0$
and *cbelow-on* (*dgrad-p-set d m*) $(\prec_p)$ (*monomial* c' *v*) $(\lambda a\ b.\ red\ F\ a\ b \lor red\ F\ b\ a)$ *p q*
shows *cbelow-on* (*dgrad-p-set d m*) $(\prec_p)$ (*monomial c* $(t \oplus v)$) $(\lambda a\ b.\ red\ F\ a\ b \lor red\ F\ b\ a)$ *p q*

⟨*proof*⟩

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* ($\lambda a\ b.$ *red F a b* $\vee$ *red F b a*) *p q*
shows *cbelow-on* (*dgrad-p-set d m*) ($\prec_p$) ($z + r$) ($\lambda a\ b.$ *red F a b* $\vee$ *red F b a*) ($p + r$) ($q + r$)
⟨*proof*⟩

lemma *is-full-pmdlI-lt-dgrad-p-set*:
assumes *dickson-grading d* **and** $B \subseteq$ *dgrad-p-set d m*
assumes $\bigwedge k.\ k \in$ *component-of-term* ‘ *Keys* (B::('t $\Rightarrow_0$ 'b::*field*) *set*) $\Longrightarrow$
($\exists\, b \in B.\ b \neq 0 \wedge$ *component-of-term* (*lt b*) = $k \wedge$ *lp b* = *0*)
shows *is-full-pmdl B*
⟨*proof*⟩

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 $\Rightarrow_0$ 'b) *list* $\Rightarrow$ 't $\Rightarrow$ ('t $\Rightarrow_0$ 'b::*zero*) *option* **where**
find-adds [] - = *None*|
find-adds (f # *fs*) *u* = (*if f* $\neq$ *0* $\wedge$ *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 $f \in$ *set fs*
⟨*proof*⟩

lemma *find-adds-SomeD2*:
assumes *find-adds fs u* = *Some f*
shows $f \neq 0$
⟨*proof*⟩

lemma *find-adds-SomeD3*:
assumes *find-adds fs u* = *Some f*
shows *lt f adds*$_t$ *u*
⟨*proof*⟩

lemma *find-adds-NoneE*:
assumes *find-adds fs u* = *None* **and** $f \in$ *set fs*

assumes $f = 0 \Longrightarrow thesis$ **and** $f \neq 0 \Longrightarrow \neg\ lt\ f\ adds_t\ u \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

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*⟩

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*⟩

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-term d* = $\{(x, y).$ *dgrad-p-set-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*⟩

function *trd-aux* :: $('t \Rightarrow_0 'b)\ list \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b{::}field)$
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*
)
⟨*proof*⟩
termination ⟨*proof*⟩

definition *trd* :: $('t \Rightarrow_0 'b{::}field)\ list \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b)$
where *trd fs p* = *trd-aux fs p 0*

lemma *trd-aux-red-rtrancl*: $(red\ (set\ fs))^{**}\ p\ (trd\text{-}aux\ fs\ p\ r - r)$
⟨*proof*⟩

corollary *trd-red-rtrancl*: $(red\ (set\ fs))^{**}\ p\ (trd\ fs\ p)$
⟨*proof*⟩

lemma *trd-aux-irred*:
assumes $\neg$ *is-red* (*set fs*) *r*
shows $\neg$ *is-red* (*set fs*) (*trd-aux fs p r*)
⟨*proof*⟩

corollary *trd-irred*: $\neg$ *is-red* (*set fs*) (*trd fs p*)
⟨*proof*⟩

lemma *trd-in-pmdl*: $p - (trd\ fs\ p) \in pmdl\ (set\ fs)$
⟨*proof*⟩

lemma *pmdl-closed-trd*:
assumes $p \in pmdl\ B$ **and** $set\ fs \subseteq pmdl\ B$
shows $(trd\ fs\ p) \in pmdl\ B$
⟨*proof*⟩

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*) ($\prec_p$)

$(monomial\ 1\ (term\text{-}of\text{-}pair\ (lcs\ (lp\ p)\ (lp\ q),\ component\text{-}of\text{-}term\ (lt\ p))))$
$(\lambda a\ b.\ red\ F\ a\ b \vee red\ F\ b\ a)\ (fst\ (crit\text{-}pair\ p\ q))\ (snd\ (crit\text{-}pair\ p\ q))$

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* :: $('t \Rightarrow_0 'b{::}field)\ set \Rightarrow bool$
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*)
⟨*proof*⟩

lemma *crit-pair-swap*: *crit-pair p q* = (*snd* (*crit-pair q p*), *fst* (*crit-pair q p*))
⟨*proof*⟩

lemma *crit-pair-zero* [*simp*]: *fst* (*crit-pair 0 q*) = *0* **and** *snd* (*crit-pair p 0*) = *0*
⟨*proof*⟩

lemma *dgrad-p-set-le-crit-pair-zero*: *dgrad-p-set-le d* {*fst* (*crit-pair p 0*)} {*p*}
⟨*proof*⟩

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*⟩

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*}
⟨*proof*⟩

lemma *dgrad-p-set-closed-fst-crit-pair*:
assumes *dickson-grading d* **and** *p* ∈ *dgrad-p-set d m* **and** *q* ∈ *dgrad-p-set d m*
shows *fst* (*crit-pair p q*) ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-snd-crit-pair*:
assumes *dickson-grading d* **and** *p* ∈ *dgrad-p-set d m* **and** *q* ∈ *dgrad-p-set d m*
shows *snd* (*crit-pair p q*) ∈ *dgrad-p-set d m*
⟨*proof*⟩

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*⟩

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*⟩

lemma *crit-pair-cbelow-same*:
assumes *dickson-grading d* **and** *p* ∈ *dgrad-p-set d m*
shows *crit-pair-cbelow-on d m F p p*
⟨*proof*⟩

lemma *crit-pair-cbelow-distinct-component*:
assumes *component-of-term* (*lt p*) ≠ *component-of-term* (*lt q*)
shows *crit-pair-cbelow-on d m F p q*
⟨*proof*⟩

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*⟩

lemma *crit-pair-cs-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 *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*⟩

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*
⟨*proof*⟩

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*⟩

corollary *lcs-red-single-snd-crit-pair*:
assumes *q* ≠ *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*)
⟨*proof*⟩

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*⟩

lemma *spoly-alt*:
assumes *p* ≠ *0* **and** *q* ≠ *0*
shows *spoly p q* = *fst* (*crit-pair p q*) − *snd* (*crit-pair p q*)
⟨*proof*⟩

lemma *spoly-same*: *spoly p p* = *0*
⟨*proof*⟩

lemma *spoly-swap*: *spoly p q* = − *spoly q p*
⟨*proof*⟩

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*⟩

lemma *dgrad-p-set-le-spoly-zero*: *dgrad-p-set-le d* {*spoly p 0*} {*p*}
⟨*proof*⟩

lemma *dgrad-p-set-le-spoly*:
assumes *dickson-grading d*
shows *dgrad-p-set-le d* {*spoly p q*} {*p*, *q*}
⟨*proof*⟩

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*⟩

lemma *components-spoly-subset*: *component-of-term* ' *keys* (*spoly p q*) ⊆ *component-of-term* ' *Keys* {*p*, *q*}
⟨*proof*⟩

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

shows *spoly p q* $\in$ *pmdl F*
$\langle proof \rangle$

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*)
$\langle proof \rangle$

corollary *crit-pair-cbelow-imp-GB-dgrad-p-set*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-p-set d m*
assumes $\bigwedge p\ q.\ p \in F \Longrightarrow q \in F \Longrightarrow p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow$ *crit-pair-cbelow-on d m F p q*
shows *is-Groebner-basis F*
$\langle proof \rangle$

corollary *Buchberger-criterion-dgrad-p-set*:
assumes *dickson-grading d* **and** $F \subseteq$ *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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *is-Groebner-basis-empty*: *is-Groebner-basis* {}

⟨*proof*⟩

lemma *is-Groebner-basis-singleton*: *is-Groebner-basis* $\{f\}$
⟨*proof*⟩

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*⟩

lemma *chain-criterion*:
assumes *dickson-grading* d **and** $F \subseteq$ *dgrad-p-set* d m **and** $p \in F$ **and** $q \in F$
and $p \neq 0$ **and** $q \neq 0$ **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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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** *pmdl G* $\subseteq$ *pmdl B*
assumes $\bigwedge f.\ f \in$ *pmdl B* $\Longrightarrow f \neq 0 \Longrightarrow (\exists\, g \in G.\ g \neq 0 \wedge$ *lt g adds*$_t$ *lt f*$)$
shows *pmdl G* $=$ *pmdl B*
⟨*proof*⟩

lemma *GB-implies-unique-nf-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
assumes *isGB*: *is-Groebner-basis G*
shows $\exists!\ h.$ $($*red G*$)^{**}$ *f h* $\wedge \neg$ *is-red G h*
⟨*proof*⟩

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*⟩

lemma *translation-property*:
assumes $p \neq q$ **and** *red-0*: $($*red F*$)^{**}$ $(p - q)$ 0
shows *is-red F p* $\vee$ *is-red F q*
⟨*proof*⟩

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-p-set d m* $\Longrightarrow ($*red G*$)^{**}$ *f 0*
shows *is-Groebner-basis G*
⟨*proof*⟩

lemma *weak-GB-is-strong-GB*:
assumes $\bigwedge f.\ f \in ($*pmdl G*$) \Longrightarrow ($*red G*$)^{**}$ *f 0*
shows *is-Groebner-basis G*
⟨*proof*⟩

corollary *GB-alt-1-dgrad-p-set*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
shows *is-Groebner-basis G* $\longleftrightarrow (\forall f \in$ *pmdl G*. $f \in$ *dgrad-p-set d m* $\longrightarrow ($*red G*$)^{**}$ *f 0*$)$
⟨*proof*⟩

corollary *GB-alt-1*: *is-Groebner-basis G* $\longleftrightarrow (\forall f \in$ *pmdl G*. $($*red G*$)^{**}$ *f 0*$)$
⟨*proof*⟩

lemma *isGB-I-is-red*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
  assumes $\bigwedge f.\ f \in$ *pmdl G* $\Longrightarrow f \in$ *dgrad-p-set d m* $\Longrightarrow f \neq 0 \Longrightarrow$ *is-red G f*
  shows *is-Groebner-basis G*
  ⟨*proof*⟩

lemma *GB-alt-2-dgrad-p-set*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
  shows *is-Groebner-basis G* $\longleftrightarrow (\forall f \in$ *pmdl G*. $f \neq 0 \longrightarrow$ *is-red G f*)
⟨*proof*⟩

lemma *GB-adds-lt*:
  assumes *is-Groebner-basis G* **and** $f \in$ *pmdl G* **and** $f \neq 0$
  obtains *g* **where** $g \in G$ **and** $g \neq 0$ **and** *lt g* $adds_t$ *lt f*
⟨*proof*⟩

lemma *isGB-I-adds-lt*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
  assumes $\bigwedge f.\ f \in$ *pmdl G* $\Longrightarrow f \in$ *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 *is-Groebner-basis G*
  ⟨*proof*⟩

lemma *GB-alt-3-dgrad-p-set*:
  assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-p-set d m*
  shows *is-Groebner-basis G* $\longleftrightarrow (\forall f \in$ *pmdl G*. $f \neq 0 \longrightarrow (\exists g \in G.\ g \neq 0 \wedge$ *lt g* $adds_t$ *lt f*))
    (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *GB-insert*:
  assumes *is-Groebner-basis G* **and** $f \in$ *pmdl G*
  shows *is-Groebner-basis* (*insert f G*)
  ⟨*proof*⟩

lemma *GB-subset*:
  assumes *is-Groebner-basis G* **and** $G \subseteq G'$ **and** *pmdl* G' = *pmdl G*
  shows *is-Groebner-basis* G'
  ⟨*proof*⟩

lemma (**in** *ordered-term*) *GB-remove-0-stable-GB*:
  assumes *is-Groebner-basis G*
  shows *is-Groebner-basis* $(G - \{0\})$
  ⟨*proof*⟩

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 dick-*

son-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 $spoly\text{-}rep\ d\ m\ G\ g1\ g2 \longleftrightarrow (\exists\, q.\ spoly\ g1\ g2 = (\sum g{\in}G.\ q\ g \odot g) \wedge$
  $(\forall\, g.\ q\ g \in punit.dgrad\text{-}p\text{-}set\ d\ m \wedge$
  $(q\ g \odot g \neq 0 \longrightarrow lt\ (q\ g \odot g) \prec_t term\text{-}of\text{-}pair\ (lcs\ (lp\ g1)\ (lp\ g2),$
  $component\text{-}of\text{-}term\ (lt\ g2))))))$

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

lemma *spoly-repI-zero*:
  assumes $spoly\ g1\ g2 = 0$
  shows $spoly\text{-}rep\ d\ m\ G\ g1\ g2$
⟨*proof*⟩

lemma *spoly-repE*:
  assumes $spoly\text{-}rep\ d\ m\ G\ g1\ g2$
  obtains *q* **where** $spoly\ g1\ g2 = (\sum g{\in}G.\ q\ g \odot g)$ **and** $\bigwedge g.\ q\ g \in punit.dgrad\text{-}p\text{-}set\ d\ m$
  and $\bigwedge g.\ q\ g \odot g \neq 0 \Longrightarrow lt\ (q\ g \odot g) \prec_t term\text{-}of\text{-}pair\ (lcs\ (lp\ g1)\ (lp\ g2),$
  $component\text{-}of\text{-}term\ (lt\ g2))$
  ⟨*proof*⟩

corollary *isGB-D-spoly-rep*:
  assumes *dickson-grading d* **and** *is-Groebner-basis G* **and** $G \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** *finite G*
  and $g1 \in G$ **and** $g2 \in G$ **and** $g1 \neq 0$ **and** $g2 \neq 0$
  shows $spoly\text{-}rep\ d\ m\ G\ g1\ g2$
⟨*proof*⟩

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*⟩

5.6 Replacing Elements in Gröbner Bases

lemma *replace-in-dgrad-p-set*:
assumes $G \subseteq$ *dgrad-p-set d m*
obtains *n* **where** $q \in$ *dgrad-p-set d n* **and** $G \subseteq$ *dgrad-p-set d n*
and *insert q* $(G - \{p\}) \subseteq$ *dgrad-p-set d n*
⟨*proof*⟩

lemma *GB-replace-lt-adds-stable-GB-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*: $q \in$ (*pmdl G*) **and** *lt q* $adds_t$ *lt p*
shows *is-Groebner-basis* (*insert q* $(G - \{p\})$) (**is** *is-Groebner-basis ?G′*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *GB-replace-red-rtranclp-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** *ptoq*: $(red\ (G - \{p\}))^{**}$ *p q*
shows *is-Groebner-basis* (*insert q* $(G - \{p\})$)
⟨*proof*⟩

lemma *GB-replace-red-rtranclp-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 $(G - \{p\})) = pmdl\ G$
⟨*proof*⟩

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** $F \subseteq$ *dgrad-p-set* d m
obtains G **where** $G \subseteq$ *dgrad-p-set* d m **and** *finite* G **and** *is-Groebner-basis* G **and** *pmdl* G = *pmdl* F
⟨*proof*⟩

The preceding lemma justifies the following definition.

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

lemma *some-GB-props-dgrad-p-set*:
assumes *dickson-grading* d **and** *finite* (*component-of-term* ‘ *Keys* F) **and** $F \subseteq$ *dgrad-p-set* d m
shows *finite* (*some-GB* F) $\wedge$ *is-Groebner-basis* (*some-GB* F) $\wedge$ *pmdl* (*some-GB* F) = *pmdl* F
⟨*proof*⟩

lemma *finite-some-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* (*some-GB* F)
⟨*proof*⟩

lemma *some-GB-isGB-dgrad-p-set*:

assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *is-Groebner-basis* (*some-GB F*)
⟨*proof*⟩

lemma *some-GB-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *pmdl* (*some-GB F*) = *pmdl F*
⟨*proof*⟩

lemma *finite-imp-finite-component-Keys*:
assumes *finite F*
shows *finite* (*component-of-term* ‘ *Keys F*)
⟨*proof*⟩

lemma *finite-some-GB-finite*: *finite F* $\Longrightarrow$ *finite* (*some-GB F*)
⟨*proof*⟩

lemma *some-GB-isGB-finite*: *finite F* $\Longrightarrow$ *is-Groebner-basis* (*some-GB F*)
⟨*proof*⟩

lemma *some-GB-pmdl-finite*: *finite F* $\Longrightarrow$ *pmdl* (*some-GB F*) = *pmdl F*
⟨*proof*⟩

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* $\Rightarrow$ ('*t* $\Rightarrow_0$ '*b*) *set* $\Rightarrow$ *bool* (**infixl** ‹$\sqsupset p$› *50*)
where *red-supset A B* $\equiv$ ($\exists$ *p*. *is-red A p* $\wedge$ $\neg$ *is-red B p*) $\wedge$ ($\forall$ *p*. *is-red B p* $\longrightarrow$ *is-red A p*)

lemma *red-supsetE*:
assumes $A \sqsupset p\ B$
obtains *p* **where** *is-red A p* **and** $\neg$ *is-red B p*
⟨*proof*⟩

lemma *red-supsetD*:
assumes *a1*: $A \sqsupset p\ B$ **and** *a2*: *is-red B p*
shows *is-red A p*
⟨*proof*⟩

lemma *red-supsetI* [*intro*]:

assumes $\bigwedge q.$ *is-red B q* $\Longrightarrow$ *is-red A q* **and** *is-red A p* **and** $\neg$ *is-red B p*
shows $A \sqsupset p\ B$
⟨*proof*⟩

lemma *red-supset-insertI*:
assumes $x \neq 0$ **and** $\neg$ *is-red A x*
shows (*insert x A*) $\sqsupset p\ A$
⟨*proof*⟩

lemma *red-supset-transitive*:
assumes $A \sqsupset p\ B$ **and** $B \sqsupset p\ C$
shows $A \sqsupset p\ C$
⟨*proof*⟩

lemma *red-supset-wf-on*:
assumes *dickson-grading d* **and** *finite K*
shows *wfp-on* ($\sqsupset p$) (*Pow* (*dgrad-p-set d m*) $\cap$ {*F. component-of-term* ' *Keys F* $\subseteq$ *K*})
⟨*proof*⟩

end

lemma *in-lex-prod-alt*:
$(x, y) \in r <*lex*> s \longleftrightarrow (((fst\ x), (fst\ y)) \in r \lor (fst\ x = fst\ y \land ((snd\ x), (snd\ y)) \in s))$
⟨*proof*⟩

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* $A\ B = A - (B \cup \mathit{prod.swap}\ `\ B)$
definition *Int-pairs* (**infixl** ‹$\cap_p$› *65*) **where** *Int-pairs* $A\ B = A \cap (B \cup \mathit{prod.swap}\ `\ B)$
definition *in-pair* (**infix** ‹$\in_p$› *50*) **where** *in-pair* $p\ A \longleftrightarrow (p \in A \cup \mathit{prod.swap}\ `\ A)$
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*: $p \in_p A \longleftrightarrow (p \in A \lor \mathit{prod.swap}\ p \in A)$
⟨*proof*⟩

lemma *in-pair-iff*: $(a,\ b) \in_p A \longleftrightarrow ((a,\ b) \in A \lor (b,\ a) \in A)$
⟨*proof*⟩

lemma *in-pair-minus-pairs* [*simp*]: $p \in_p A -_p B \longleftrightarrow (p \in_p A \land p \notin_p B)$
⟨*proof*⟩

lemma *in-minus-pairs* [*simp*]: $p \in A -_p B \longleftrightarrow (p \in A \land p \notin_p B)$
⟨*proof*⟩

lemma *in-pair-Int-pairs* [*simp*]: $p \in_p A \cap_p B \longleftrightarrow (p \in_p A \land p \in_p B)$
⟨*proof*⟩

lemma *in-pair-Un* [*simp*]: $p \in_p A \cup B \longleftrightarrow (p \in_p A \lor p \in_p B)$
⟨*proof*⟩

lemma *in-pair-trans* [*trans*]:
assumes $p \in_p A$ **and** $A \subseteq B$
shows $p \in_p B$
⟨*proof*⟩

lemma *in-pair-same* [*simp*]: $p \in_p A \times A \longleftrightarrow p \in A \times A$
⟨*proof*⟩

lemma *subset-pairsI* [*intro*]:
assumes $\bigwedge x.\ x \in_p A \Longrightarrow x \in_p B$
shows $A \subseteq_p B$
⟨*proof*⟩

lemma *subset-pairsD* [*trans*]:

assumes $x \in_p A$ **and** $A \subseteq_p B$
shows $x \in_p B$
$\langle proof \rangle$

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 $(a, b)\ xs\ ps \longleftrightarrow ((a \in set\ xs) \wedge (b \in set\ xs) \wedge (a, b) \notin_p set\ ps)$
$\langle proof \rangle$

lemma *processedI*:
assumes $a \in set\ xs$ **and** $b \in set\ xs$ **and** $(a, b) \notin_p set\ ps$
shows *processed* $(a, b)\ xs\ ps$
$\langle proof \rangle$

lemma *processedD1*:
assumes *processed* $(a, b)\ xs\ ps$
shows $a \in set\ xs$
$\langle proof \rangle$

lemma *processedD2*:
assumes *processed* $(a, b)\ xs\ ps$
shows $b \in set\ xs$
$\langle proof \rangle$

lemma *processedD3*:
assumes *processed* $(a, b)\ xs\ ps$
shows $(a, b) \notin_p set\ ps$
$\langle proof \rangle$

lemma *processed-Nil*: *processed* $(a, b)\ xs\ [] \longleftrightarrow (a \in set\ xs \wedge b \in set\ xs)$
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *processed-minus*:
assumes *processed* $(a, b)\ xs\ (ps -- qs)$
and *a1*: $(a, b) \in_p set\ qs \Longrightarrow thesis$
and *a2*: *processed* $(a, b)\ xs\ ps \Longrightarrow thesis$
shows *thesis*
$\langle proof \rangle$

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*
⟨*proof*⟩

lemma *const-lt-component-SomeD1*:
assumes *const-lt-component p* = *Some cmp*
shows *lp p* = *0*
⟨*proof*⟩

lemma *const-lt-component-SomeD2*:
assumes *const-lt-component p* = *Some cmp*
shows *component-of-term* (*lt p*) = *cmp*
⟨*proof*⟩

lemma *const-lt-component-subset*:
const-lt-component ‘ ($B - \{0\}$) $-$ {*None*} $\subseteq$ *Some* ‘ *component-of-term* ‘ *Keys B*
⟨*proof*⟩

corollary *card-const-lt-component-le*:
assumes *finite B*
shows *card* (*const-lt-component* ‘ ($B - \{0\}$) $-$ {*None*}) $\leq$ *card* (*component-of-term* ‘ *Keys B*)
⟨*proof*⟩

end

6.2.2 Type synonyms

type-synonym ('*a*, '*b*, '*c*) *pdata'* = ('*a* $\Rightarrow_0$ '*b*) × '*c*
type-synonym ('*a*, '*b*, '*c*) *pdata* = ('*a* $\Rightarrow_0$ '*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*
  ⟨*proof*⟩

lemma *sel-specD1*:
  assumes *sel-spec sel* **and** *ps* ≠ []
  shows *sel gs bs ps data* ≠ []
  ⟨*proof*⟩

lemma *sel-specD2*:
  assumes *sel-spec sel* **and** *ps* ≠ []
  shows *set* (*sel gs bs ps data*) ⊆ *set ps*
  ⟨*proof*⟩

6.2.4 Specification of the *add-basis* parameter

definition *ab-spec* :: ('a, 'b, 'c, 'd) *abT* ⇒ *bool*
  where *ab-spec ab* ⟷
    (∀ *gs bs ns data*. *ns* ≠ [] ⟶ *set* (*ab gs bs ns data*) = *set bs* ∪ *set ns*) ∧
    (∀ *gs bs data*. *ab gs bs* [] *data* = *bs*)

lemma *ab-specI*:
  assumes ⋀*gs bs ns data*. *ns* ≠ [] ⟹ *set* (*ab gs bs ns data*) = *set bs* ∪ *set ns*
    and ⋀*gs bs data*. *ab gs bs* [] *data* = *bs*
  shows *ab-spec ab*
  ⟨*proof*⟩

lemma *ab-specD1*:

assumes *ab-spec ab*
shows *set* (*ab gs bs ns data*) = *set bs* ∪ *set ns*
⟨*proof*⟩

lemma *ab-specD2*:
assumes *ab-spec ab*
shows *ab gs bs* [] *data* = *bs*
⟨*proof*⟩

6.2.5 Specification of the *add-pairs* parameter

definition *unique-idx* :: ('*t*, '*b*, '*c*) *pdata list* ⇒ (*nat* × '*d*) ⇒ *bool*
where *unique-idx bs data* ⟷
(∀ *f*∈*set bs*. ∀ *g*∈*set bs*. *fst* (*snd f*) = *fst* (*snd g*) ⟶ *f* = *g*) ∧
(∀ *f*∈*set bs*. *fst* (*snd f*) < *fst data*)

lemma *unique-idxI*:
assumes ⋀*f g*. *f* ∈ *set bs* ⟹ *g* ∈ *set bs* ⟹ *fst* (*snd f*) = *fst* (*snd g*) ⟹ *f* = *g*
and ⋀*f*. *f* ∈ *set bs* ⟹ *fst* (*snd f*) < *fst data*
shows *unique-idx bs data*
⟨*proof*⟩

lemma *unique-idxD1*:
assumes *unique-idx bs data* **and** *f* ∈ *set bs* **and** *g* ∈ *set bs* **and** *fst* (*snd f*) = *fst* (*snd g*)
shows *f* = *g*
⟨*proof*⟩

lemma *unique-idxD2*:
assumes *unique-idx bs data* **and** *f* ∈ *set bs*
shows *fst* (*snd f*) < *fst data*
⟨*proof*⟩

lemma *unique-idx-Nil*: *unique-idx* [] *data*
⟨*proof*⟩

lemma *unique-idx-subset*:
assumes *unique-idx bs data* **and** *set bs'* ⊆ *set bs*
shows *unique-idx bs' data*
⟨*proof*⟩

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* ⟶

$set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \longrightarrow$
$is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs) \longrightarrow h \neq g \longrightarrow fst\ h \neq 0 \longrightarrow fst\ g \neq 0 \longrightarrow$
$(\forall\, a\ b.\ (a,\ b) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data) \longrightarrow fst\ a \neq 0 \longrightarrow fst\ b \neq 0 \longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)) \longrightarrow$
$(\forall\, a\ b.\ a \in set\ gs \cup set\ bs \longrightarrow b \in set\ gs \cup set\ bs \longrightarrow fst\ a \neq 0 \longrightarrow fst\ b \neq 0 \longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)) \longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)) \wedge$
$(\forall\, B\ d\ m.\ \forall\, h\ g.\ dickson\text{-}grading\ d \longrightarrow$
$set\ gs \cup set\ bs \cup set\ hs \subseteq B \longrightarrow fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m \longrightarrow$
$set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow (set\ gs \cup set\ bs) \cap set\ hs = \{\} \longrightarrow$
$unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \longrightarrow is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs) \longrightarrow$
$h \neq g \longrightarrow fst\ h \neq 0 \longrightarrow fst\ g \neq 0 \longrightarrow$
$(h,\ g) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data) \longrightarrow$
$(\forall\, a\ b.\ (a,\ b) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data) \longrightarrow (a,\ b) \in_p set\ hs \times (set\ gs \cup set\ bs \cup set\ hs) \longrightarrow$
$fst\ a \neq 0 \longrightarrow fst\ b \neq 0 \longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)) \longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)))$

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 $\bigwedge gs\ bs\ ps\ hs\ data.\ set\ (ap\ gs\ bs\ ps\ hs\ data) \subseteq set\ ps \cup (set\ hs \times (set\ gs \cup set\ bs \cup set\ hs))$
assumes $\bigwedge gs\ bs\ ps\ hs\ data\ B\ d\ m\ h\ g.\ dickson\text{-}grading\ d \Longrightarrow$
$set\ gs \cup set\ bs \cup set\ hs \subseteq B \Longrightarrow fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow$
$h \in set\ hs \Longrightarrow g \in set\ gs \cup set\ bs \cup set\ hs \Longrightarrow$
$set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \Longrightarrow unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \Longrightarrow$
$is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs) \Longrightarrow h \neq g \Longrightarrow fst\ h \neq 0 \Longrightarrow fst\ g \neq 0 \Longrightarrow$
$(\bigwedge a\ b.\ (a,\ b) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data) \Longrightarrow fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)) \Longrightarrow$
$(\bigwedge a\ b.\ a \in set\ gs \cup set\ bs \Longrightarrow b \in set\ gs \cup set\ bs \Longrightarrow fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)) \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)$
assumes $\bigwedge gs\ bs\ ps\ hs\ data\ B\ d\ m\ h\ g.\ dickson\text{-}grading\ d \Longrightarrow$
$set\ gs \cup set\ bs \cup set\ hs \subseteq B \Longrightarrow fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow$
$set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \Longrightarrow (set\ gs \cup set\ bs) \cap set\ hs = \{\} \Longrightarrow$
$unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \Longrightarrow is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs) \Longrightarrow h \neq g \Longrightarrow$

$fst\ h \neq 0 \Longrightarrow fst\ g \neq 0 \Longrightarrow (h,\ g) \in set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data)$
$\Longrightarrow$
$(\bigwedge a\ b.\ (a,\ b) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data) \Longrightarrow (a,\ b) \in_p set\ hs \times (set$
$gs \cup set\ bs \cup set\ hs) \Longrightarrow$
$fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst$
$a)\ (fst\ b)) \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)$
shows *ap-spec ap*
⟨*proof*⟩

lemma *ap-specD1*:
assumes *ap-spec ap*
shows $set\ (ap\ gs\ bs\ ps\ hs\ data) \subseteq set\ ps \cup (set\ hs \times (set\ gs \cup set\ bs \cup set\ hs))$
⟨*proof*⟩

lemma *ap-specD2*:
assumes *ap-spec ap* **and** *dickson-grading d* **and** $set\ gs \cup set\ bs \cup set\ hs \subseteq B$
and $fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** $(h,\ g) \in_p set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$
and $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs)$ **and** $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data$
and $is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs)$ **and** $h \neq g$ **and** $fst\ h \neq 0$ **and** $fst\ g \neq 0$
and $\bigwedge a\ b.\ (a,\ b) \in_p set\ (ap\ gs\ bs\ ps\ hs\ data) \Longrightarrow fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)$
and $\bigwedge a\ b.\ a \in set\ gs \cup set\ bs \Longrightarrow b \in set\ gs \cup set\ bs \Longrightarrow fst\ a \neq 0 \Longrightarrow fst\ b$
$\neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)$
⟨*proof*⟩

lemma *ap-specD3*:
assumes *ap-spec ap* **and** *dickson-grading d* **and** $set\ gs \cup set\ bs \cup set\ hs \subseteq B$
and $fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs)$
and $(set\ gs \cup set\ bs) \cap set\ hs = \{\}$ **and** $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data$
and $is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs)$ **and** $h \neq g$ **and** $fst\ h \neq 0$ **and** $fst\ g \neq 0$
and $(h,\ g) \in_p set\ ps -_p set\ (ap\ gs\ bs\ ps\ hs\ data)$
and $\bigwedge a\ b.\ a \in set\ hs \Longrightarrow b \in set\ gs \cup set\ bs \cup set\ hs \Longrightarrow (a,\ b) \in_p set\ (ap\ gs$
$bs\ ps\ hs\ data) \Longrightarrow$
$fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)$
$(fst\ b)$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ h)\ (fst\ g)$
⟨*proof*⟩

lemma *ap-spec-Nil-subset*:
assumes *ap-spec ap*
shows $set\ (ap\ gs\ bs\ ps\ []\ data) \subseteq set\ ps$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *args-to-set-subset1*:
assumes *set gs1* ⊆ *set gs2*
shows *args-to-set* (*gs1*, *bs*, *ps*) ⊆ *args-to-set* (*gs2*, *bs*, *ps*)
⟨*proof*⟩

lemma *args-to-set-subset2*:
assumes *set bs1* ⊆ *set bs2*
shows *args-to-set* (*gs*, *bs1*, *ps*) ⊆ *args-to-set* (*gs*, *bs2*, *ps*)
⟨*proof*⟩

lemma *args-to-set-subset3*:
assumes *set ps1* ⊆ *set ps2*
shows *args-to-set* (*gs*, *bs*, *ps1*) ⊆ *args-to-set* (*gs*, *bs*, *ps2*)
⟨*proof*⟩

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*})
⟨*proof*⟩

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*⟩

lemma *rem-comps-spec-count-rem-components*: *rem-comps-spec bs* (*count-rem-components bs*, *data*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Keys-full-gb*:
Keys (*fst* ' *set* (*full-gb bs*)) = (λ*v*. *term-of-pair* (*0*, *component-of-term v*)) ' *Keys* (*fst* ' *set bs*)
⟨*proof*⟩

lemma *pps-full-gb*: *pp-of-term* ' *Keys* (*fst* ' *set* (*full-gb bs*)) ⊆ {*0*}
⟨*proof*⟩

lemma *components-full-gb*:
component-of-term ' *Keys* (*fst* ' *set* (*full-gb bs*)) = *component-of-term* ' *Keys* (*fst* ' *set bs*)
⟨*proof*⟩

lemma *full-gb-is-full-pmdl*: *is-full-pmdl* (*fst* ' *set* (*full-gb bs*))
for *bs*::(′*t*, ′*b*::*field*, ′*c*::*default*) *pdata list*
⟨*proof*⟩

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*⟩

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 $\bigwedge$*gs bs ps sps data. sps* $\neq$ [] $\Longrightarrow$ *set sps* $\subseteq$ *set ps* $\Longrightarrow$
component-of-term ‘ *Keys* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) $\subseteq$
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
assumes $\bigwedge$*gs bs ps sps data. sps* $\neq$ [] $\Longrightarrow$ *set sps* $\subseteq$ *set ps* $\Longrightarrow$ *0* $\notin$ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
assumes $\bigwedge$*gs bs ps sps h b data. sps* $\neq$ [] $\Longrightarrow$ *set sps* $\subseteq$ *set ps* $\Longrightarrow$ *h* $\in$ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) $\Longrightarrow$
b $\in$ *set gs* $\cup$ *set bs* $\Longrightarrow$ *fst b* $\neq$ *0* $\Longrightarrow$ $\neg$ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
shows *compl-struct compl*
⟨*proof*⟩

lemma *compl-structD1*:
assumes *compl-struct compl* **and** *dickson-grading d* **and** *sps* $\neq$ [] **and** *set sps* $\subseteq$ *set ps*
shows *dgrad-p-set-le d* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) (*args-to-set* (*gs*, *bs*, *ps*))
⟨*proof*⟩

lemma *compl-structD2*:
assumes *compl-struct compl* **and** *sps* $\neq$ [] **and** *set sps* $\subseteq$ *set ps*
shows *component-of-term* ‘ *Keys* (*fst* ‘ (*set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)))) $\subseteq$
component-of-term ‘ *Keys* (*args-to-set* (*gs*, *bs*, *ps*))
⟨*proof*⟩

lemma *compl-structD3*:
assumes *compl-struct compl* **and** *sps* $\neq$ [] **and** *set sps* $\subseteq$ *set ps*
shows *0* $\notin$ *fst* ‘ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*))
⟨*proof*⟩

lemma *compl-structD4*:
assumes *compl-struct compl* **and** *sps* $\neq$ [] **and** *set sps* $\subseteq$ *set ps*
and *h* $\in$ *set* (*fst* (*compl gs bs* (*ps* −− *sps*) *sps data*)) **and** *b* $\in$ *set gs* $\cup$ *set bs* **and** *fst b* $\neq$ *0*
shows $\neg$ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
⟨*proof*⟩

definition *struct-spec* :: (*'t*, *'b::field*, *'c*, *'d*) *selT* $\Rightarrow$ (*'t*, *'b*, *'c*, *'d*) *apT* $\Rightarrow$ (*'t*, *'b*, *'c*, *'d*) *abT* $\Rightarrow$
(*'t*, *'b*, *'c*, *'d*) *complT* $\Rightarrow$ *bool*
where *struct-spec sel ap ab compl* $\longleftrightarrow$ (*sel-spec sel* $\wedge$ *ap-spec ap* $\wedge$ *ab-spec ab* $\wedge$ *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*
⟨*proof*⟩

lemma *struct-specD1*:
assumes *struct-spec sel ap ab compl*
shows *sel-spec sel*
⟨*proof*⟩

lemma *struct-specD2*:
assumes *struct-spec sel ap ab compl*
shows *ap-spec ap*
⟨*proof*⟩

lemma *struct-specD3*:
assumes *struct-spec sel ap ab compl*
shows *ab-spec ab*
⟨*proof*⟩

lemma *struct-specD4*:
assumes *struct-spec sel ap ab compl*
shows *compl-struct compl*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fst-add-indices*: *map fst* (*fst* (*add-indices ns data′*)) = *map fst* (*fst ns*)

⟨*proof*⟩

corollary *fst-set-add-indices*: *fst* ‘ *set* (*fst* (*add-indices ns data'*)) = *fst* ‘ *set* (*fst ns*)
⟨*proof*⟩

lemma *in-set-add-indicesE*:
assumes $f \in$ *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*⟩

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*⟩

lemma *gb-schema-aux-term-wf*:
assumes *dickson-grading d*
shows *wf* (*gb-schema-aux-term d gs*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *unique-idx-append*:
assumes *unique-idx gs data* **and** $(hs,\ data') = add\text{-}indices\ aux\ data$
shows *unique-idx* $(gs\ @\ hs)\ data'$
⟨*proof*⟩

corollary *unique-idx-ab*:
assumes *ab-spec ab* **and** *unique-idx* $(gs\ @\ bs)$ *data* **and** $(hs,\ data') = add\text{-}indices\ aux\ data$
shows *unique-idx* $(gs\ @\ ab\ gs\ bs\ hs\ data')\ data'$
⟨*proof*⟩

lemma *rem-comps-spec-struct*:
assumes *struct-spec sel ap ab compl* **and** *rem-comps-spec* $(gs\ @\ bs)$ *data* **and** $ps \neq []$
and $set\ ps \subseteq (set\ bs) \times (set\ gs \cup set\ bs)$ **and** $sps = sel\ gs\ bs\ ps\ (snd\ data)$
and $aux = compl\ gs\ bs\ (ps\ {-}{-}\ sps)\ sps\ (snd\ data)$ **and** $(hs,\ data') = add\text{-}indices\ aux\ (snd\ data)$
shows *rem-comps-spec* $(gs\ @\ ab\ gs\ bs\ hs\ data')\ (fst\ data - count\text{-}const\text{-}lt\text{-}components\ (fst\ aux),\ data')$
⟨*proof*⟩

lemma *pmdl-struct*:
assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl* **and** *is-Groebner-basis* $(fst\ `\ set\ gs)$
and $ps \neq []$ **and** $set\ ps \subseteq (set\ bs) \times (set\ gs \cup set\ bs)$ **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 $(hs,\ data') = add\text{-}indices\ aux\ (snd\ data)$
shows $pmdl\ (fst\ `\ set\ (gs\ @\ ab\ gs\ bs\ hs\ data')) = pmdl\ (fst\ `\ set\ (gs\ @\ bs))$
⟨*proof*⟩

lemma *discarded-subset*:
assumes *ab-spec ab*
and $D' = D \cup (set\ hs \times (set\ gs \cup set\ bs \cup set\ hs) \cup set\ (ps\ {-}{-}\ sps) -_p set\ (ap\ gs\ bs\ (ps\ {-}{-}\ sps)\ hs\ data'))$
and $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs)$ **and** $D \subseteq (set\ gs \cup set\ bs) \times (set\ gs \cup set\ bs)$
shows $D' \subseteq (set\ gs \cup set\ (ab\ gs\ bs\ hs\ data')) \times (set\ gs \cup set\ (ab\ gs\ bs\ hs\ data'))$
⟨*proof*⟩

lemma *compl-struct-disjoint*:
assumes *compl-struct compl* **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$
shows $fst\ `\ set\ (fst\ (compl\ gs\ bs\ (ps\ {-}{-}\ sps)\ sps\ data)) \cap fst\ `\ (set\ gs \cup set\ bs) = \{\}$
⟨*proof*⟩

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*′)
)
)
)
⟨*proof*⟩

lemma *gb-schema-dummy-domI1*: *gb-schema-dummy-dom* (*data*, *D*, *bs*, [])
⟨*proof*⟩

lemma *gb-schema-dummy-domI2*:
assumes *struct-spec sel ap ab compl*
shows *gb-schema-dummy-dom* (*data*, *D*, *args*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*)
)
)
⟨*proof*⟩

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*) ⟹
rc = *fst data* − *count-const-lt-components* (*fst aux*) ⟹ *0* < *rc* ⟹
D′ = (*D* ∪ ((*set hs* × (*set gs* ∪ *set bs* ∪ *set hs*) ∪ *set* (*ps* −− *sps*)) $-_p$ *set* (*ap gs bs* (*ps* −− *sps*) *hs data′*))) ⟹
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′*)) ⟹
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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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* $\subseteq$ *set bs* $\times$ (*set gs* $\cup$ *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*⟩

lemma *snd-gb-schema-dummy-subset*:
assumes *struct-spec sel ap ab compl* **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** *res* = *gb-schema-dummy data D bs ps*
shows *snd res* $\subseteq$ *set* (*fst res*) $\times$ *set* (*fst res*) $\vee$ ($\exists$ *xs*. *fst* (*res*) = *full-gb xs*)
⟨*proof*⟩

lemma *gb-schema-dummy-connectible1*:
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 *unique-idx* (*gs* @ *bs*) (*snd data*)
and $\bigwedge$*p q*. *processed* (*p*, *q*) (*gs* @ *bs*) *ps* $\Longrightarrow$ (*p*, *q*) $\notin_p$ *D* $\Longrightarrow$ *fst p* $\neq$ *0* $\Longrightarrow$ *fst q* $\neq$ *0* $\Longrightarrow$
crit-pair-cbelow-on d m (*fst ‘* (*set gs* $\cup$ *set bs*)) (*fst p*) (*fst q*)
and $\neg$($\exists$ *xs*. *fst* (*gb-schema-dummy data D bs ps*) = *full-gb xs*)
assumes *f* $\in$ *set* (*fst* (*gb-schema-dummy data D bs ps*))
and *g* $\in$ *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* $\neq$ *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*)
⟨*proof*⟩

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 $\bigwedge x\ y.\ x \in set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)) \Longrightarrow y \in set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)) \Longrightarrow$
$(x, y) \notin_p snd\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps) \Longrightarrow fst\ x \neq 0 \Longrightarrow fst\ y \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)))\ (fst\ x)\ (fst\ y)$
and $\neg(\exists xs.\ fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps) = full\text{-}gb\ xs)$
assumes $(f, g) \in_p snd\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)$
and $fst\ f \neq 0$ **and** $fst\ g \neq 0$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)))\ (fst\ f)\ (fst\ g)$
⟨*proof*⟩

corollary *gb-schema-dummy-connectible*:
assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *dickson-grading d*
and $fst\ `\ set\ gs \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** $is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs)$
and $fst\ `\ set\ bs \subseteq dgrad\text{-}p\text{-}set\ d\ m$
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** $unique\text{-}idx\ (gs\ @\ bs)\ (snd\ data)$
and $\bigwedge p\ q.\ processed\ (p, q)\ (gs\ @\ bs)\ ps \Longrightarrow (p, q) \notin_p D \Longrightarrow fst\ p \neq 0 \Longrightarrow fst\ q \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ (set\ gs \cup set\ bs))\ (fst\ p)\ (fst\ q)$
and $\bigwedge B\ a\ b.\ set\ gs \cup set\ bs \subseteq B \Longrightarrow fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow (a, b) \in_p D \Longrightarrow$
$fst\ a \neq 0 \Longrightarrow fst\ b \neq 0 \Longrightarrow$
$(\bigwedge x\ y.\ x \in set\ gs \cup set\ bs \Longrightarrow y \in set\ gs \cup set\ bs \Longrightarrow \neg\ (x, y) \in_p D \Longrightarrow$
$fst\ x \neq 0 \Longrightarrow fst\ y \neq 0 \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ x)\ (fst\ y)) \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ a)\ (fst\ b)$
assumes $f \in set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps))$
and $g \in set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps))$
and $fst\ f \neq 0$ **and** $fst\ g \neq 0$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps)))\ (fst\ f)\ (fst\ g)$
⟨*proof*⟩

lemma *fst-gb-schema-dummy-dgrad-p-set-le-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ (fst\ `\ set\ (fst\ (gb\text{-}schema\text{-}dummy\ data\ D\ (ab\ gs\ []\ bs\ (snd\ data))\ (ap\ gs\ []\ []\ bs\ (snd\ data)))))$
$(fst\ `\ (set\ gs \cup set\ bs))$
⟨*proof*⟩

corollary *fst-gb-schema-dummy-dgrad-p-set-init*:
assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
and $fst\ `\ (set\ gs \cup set\ bs) \subseteq dgrad\text{-}p\text{-}set\ d\ m$
shows $fst\ `\ set\ (fst\ (gb\text{-}schema\text{-}dummy\ (rc, data)\ D\ (ab\ gs\ []\ bs\ data)\ (ap\ gs\ []\ []$

bs data))) ⊆ *dgrad-p-set d m*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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'*)
)
)
)
⟨*proof*⟩

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*, [])
⟨*proof*⟩

lemma *gb-schema-aux-domI2*:
assumes *struct-spec sel ap ab compl*
shows *gb-schema-aux-dom* (*data*, *args*)
⟨*proof*⟩

lemma *gb-schema-aux-Nil* [*simp*, *code*]: *gb-schema-aux data bs* [] = *gs* @ *bs*
⟨*proof*⟩

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*: $\bigwedge$*bs data. P data bs* [] (*gs* @ *bs*)
and *rec1*: $\bigwedge$*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*: $\bigwedge$*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′)) $\Longrightarrow$
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*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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))
⟨*proof*⟩

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)))
⟨*proof*⟩

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)))*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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''*)))
)
⟨*proof*⟩

lemma *gb-schema-incr-dgrad-p-set*:
  assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
    and *fst ‘ set bs* $\subseteq$ *dgrad-p-set d m*
  shows *fst ‘ set (gb-schema-incr sel ap ab compl upd bs data)* $\subseteq$ *dgrad-p-set d m*
  ⟨*proof*⟩

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*⟩

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*⟩

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) $\subseteq$ *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*
⟨*proof*⟩

lemma *icrit-specD*:
assumes *icrit-spec crit* **and** *dickson-grading d*
and fst ‘ $(set\ gs \cup set\ bs \cup set\ hs) \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data$
and $is\text{-}Groebner\text{-}basis\ (fst$ ‘ $set\ gs)$ **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** $crit\ data\ gs\ bs\ hs\ p\ q$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $(set\ gs \cup set\ bs \cup set\ hs))\ (fst\ p)\ (fst\ q)$
⟨*proof*⟩

lemma *ncrit-specI*:
assumes $\bigwedge d\ m\ data\ gs\ bs\ hs\ ps\ B\ q\text{-}in\text{-}bs\ p\ q.$
$dickson\text{-}grading\ d \Longrightarrow set\ gs \cup set\ bs \cup set\ hs \subseteq B \Longrightarrow$
fst ‘ $B \subseteq dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow snd$ ‘ $set\ ps \subseteq set\ hs \times (set\ gs \cup set\ bs \cup set\ hs) \Longrightarrow$
$unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \Longrightarrow is\text{-}Groebner\text{-}basis\ (fst$ ‘ $set\ gs) \Longrightarrow$
$(q\text{-}in\text{-}bs \longrightarrow q \in set\ gs \cup set\ bs) \Longrightarrow$
$(\bigwedge p'\ q'.\ (p', q') \in_p snd$ ‘ $set\ ps \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p')\ (fst\ q')) \Longrightarrow$
$(\bigwedge p'\ q'.\ p' \in set\ gs \cup set\ bs \Longrightarrow q' \in set\ gs \cup set\ bs \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p')\ (fst\ q')) \Longrightarrow$
$p \in set\ hs \Longrightarrow q \in set\ gs \cup set\ bs \cup set\ hs \Longrightarrow fst\ p \neq 0 \Longrightarrow fst\ q \neq 0 \Longrightarrow$
$crit\ data\ gs\ bs\ hs\ q\text{-}in\text{-}bs\ ps\ p\ q \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p)\ (fst\ q)$
shows *ncrit-spec crit*
⟨*proof*⟩

lemma *ncrit-specD*:
assumes *ncrit-spec crit* **and** *dickson-grading d* **and** $set\ gs \cup set\ bs \cup set\ hs \subseteq B$
and fst ‘ $B \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** snd ‘ $set\ ps \subseteq set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$
and $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data$ **and** $is\text{-}Groebner\text{-}basis\ (fst$ ‘ $set\ gs)$
and $q\text{-}in\text{-}bs \Longrightarrow q \in set\ gs \cup set\ bs$
and $\bigwedge p'\ q'.\ (p', q') \in_p snd$ ‘ $set\ ps \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p')\ (fst\ q')$
and $\bigwedge p'\ q'.\ p' \in set\ gs \cup set\ bs \Longrightarrow q' \in set\ gs \cup set\ bs \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p')\ (fst\ q')$
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 $crit\ data\ gs\ bs\ hs\ q\text{-}in\text{-}bs\ ps\ p\ q$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst$ ‘ $B)\ (fst\ p)\ (fst\ q)$
⟨*proof*⟩

lemma *ocrit-specI*:
assumes $\bigwedge d\ m\ data\ hs\ ps\ B\ p\ q.$
$dickson\text{-}grading\ d \Longrightarrow set\ hs \subseteq B \Longrightarrow fst$ ‘ $B \subseteq dgrad\text{-}p\text{-}set\ d\ m \Longrightarrow$
$unique\text{-}idx\ (p\ \#\ q\ \#\ hs\ @\ (map\ (fst \circ snd)\ ps)\ @\ (map\ (snd \circ snd)$

ps)) *data* $\Longrightarrow$
$(\bigwedge p'\ q'.\ (p',\ q') \in_p snd\ `\ set\ ps \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p')\ (fst\ q')) \Longrightarrow$
$p \in B \Longrightarrow q \in B \Longrightarrow fst\ p \neq 0 \Longrightarrow fst\ q \neq 0 \Longrightarrow$
$crit\ data\ hs\ ps\ p\ q \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p)\ (fst\ q)$
shows *ocrit-spec crit*
⟨*proof*⟩

lemma *ocrit-specD*:
assumes *ocrit-spec crit* **and** *dickson-grading d* **and** $set\ hs \subseteq B$ **and** $fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m$
and *unique-idx* (p # q # *hs* @ (*map* (*fst* ∘ *snd*) *ps*) @ (*map* (*snd* ∘ *snd*) *ps*)) *data*
and $\bigwedge p'\ q'.\ (p',\ q') \in_p snd\ `\ set\ ps \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p')\ (fst\ q')$
and $p \in B$ **and** $q \in B$ **and** $fst\ p \neq 0$ **and** $fst\ q \neq 0$
and *crit data hs ps p q*
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p)\ (fst\ q)$
⟨*proof*⟩

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*⟩

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*⟩

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* =
$((i = i' \wedge j = j') \vee (i = j' \wedge j = i') \vee pair\text{-}in\text{-}list\ ps\ i\ j)$

lemma (**in** −) *pair-in-listE*:
assumes *pair-in-list ps i j*
obtains *p q a b* **where** $((p,\ i,\ a),\ (q,\ j,\ b)) \in_p snd\ `\ set\ ps$

$\langle proof \rangle$

definition *chain-ncrit* :: ('t, 'b::zero, 'c, 'd) *ncritT*
where *chain-ncrit data gs bs hs q-in-bs ps p q* $\longleftrightarrow$
(*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* $\longleftrightarrow$
(*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*
($\exists$ *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*))

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* $\subseteq$ *set hs* $\times$ (*set gs* $\cup$ *set bs* $\cup$ *set hs*)
and *unique-idx* (*gs* @ *bs* @ *hs*) *data* **and** *p* $\in$ *set hs* **and** *q* $\in$ *set gs* $\cup$ *set bs* $\cup$ *set hs*
obtains *r* **where** *r* $\in$ *set gs* $\cup$ *set bs* $\cup$ *set hs* **and** *fst r* $\neq$ *0* **and** *r* $\neq$ *p* **and** *r* $\neq$ *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*) $\in_p$ *snd* ‘ *set ps* **and** (*r* $\in$ *set gs* $\cup$ *set bs* $\wedge$ *q-in-bs*) $\vee$ (*q*, *r*) $\in_p$ *snd* ‘ *set ps*
$\langle proof \rangle$

lemma *chain-ocritE*:
assumes *chain-ocrit data hs ps p q*
and *unique-idx* (*p* # *q* # *hs* @ (*map* (*fst* $\circ$ *snd*) *ps*) @ (*map* (*snd* $\circ$ *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*
$\langle proof \rangle$

lemma *ncrit-spec-chain-ncrit*: *ncrit-spec* (*chain-ncrit*::('t, 'b::*field*, 'c, 'd) *ncritT*)
⟨*proof*⟩

lemma *ocrit-spec-chain-ocrit*: *ocrit-spec* (*chain-ocrit*::('t, 'b::*field*, 'c, 'd) *ocritT*)
⟨*proof*⟩

lemma *icrit-spec-no-crit*: *icrit-spec* ((λ- - - - - -. *False*)::('t, 'b::*field*, 'c, 'd) *icritT*)
⟨*proof*⟩

lemma *ncrit-spec-no-crit*: *ncrit-spec* ((λ- - - - - - - -. *False*)::('t, 'b::*field*, 'c, 'd) *ncritT*)
⟨*proof*⟩

lemma *ocrit-spec-no-crit*: *ocrit-spec* ((λ- - - - -. *False*)::('t, 'b::*field*, 'c, 'd) *ocritT*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *np-specD1*:
  assumes *np-spec np*
  shows *snd ‘ set (np gs bs hs data)* $\subseteq$ *set hs* $\times$ *(set gs* $\cup$ *set bs* $\cup$ *set hs)*
  ⟨*proof*⟩

lemma *np-specD2*:
  assumes *np-spec np*
  shows *set hs* $\times$ *(set gs* $\cup$ *set bs)* $\subseteq$ *snd ‘ set (np gs bs hs data)*
  ⟨*proof*⟩

lemma *np-specD3*:
  assumes *np-spec np* **and** $a \in$ *set hs* **and** $b \in$ *set hs* **and** $a \neq b$
  shows $(a, b) \in_p$ *snd ‘ set (np gs bs hs data)*
  ⟨*proof*⟩

lemma *np-specD4*:
  assumes *np-spec np* **and** *(True, p, q)* $\in$ *set (np gs bs hs data)*
  shows $q \in$ *set gs* $\cup$ *set bs*
  ⟨*proof*⟩

lemma *np-specE*:
  assumes *np-spec np* **and** $p \in$ *set hs* **and** $q \in$ *set gs* $\cup$ *set bs* $\cup$ *set hs* **and** $p \neq q$
  assumes *1*: $\bigwedge$*q-in-bs. (q-in-bs, p, q)* $\in$ *set (np gs bs hs data)* $\Longrightarrow$ *thesis*
  assumes *2*: $\bigwedge$*p-in-bs. (p-in-bs, q, p)* $\in$ *set (np gs bs hs data)* $\Longrightarrow$ *thesis*
  shows *thesis*
⟨*proof*⟩

definition *add-pairs-single-naive* :: *'d* $\Rightarrow$ *('t, 'b::zero, 'c) apsT*
  where *add-pairs-single-naive data flag gs bs h ps = ps @ (map (λg. (flag, h, g)) gs) @ (map (λb. (flag, h, b)) bs)*

lemma *set-add-pairs-single-naive*:
  set (add-pairs-single-naive data flag gs bs h ps) = set ps $\cup$ *Pair flag ‘ ({h}* $\times$ *(set gs* $\cup$ *set bs))*
  ⟨*proof*⟩

fun *add-pairs-single-sorted* :: *((bool* $\times$ *('t, 'b, 'c) pdata-pair)* $\Rightarrow$ *(bool* $\times$ *('t, 'b, 'c) pdata-pair)* $\Rightarrow$ *bool)* $\Rightarrow$
      ('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 $\cup$ *Pair flag ‘ ({h}* $\times$ *(set gs* $\cup$ *set bs))*

$\langle proof \rangle$

primrec (**in** $-$) *pairs* :: $('t, 'b, 'c)$ *apsT* $\Rightarrow$ *bool* $\Rightarrow$ $('t, 'b, 'c)$ *pdata list* $\Rightarrow$ (*bool* $\times$ $('t, 'b, 'c)$ *pdata-pair*) *list*
where
pairs - - [] = []|
pairs aps flag (x # xs) = *aps flag* [] xs x (*pairs aps flag xs*)

lemma *pairs-subset*:
assumes $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ `\ (\{h\} \times (set\ gs \cup set\ bs))$
shows $set\ (pairs\ aps\ flag\ xs) \subseteq Pair\ flag\ `\ (set\ xs \times set\ xs)$
$\langle proof \rangle$

lemma *in-pairsI*:
assumes $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ `\ (\{h\} \times (set\ gs \cup set\ bs))$
and $a \neq b$ **and** $a \in set\ xs$ **and** $b \in set\ xs$
shows $(flag, a, b) \in set\ (pairs\ aps\ flag\ xs) \vee (flag, b, a) \in set\ (pairs\ aps\ flag\ xs)$
$\langle proof \rangle$

corollary *in-pairsI'*:
assumes $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ `\ (\{h\} \times (set\ gs \cup set\ bs))$
and $a \in set\ xs$ **and** $b \in set\ xs$ **and** $a \neq b$
shows $(a, b) \in_p snd\ `\ set\ (pairs\ aps\ flag\ xs)$
$\langle proof \rangle$

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* $\times$ $'d$ $\Rightarrow$ (*bool* $\times$ $('t, 'b, 'c)$ *pdata-pair*) $\Rightarrow$ (*bool* $\times$ $('t, 'b, 'c)$ *pdata-pair*) $\Rightarrow$ *bool*) $\Rightarrow$
$('t, 'b{::}zero, 'c, 'd)$ *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 $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ `\ (\{h\} \times (set\ gs \cup set\ bs))$
shows $set\ (fold\ (aps\ flag\ gs\ bs)\ hs\ ps) = Pair\ flag\ `\ (set\ hs \times (set\ gs \cup set\ bs)) \cup set\ ps$
$\langle proof \rangle$

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*⟩

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*⟩

lemma (**in** −) *fst-snd-Pair* [*simp*]:
shows *fst* ∘ *Pair x* = (λ-. *x*) **and** *snd* ∘ *Pair x* = *id*
⟨*proof*⟩

lemma *np-spec-new-pairs-naive*: *np-spec new-pairs-naive*
⟨*proof*⟩

lemma *np-spec-new-pairs-sorted*: *np-spec* (*new-pairs-sorted rel*)
⟨*proof*⟩

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*⟩

lemma *snd-apply-icrit* [*simp*]: *map snd* (*apply-icrit crit data gs bs hs ps*) = *ps*
⟨*proof*⟩

lemma *set-snd-apply-icrit* [*simp*]: *snd* ‘ *set* (*apply-icrit crit data gs bs hs ps*) = *set ps*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

corollary *apply-ncrit-subset*:
assumes (*ic*, *p*, *q*) ∈ *set* (*apply-ncrit crit data gs bs hs ps*)

obtains *q-in-bs* **where** $(ic,\ q\text{-}in\text{-}bs,\ p,\ q) \in set\ ps$
$\langle proof \rangle$

corollary *apply-ncrit-subset'*: $snd\ `\ set\ (apply\text{-}ncrit\ crit\ data\ gs\ bs\ hs\ ps) \subseteq snd\ `\ snd\ `\ set\ ps$
$\langle proof \rangle$

lemma *not-in-apply-ncrit*:
assumes $(ic,\ p,\ q) \notin set\ (apply\text{-}ncrit\ crit\ data\ gs\ bs\ hs\ (xs\ @\ ((ic,\ q\text{-}in\text{-}bs,\ p,\ q)\ \#\ ys)))$
shows $crit\ data\ gs\ bs\ hs\ q\text{-}in\text{-}bs\ (rev\ (apply\text{-}ncrit\ crit\ data\ gs\ bs\ hs\ xs))\ p\ q$
$\langle proof \rangle$

lemma (**in** $-$) *setE*:
assumes $x \in set\ xs$
obtains *ys zs* **where** $xs = ys\ @\ (x\ \#\ zs)$
$\langle proof \rangle$

lemma *apply-ncrit-connectible*:
assumes *ncrit-spec crit* **and** *dickson-grading d*
and $set\ gs \cup set\ bs \cup set\ hs \subseteq B$ **and** $fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m$
and $snd\ `\ snd\ `\ set\ ps \subseteq set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)$ **and** $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data$
and $is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs)$
and $\bigwedge p'\ q'.\ (p',\ q') \in snd\ `\ set\ (apply\text{-}ncrit\ crit\ data\ gs\ bs\ hs\ ps) \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p')\ (fst\ q')$
and $\bigwedge p'\ q'.\ p' \in set\ gs \cup set\ bs \Longrightarrow q' \in set\ gs \cup set\ bs \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p')\ (fst\ q')$
assumes $(ic,\ q\text{-}in\text{-}bs,\ p,\ q) \in set\ ps$ **and** $fst\ p \neq 0$ **and** $fst\ q \neq 0$
and $q\text{-}in\text{-}bs \Longrightarrow (q \in set\ gs \cup set\ bs)$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p)\ (fst\ q)$
$\langle proof \rangle$

6.3.5 Applying Criteria to Old Pairs

definition $apply\text{-}ocrit :: ('t,\ 'b,\ 'c,\ 'd)\ ocritT \Rightarrow (nat \times 'd) \Rightarrow ('t,\ 'b,\ 'c)\ pdata\ list \Rightarrow (bool \times ('t,\ 'b,\ 'c)\ pdata\text{-}pair)\ list \Rightarrow ('t,\ 'b,\ 'c)\ pdata\text{-}pair\ list \Rightarrow ('t,\ 'b,\ 'c)\ pdata\text{-}pair\ list$
where $apply\text{-}ocrit\ crit\ data\ hs\ ps'\ ps = (let\ c = crit\ data\ hs\ ps'\ in\ [(p,\ q) \leftarrow ps\ .\ \neg\ c\ p\ q])$

lemma *set-apply-ocrit*:
$set\ (apply\text{-}ocrit\ crit\ data\ hs\ ps'\ ps) = \{(p,\ q)\ |\ p\ q.\ (p,\ q) \in set\ ps \wedge \neg\ crit\ data\ hs\ ps'\ p\ q\}$
$\langle proof \rangle$

corollary *set-apply-ocrit-iff*:
$(p, q) \in set\ (apply\text{-}ocrit\ crit\ data\ hs\ ps'\ ps) \longleftrightarrow ((p, q) \in set\ ps \wedge \neg\ crit\ data\ hs\ ps'\ p\ q)$
⟨*proof*⟩

lemma *apply-ocrit-connectible*:
assumes *ocrit-spec crit* **and** *dickson-grading d* **and** $set\ hs \subseteq B$ **and** $fst\ `\ B \subseteq dgrad\text{-}p\text{-}set\ d\ m$
and $unique\text{-}idx\ (p\ \#\ q\ \#\ hs\ @\ (map\ (fst \circ snd)\ ps')\ @\ (map\ (snd \circ snd)\ ps'))\ data$
and $\bigwedge p'\ q'.\ (p', q') \in snd\ `\ set\ ps' \Longrightarrow fst\ p' \neq 0 \Longrightarrow fst\ q' \neq 0 \Longrightarrow$
$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p')\ (fst\ q')$
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** $(p, q) \notin set\ (apply\text{-}ocrit\ crit\ data\ hs\ ps'\ ps)$
shows $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ `\ B)\ (fst\ p)\ (fst\ q)$
⟨*proof*⟩

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* [$x \leftarrow ps1$. $\neg$ *fst x*]) *ps2*)

lemma *set-add-pairs*:
assumes $\bigwedge xs\ ys.\ set\ (comb\ xs\ ys) = set\ xs \cup set\ ys$
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*) =
$\{(p, q) \mid p\ q.\ (False, p, q) \in set\ ps1 \vee ((p, q) \in set\ ps \wedge \neg\ ocrit\ data\ hs\ ps1\ p\ q)\}$
⟨*proof*⟩

lemma *set-add-pairs-iff*:
assumes $\bigwedge xs\ ys.\ set\ (comb\ xs\ ys) = set\ xs \cup set\ ys$
assumes *ps1* = *apply-ncrit ncrit data gs bs hs* (*apply-icrit icrit data gs bs hs* (*np gs bs hs data*))

shows $((p,\ q) \in set\ (add\text{-}pairs\ gs\ bs\ ps\ hs\ data)) \longleftrightarrow$
$((False,\ p,\ q) \in set\ ps1 \lor ((p,\ q) \in set\ ps \land \neg\ ocrit\ data\ hs\ ps1\ p\ q))$
$\langle proof \rangle$

lemma *ap-spec-add-pairs*:
assumes *np-spec np* **and** *icrit-spec icrit* **and** *ncrit-spec ncrit* **and** *ocrit-spec ocrit*
and $\bigwedge xs\ ys.\ set\ (comb\ xs\ ys) = set\ xs \cup set\ ys$
shows *ap-spec add-pairs*
$\langle proof \rangle$

end

abbreviation *add-pairs-canon* $\equiv$
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*
$\langle proof \rangle$

6.4 Suitable Instances of the *completion* Parameter

definition *rcp-spec* :: $('t,\ 'b{::}field,\ 'c,\ 'd)\ complT \Rightarrow bool$
where $rcp\text{-}spec\ rcp \longleftrightarrow$
$(\forall\, gs\ bs\ ps\ sps\ data.$
$0 \notin fst\ `\ set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \land$
$(\forall\, h\ b.\ h \in set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \longrightarrow b \in set\ gs \cup set\ bs \longrightarrow fst\ b \neq 0 \longrightarrow$
$\neg\ lt\ (fst\ b)\ adds_t\ lt\ (fst\ h)) \land$
$(\forall\, d.\ dickson\text{-}grading\ d \longrightarrow$
$dgrad\text{-}p\text{-}set\text{-}le\ d\ (fst\ `\ set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))\ (args\text{-}to\text{-}set\ (gs,\ bs,\ sps))) \land$
$component\text{-}of\text{-}term\ `\ Keys\ (fst\ `\ (set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))) \subseteq$
$component\text{-}of\text{-}term\ `\ Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ sps)) \land$
$(is\text{-}Groebner\text{-}basis\ (fst\ `\ set\ gs) \longrightarrow unique\text{-}idx\ (gs\ @\ bs)\ data \longrightarrow$
$(fst\ `\ set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \subseteq pmdl\ (args\text{-}to\text{-}set\ (gs,\ bs,\ sps)) \land$
$(\forall\, (p,\ q){\in}set\ sps.\ \ set\ sps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow$
$(red\ (fst\ `\ (set\ gs \cup set\ bs) \cup fst\ `\ set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))))^{**}\ (spoly\ (fst\ p)\ (fst\ q))\ 0))))$

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* $\cup$ *set gs*, whose leading terms are not divisible by the leading term of any non-zero $b \in set\ bs$, 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 $\bigwedge$*gs bs ps sps data. 0* $\notin$ *fst ' set* (*fst* (*rcp gs bs ps sps data*))
  assumes $\bigwedge$*gs bs ps sps h b data. h* $\in$ *set* (*fst* (*rcp gs bs ps sps data*)) $\Longrightarrow$ *b* $\in$ *set gs* $\cup$ *set bs* $\Longrightarrow$ *fst b* $\neq$ *0* $\Longrightarrow$
    $\neg$ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
  assumes $\bigwedge$*gs bs ps sps d data. dickson-grading d* $\Longrightarrow$
    dgrad-p-set-le d (*fst ' set* (*fst* (*rcp gs bs ps sps data*))) (*args-to-set* (*gs*, *bs*, *sps*))
  assumes $\bigwedge$*gs bs ps sps data. component-of-term ' Keys* (*fst '* (*set* (*fst* (*rcp gs bs ps sps data*)))) $\subseteq$
    component-of-term ' Keys (*args-to-set* (*gs*, *bs*, *sps*))
  assumes $\bigwedge$*gs bs ps sps data. is-Groebner-basis* (*fst ' set gs*) $\Longrightarrow$ *unique-idx* (*gs* @ *bs*) *data* $\Longrightarrow$
    (*fst ' set* (*fst* (*rcp gs bs ps sps data*)) $\subseteq$ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*)) $\wedge$
    ($\forall$ (*p*, *q*)$\in$*set sps*. *set sps* $\subseteq$ *set bs* $\times$ (*set gs* $\cup$ *set bs*) $\longrightarrow$
    (*red* (*fst '* (*set gs* $\cup$ *set bs*) $\cup$ *fst ' set* (*fst* (*rcp gs bs ps sps data*))))** (*spoly* (*fst p*) (*fst q*)) *0*))
  shows *rcp-spec rcp*
  ⟨*proof*⟩

lemma *rcp-specD1*:
  assumes *rcp-spec rcp*
  shows *0* $\notin$ *fst ' set* (*fst* (*rcp gs bs ps sps data*))
  ⟨*proof*⟩

lemma *rcp-specD2*:
  assumes *rcp-spec rcp*
    and *h* $\in$ *set* (*fst* (*rcp gs bs ps sps data*)) **and** *b* $\in$ *set gs* $\cup$ *set bs* **and** *fst b* $\neq$ *0*
  shows $\neg$ *lt* (*fst b*) $adds_t$ *lt* (*fst h*)
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma *rcp-specD4*:
  assumes *rcp-spec rcp*
  shows *component-of-term ' Keys* (*fst '* (*set* (*fst* (*rcp gs bs ps sps data*)))) $\subseteq$
    component-of-term ' Keys (*args-to-set* (*gs*, *bs*, *sps*))
  ⟨*proof*⟩

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*)) $\subseteq$ *pmdl* (*args-to-set* (*gs*, *bs*, *sps*))
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *compl-struct-rcp*:
assumes *rcp-spec rcp*
shows *compl-struct rcp*
⟨*proof*⟩

lemma *compl-pmdl-rcp*:
assumes *rcp-spec rcp*
shows *compl-pmdl rcp*
⟨*proof*⟩

lemma *compl-conn-rcp*:
assumes *rcp-spec rcp*
shows *compl-conn rcp*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *full-gb-punit* [*code*]:
punit.full-gb bs = (*if* (∀ *b*∈*set bs*. *fst b* = *0*) *then* [] *else* [(*1*, *0*, *default*)])
⟨*proof*⟩

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*
⟨*proof*⟩

end

end

7 Buchberger's Algorithm

theory *Buchberger*
imports *Algorithm-Schema*
begin

context *gd-term*
begin

7.1 Reduction

definition $trdsp::('t \Rightarrow_0 'b)\ list \Rightarrow ('t, 'b, 'c)\ pdata\text{-}pair \Rightarrow ('t \Rightarrow_0 'b::field)$
  where $trdsp\ bs\ p \equiv trd\ bs\ (spoly\ (fst\ (fst\ p))\ (fst\ (snd\ p)))$

lemma *trdsp-alt*: $trdsp\ bs\ (p, q) = trd\ bs\ (spoly\ (fst\ p)\ (fst\ q))$
  ⟨*proof*⟩

lemma *trdsp-in-pmdl*: $trdsp\ bs\ (p, q) \in pmdl\ (insert\ (fst\ p)\ (insert\ (fst\ q)\ (set\ bs)))$
  ⟨*proof*⟩

lemma *dgrad-p-set-le-trdsp*:
  assumes *dickson-grading d*
  shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ \{trdsp\ bs\ (p, q)\}\ (insert\ (fst\ p)\ (insert\ (fst\ q)\ (set\ bs)))$
⟨*proof*⟩

lemma *components-trdsp-subset*:
  $component\text{-}of\text{-}term\ `\ keys\ (trdsp\ bs\ (p, q)) \subseteq component\text{-}of\text{-}term\ `\ Keys\ (insert\ (fst\ p)\ (insert\ (fst\ q)\ (set\ bs)))$
⟨*proof*⟩

definition $gb\text{-}red\text{-}aux :: ('t, 'b::field, 'c)\ pdata\ list \Rightarrow ('t, 'b, 'c)\ pdata\text{-}pair\ list \Rightarrow$
    $('t \Rightarrow_0 'b)\ list$
  where $gb\text{-}red\text{-}aux\ bs\ ps =$
    $(let\ bs' = map\ fst\ bs\ in$
      $filter\ (\lambda h.\ h \neq 0)\ (map\ (trdsp\ bs')\ ps)$
    $)$

Actually, *gb-red-aux* is only called on singleton lists.

lemma *set-gb-red-aux*: $set\ (gb\text{-}red\text{-}aux\ bs\ ps) = (trdsp\ (map\ fst\ bs))\ `\ set\ ps - \{0\}$
  ⟨*proof*⟩

lemma *in-set-gb-red-auxI*:
  assumes $(p, q) \in set\ ps$ **and** $h = trdsp\ (map\ fst\ bs)\ (p, q)$ **and** $h \neq 0$
  shows $h \in set\ (gb\text{-}red\text{-}aux\ bs\ ps)$
  ⟨*proof*⟩

lemma *in-set-gb-red-auxE*:
  assumes $h \in set\ (gb\text{-}red\text{-}aux\ bs\ ps)$
  obtains $p\ q$ **where** $(p, q) \in set\ ps$ **and** $h = trdsp\ (map\ fst\ bs)\ (p, q)$
  ⟨*proof*⟩

lemma *gb-red-aux-not-zero*: $0 \notin set\ (gb\text{-}red\text{-}aux\ bs\ ps)$
  ⟨*proof*⟩

lemma *gb-red-aux-irredudible*:
assumes $h \in$ *set* (*gb-red-aux bs ps*) **and** $b \in$ *set bs* **and** *fst* $b \neq 0$
shows $\neg$ *lt* (*fst b*) $adds_t$ *lt h*
⟨*proof*⟩

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*⟩

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*⟩

lemma *pmdl-gb-red-aux*: *set* (*gb-red-aux bs ps*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *rcp-spec-gb-red*: *rcp-spec gb-red*
⟨*proof*⟩

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*⟩

7.3 Buchberger's Algorithm

lemma *struct-spec-gb*: *struct-spec gb-sel add-pairs-canon add-basis-canon gb-red*

⟨*proof*⟩

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*
⟨*proof*⟩

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 ((∑*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 $1 \leq n$, *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* $\Rightarrow_0$ *'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*⟩

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* ⟨*proof*⟩

lift-definition *quotient-of′* :: *rat* ⇒ *integer* × *integer* **is** *quotient-of* ⟨*proof*⟩

lemma [*code*]: *Rat.of-int* (*int-of-integer x*) = *rat-of-integer x*
⟨*proof*⟩

lemma [*code-unfold*]: *quotient-of* = (*λx. map-prod int-of-integer int-of-integer* (*quotient-of′ x*))
⟨*proof*⟩

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*
)
⟨*proof*⟩

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.
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *compute-trd-punit* [*code*]: *trd-punit to fs p* = *trd-aux-punit to fs p* (*change-ord to 0*)
⟨*proof*⟩

experiment begin interpretation *trivariate*$_0$*-rat* ⟨*proof*⟩

lemma
lt-punit DRLEX ($X^2 * Z$ ^ $3 + 3 * X^2 * Y$) = *sparse*$_0$ [(*0*, *2*), (*2*, *3*)]
⟨*proof*⟩

lemma
lc-punit DRLEX ($X^2 * Z$ ^ $3 + 3 * X^2 * Y$) = *1*

$\langle proof \rangle$

lemma
tail-punit DRLEX $(X^2 * Z \,\hat{}\, 3 + 3 * X^2 * Y) = 3 * X^2 * Y$
$\langle proof \rangle$

lemma
ord-strict-p-punit DRLEX $(X^2 * Z \,\hat{}\, 4 - 2 * Y \,\hat{}\, 3 * Z^2)$ $(X^2 * Z \,\hat{}\, 7 + 2 * Y \,\hat{}\, 3 * Z^2)$
$\langle proof \rangle$

lemma
trd-punit DRLEX $[Y^2 * Z + 2 * Y * Z \,\hat{}\, 3]$ $(X^2 * Z \,\hat{}\, 4 - 2 * Y \,\hat{}\, 3 * Z \,\hat{}\, 3) =$
$X^2 * Z \,\hat{}\, 4 + Y \,\hat{}\, 4 * Z$
$\langle proof \rangle$

lemma
spoly-punit DRLEX $(X^2 * Z \,\hat{}\, 4 - 2 * Y \,\hat{}\, 3 * Z^2)$ $(Y^2 * Z + 2 * Z \,\hat{}\, 3) =$
$-2 * Y \,\hat{}\, 3 * Z^2 - (C_0\ (1\ /\ 2)) * X^2 * Y^2 * Z^2$
$\langle proof \rangle$

lemma
gb-punit DRLEX
[
$(X^2 * Z \,\hat{}\, 4 - 2 * Y \,\hat{}\, 3 * Z^2, ())$,
$(Y^2 * Z + 2 * Z \,\hat{}\, 3, ())$
] () =
[
$(-2 * Y \,\hat{}\, 3 * Z^2 - (C_0\ (1\ /\ 2)) * X^2 * Y^2 * Z^2, ())$,
$(X^2 * Z \,\hat{}\, 4 - 2 * Y \,\hat{}\, 3 * Z^2, ())$,
$(Y^2 * Z + 2 * Z \,\hat{}\, 3, ())$,
$(-\ (C_0\ (1\ /\ 2)) * X^2 * Y \,\hat{}\, 4 * Z - 2 * Y \,\hat{}\, 5 * Z, ())$
]
$\langle proof \rangle$

lemma
gb-punit DRLEX
[
$(X^2 * Z^2 - Y, ())$,
$(Y^2 * Z - 1, ())$
] () =
[
$(-\ (Y \,\hat{}\, 3) + X^2 * Z, ())$,
$(X^2 * Z^2 - Y, ())$,
$(Y^2 * Z - 1, ())$
]
$\langle proof \rangle$

lemma
gb-punit DRLEX
[
$(X \mathbin{\char`\^} 3 - X * Y * Z^2, ())$,
$(Y^2 * Z - 1, ())$
] () =
[
$(- (X \mathbin{\char`\^} 3 * Y) + X * Z, ())$,
$(X \mathbin{\char`\^} 3 - X * Y * Z^2, ())$,
$(Y^2 * Z - 1, ())$,
$(- (X * Z \mathbin{\char`\^} 3) + X \mathbin{\char`\^} 5, ())$
]
⟨*proof*⟩

lemma
gb-punit DRLEX
[
$(X^2 + Y^2 + Z^2 - 1, ())$,
$(X * Y - Z - 1, ())$,
$(Y^2 + X, ())$,
$(Z^2 + X, ())$
] () =
[
$(1, ())$
]
⟨*proof*⟩

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* $\lambda x::('a, 'b)$ *pp* $\times$ *'c. x* $\lambda x.$ *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* $\times$ *'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*
⟨*proof*⟩

lemma *compute-adds-term-pprod* [*code*]:
adds-term-pprod u v = (*snd u* = *snd v* $\wedge$ *adds-pp-add-linorder* (*fst u*) (*fst v*))

⟨*proof*⟩

lemma *compute-splus-pprod* [*code*]: *splus-pprod t* (*s*, *i*) = (*t* + *s*, *i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compute-trd-pprod* [*code*]: *trd-pprod to fs p* = *trd-aux-pprod to fs p* (*change-ord to 0*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma
ord-p-pprod (*POT DRLEX*) (Vec_0 *1* ($X^2 * Z$) + Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)) (Vec_0 *1* ($X^2 * Z^2 + 2 * Y$ ^ $3 * Z^2$))
⟨*proof*⟩

lemma
tail-pprod (*POT DRLEX*) (Vec_0 *1* ($X^2 * Z$) + Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)) = Vec_0 *0* ($2 * Y$ ^ $3 * Z^2$)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)}
⟨*proof*⟩

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*)}
⟨*proof*⟩

lemma
Vec_0 *1* ($X^2 * Z$ ^ $7 + 2 * Y$ ^ $3 * Z^2$) + Vec_0 *3* ($X^2 * Z$ ^ *4*) + Vec_0 *1* (− *2*

$* \ Y \ \hat{} \ 3 \ * \ Z^2) =$
  $Vec_0 \ 1 \ (X^2 * Z \ \hat{} \ 7) + Vec_0 \ 3 \ (X^2 * Z \ \hat{} \ 4)$
$\langle proof \rangle$

lemma
$lookup \ (Vec_0 \ 0 \ (X^2 * Z \ \hat{} \ 7) + Vec_0 \ 1 \ (2 * Y \ \hat{} \ 3 * Z^2 + 2)) \ (sparse_0 \ [(0, 2), (2, 7)], 0) = 1$
$\langle proof \rangle$

lemma
$lookup \ (Vec_0 \ 0 \ (X^2 * Z \ \hat{} \ 7) + Vec_0 \ 1 \ (2 * Y \ \hat{} \ 3 * Z^2 + 2)) \ (sparse_0 \ [(0, 2), (2, 7)], 1) = 0$
$\langle proof \rangle$

lemma
$Vec_0 \ 0 \ (0 * X\hat{}2 * Z\hat{}7) + Vec_0 \ 1 \ (0 * Y\hat{}3*Z^2) = 0$
$\langle proof \rangle$

lemma
$monom\text{-}mult\text{-}pprod \ 3 \ (sparse_0 \ [(1, 2{::}nat)]) \ (Vec_0 \ 0 \ (X^2 * Z) + Vec_0 \ 1 \ (2 * Y \ \hat{} \ 3 * Z^2)) =$
  $Vec_0 \ 0 \ (3 * Y^2 * Z * X^2) + Vec_0 \ 1 \ (6 * Y \ \hat{} \ 5 * Z^2)$
$\langle proof \rangle$

lemma
$trd\text{-}pprod \ DRLEX \ [Vec_0 \ 0 \ (Y^2 * Z + 2 * Y * Z \ \hat{} \ 3)] \ (Vec_0 \ 0 \ (X^2 * Z \ \hat{} \ 4 - 2 * Y \ \hat{} \ 3 * Z \ \hat{} \ 3)) =$
  $Vec_0 \ 0 \ (X^2 * Z \ \hat{} \ 4 + Y \ \hat{} \ 4 * Z)$
$\langle proof \rangle$

lemma
$length \ (gb\text{-}pprod \ (POT \ DRLEX)$
  [
   $(Vec_0 \ 0 \ (X^2 * Z \ \hat{} \ 4 - 2 * Y \ \hat{} \ 3 * Z^2), ()),$
   $(Vec_0 \ 0 \ (Y^2 * Z + 2 * Z \ \hat{} \ 3), ())$
  $] \ ()) = 4$
$\langle proof \rangle$

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 $p \in span\ (set\ bs)$
obtains *qs* **where** $length\ qs = length\ bs$ **and** $p = sum\text{-}list\ (map2\ (*s)\ qs\ bs)$
⟨*proof*⟩

lemma *span-listI*: $sum\text{-}list\ (map2\ (*s)\ qs\ bs) \in span\ (set\ bs)$
⟨*proof*⟩

end

lemma (**in** *term-powerprod*) *monomial-1-in-pmdlI*:
assumes $(f::\text{-} \Rightarrow_0 {'b}::field) \in pmdl\ F$ **and** $keys\ f = \{t\}$
shows $monomial\ 1\ t \in pmdl\ F$
⟨*proof*⟩

11.2 Ordered Polynomials

context *ordered-term*
begin

11.2.1 Sets of Leading Terms and -Coefficients

definition *lt-set* :: $('t, 'b::zero)\ poly\text{-}mapping\ set \Rightarrow 't\ set$ **where**
$lt\text{-}set\ F = lt\ `\ (F - \{0\})$

definition *lc-set* :: $('t, 'b::zero)\ poly\text{-}mapping\ set \Rightarrow 'b\ set$ **where**
$lc\text{-}set\ F = lc\ `\ (F - \{0\})$

lemma *lt-setI*:
assumes $f \in F$ **and** $f \neq 0$
shows $lt\ f \in lt\text{-}set\ F$
⟨*proof*⟩

lemma *lt-setE*:
assumes $t \in lt\text{-}set\ F$
obtains *f* **where** $f \in F$ **and** $f \neq 0$ **and** $lt\ f = t$
⟨*proof*⟩

lemma *lt-set-iff*:
shows $t \in lt\text{-}set\ F \longleftrightarrow (\exists f{\in}F.\ f \neq 0 \wedge lt\ f = t)$
⟨*proof*⟩

lemma *lc-setI*:
assumes $f \in F$ **and** $f \neq 0$
shows $lc\ f \in lc\text{-}set\ F$
⟨*proof*⟩

lemma *lc-setE*:
assumes $c \in lc\text{-}set\ F$
obtains f **where** $f \in F$ **and** $f \neq 0$ **and** $lc\ f = c$
⟨*proof*⟩

lemma *lc-set-iff*:
shows $c \in lc\text{-}set\ F \longleftrightarrow (\exists f \in F.\ f \neq 0 \wedge lc\ f = c)$
⟨*proof*⟩

lemma *lc-set-nonzero*:
shows $0 \notin lc\text{-}set\ F$
⟨*proof*⟩

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 $lt\ (sum\ p\ I) = ord\text{-}term\text{-}lin.Max\ (lt\text{-}set\ (p\ \text{`}\ I))$
⟨*proof*⟩

lemma *lt-sum-distinct-in-lt-set*:
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 $lt\ (sum\ p\ I) \in lt\text{-}set\ (p\ \text{`}\ I)$
⟨*proof*⟩

11.2.2 Monicity

definition *monic* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b{::}field)$ **where**
$monic\ p = monom\text{-}mult\ (1\ /\ lc\ p)\ 0\ p$

definition *is-monic-set* :: $('t \Rightarrow_0 'b{::}field)\ set \Rightarrow bool$ **where**
$is\text{-}monic\text{-}set\ B \equiv (\forall b \in B.\ b \neq 0 \longrightarrow lc\ b = 1)$

lemma *lookup-monic*: $lookup\ (monic\ p)\ v = (lookup\ p\ v)\ /\ lc\ p$
⟨*proof*⟩

lemma *lookup-monic-lt*:
assumes $p \neq 0$
shows $lookup\ (monic\ p)\ (lt\ p) = 1$
⟨*proof*⟩

lemma *monic-0* [*simp*]: *monic 0 = 0*
⟨*proof*⟩

lemma *monic-0-iff*: (*monic p = 0*) ⟷ (*p = 0*)
⟨*proof*⟩

lemma *keys-monic* [*simp*]: *keys* (*monic p*) = *keys p*
⟨*proof*⟩

lemma *lt-monic* [*simp*]: *lt* (*monic p*) = *lt p*
⟨*proof*⟩

lemma *lc-monic*:
assumes $p \neq 0$
shows *lc* (*monic p*) = *1*
⟨*proof*⟩

lemma *mult-lc-monic*:
assumes $p \neq 0$
shows *monom-mult* (*lc p*) *0* (*monic p*) = *p* (**is** *?q* = *p*)
⟨*proof*⟩

lemma *is-monic-setI*:
assumes $\bigwedge b.\ b \in B \Longrightarrow b \neq 0 \Longrightarrow lc\ b = 1$
shows *is-monic-set B*
⟨*proof*⟩

lemma *is-monic-setD*:
assumes *is-monic-set B* **and** $b \in B$ **and** $b \neq 0$
shows *lc b = 1*
⟨*proof*⟩

lemma *Keys-image-monic* [*simp*]: *Keys* (*monic ‘ A*) = *Keys A*
⟨*proof*⟩

lemma *image-monic-is-monic-set*: *is-monic-set* (*monic ‘ A*)
⟨*proof*⟩

lemma *pmdl-image-monic* [*simp*]: *pmdl* (*monic ‘ B*) = *pmdl B*
⟨*proof*⟩

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) $\longleftrightarrow$ *is-red* B p
⟨*proof*⟩

lemma *red-image-monic* [*simp*]: *red* (*monic* ‘ B) = *red* B
⟨*proof*⟩

lemma *is-red-image-monic* [*simp*]: *is-red* (*monic* ‘ B) p $\longleftrightarrow$ *is-red* B p
⟨*proof*⟩

12.2 Minimal Bases and Auto-reduced Bases

definition *is-auto-reduced* :: ($'t \Rightarrow_0 'b$::*field*) *set* $\Rightarrow$ *bool* **where**
is-auto-reduced $B \equiv (\forall\, b \in B.\ \neg$ *is-red* $(B - \{b\})\ b)$

definition *is-minimal-basis* :: ($'t \Rightarrow_0 'b$::*zero*) *set* $\Rightarrow$ *bool* **where**
is-minimal-basis $B \longleftrightarrow (0 \notin B \wedge (\forall\, p\ q.\ p \in B \longrightarrow q \in B \longrightarrow p \neq q \longrightarrow \neg$ *lt* p *adds*$_t$ *lt* $q))$

lemma *is-auto-reducedD*:
assumes *is-auto-reduced* B **and** $b \in B$
shows $\neg$ *is-red* $(B - \{b\})\ b$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *is-minimal-basisI*:
assumes $\bigwedge p.\ p \in B \Longrightarrow p \neq 0$ **and** $\bigwedge p\ q.\ p \in B \Longrightarrow q \in B \Longrightarrow p \neq q \Longrightarrow \neg$ *lt* p *adds*$_t$ *lt* q
shows *is-minimal-basis* B
⟨*proof*⟩

lemma *is-minimal-basisD1*:
assumes *is-minimal-basis* B **and** $p \in B$
shows $p \neq 0$
⟨*proof*⟩

lemma *is-minimal-basisD2*:
assumes *is-minimal-basis* B **and** $p \in B$ **and** $q \in B$ **and** $p \neq q$

shows $\neg$ *lt p adds*$_t$ *lt q*
⟨*proof*⟩

lemma *is-minimal-basisD3*:
assumes *is-minimal-basis B* **and** $p \in B$ **and** $q \in B$ **and** $p \neq q$
shows $\neg$ *lt q adds*$_t$ *lt p*
⟨*proof*⟩

lemma *is-minimal-basis-subset*:
assumes *is-minimal-basis B* **and** $A \subseteq B$
shows *is-minimal-basis A*
⟨*proof*⟩

lemma *nadds-red*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow \neg$ *lt q adds*$_t$ *lt p* **and** *red*: *red B p r*
shows $r \neq 0 \wedge$ *lt r* $=$ *lt p*
⟨*proof*⟩

lemma *nadds-red-nonzero*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow \neg$ *lt q adds*$_t$ *lt p* **and** *red B p r*
shows $r \neq 0$
⟨*proof*⟩

lemma *nadds-red-lt*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow \neg$ *lt q adds*$_t$ *lt p* **and** *red B p r*
shows *lt r* $=$ *lt p*
⟨*proof*⟩

lemma *nadds-red-rtrancl-lt*:
assumes *nadds*: $\bigwedge q.\ q \in B \Longrightarrow \neg$ *lt q adds*$_t$ *lt p* **and** *rtrancl*: $(red\ B)^{**}$ *p r*
shows *lt r* $=$ *lt p*
⟨*proof*⟩

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 $r \neq 0$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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'*: $set\ (comp\text{-}min\text{-}basis\ xs) \subseteq \{x \in set\ xs.\ x \neq 0\}$
⟨*proof*⟩

lemma *comp-min-basis-subset*: $set\ (comp\text{-}min\text{-}basis\ xs) \subseteq set\ xs$
⟨*proof*⟩

lemma *comp-min-basis-nonzero*: $p \in set\ (comp\text{-}min\text{-}basis\ xs) \Longrightarrow p \neq 0$
⟨*proof*⟩

lemma *comp-min-basis-adds*:
assumes $p \in set\ xs$ **and** $p \neq 0$
obtains *q* **where** $q \in set\ (comp\text{-}min\text{-}basis\ xs)$ **and** $lt\ q\ adds_t\ lt\ p$
⟨*proof*⟩

lemma *comp-min-basis-is-red*:
assumes *is-red* (*set xs*) *f*
shows *is-red* (*set* (*comp-min-basis xs*)) *f*
⟨*proof*⟩

lemma *comp-min-basis-nadds*:
assumes $p \in set\ (comp\text{-}min\text{-}basis\ xs)$ **and** $q \in set\ (comp\text{-}min\text{-}basis\ xs)$ **and** $p \neq q$
shows $\neg\ lt\ q\ adds_t\ lt\ p$
⟨*proof*⟩

lemma *comp-min-basis-is-minimal-basis*: *is-minimal-basis* (*set* (*comp-min-basis xs*))
⟨*proof*⟩

lemma *comp-min-basis-distinct*: *distinct* (*comp-min-basis xs*)
⟨*proof*⟩

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*⟩

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*⟩

primrec *comp-red-basis-aux* :: ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t \Rightarrow_0 'b$::*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* ⊆ *set* (*comp-red-basis-aux xs ys*)
⟨*proof*⟩

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 $p \neq 0$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

definition *comp-red-basis* :: ($'t \Rightarrow_0 'b$) *list* ⇒ ($'t \Rightarrow_0 'b$::*field*) *list*
where *comp-red-basis xs* = *comp-red-basis-aux* (*comp-min-basis xs*) []

lemma *comp-red-basis-nonzero*:

assumes $p \in set\ (comp\text{-}red\text{-}basis\ xs)$
shows $p \neq 0$
⟨*proof*⟩

lemma *pmdl-comp-red-basis-subset*: $pmdl\ (set\ (comp\text{-}red\text{-}basis\ xs)) \subseteq pmdl\ (set\ xs)$
⟨*proof*⟩

lemma *comp-red-basis-adds*:
assumes $p \in set\ xs$ **and** $p \neq 0$
obtains q **where** $q \in set\ (comp\text{-}red\text{-}basis\ xs)$ **and** $lt\ q\ adds_t\ lt\ p$
⟨*proof*⟩

lemma *comp-red-basis-lt*:
assumes $p \in set\ (comp\text{-}red\text{-}basis\ xs)$
obtains q **where** $q \in set\ xs$ **and** $q \neq 0$ **and** $lt\ q = lt\ p$
⟨*proof*⟩

lemma *comp-red-basis-is-red*: $is\text{-}red\ (set\ (comp\text{-}red\text{-}basis\ xs))\ f \longleftrightarrow is\text{-}red\ (set\ xs)\ f$
⟨*proof*⟩

lemma *comp-red-basis-is-auto-reduced*: $is\text{-}auto\text{-}reduced\ (set\ (comp\text{-}red\text{-}basis\ xs))$
⟨*proof*⟩

lemma *comp-red-basis-dgrad-p-set-le*:
assumes $dickson\text{-}grading\ d$
shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ (set\ (comp\text{-}red\text{-}basis\ xs))\ (set\ xs)$
⟨*proof*⟩

12.5 Auto-Reduction and Monicity

definition $comp\text{-}red\text{-}monic\text{-}basis :: ('t \Rightarrow_0 'b)\ list \Rightarrow ('t \Rightarrow_0 'b::field)\ list$ **where**
$comp\text{-}red\text{-}monic\text{-}basis\ xs = map\ monic\ (comp\text{-}red\text{-}basis\ xs)$

lemma *set-comp-red-monic-basis*: $set\ (comp\text{-}red\text{-}monic\text{-}basis\ xs) = monic\ `\ (set\ (comp\text{-}red\text{-}basis\ xs))$
⟨*proof*⟩

lemma *comp-red-monic-basis-nonzero*:
assumes $p \in set\ (comp\text{-}red\text{-}monic\text{-}basis\ xs)$
shows $p \neq 0$
⟨*proof*⟩

lemma *comp-red-monic-basis-is-monic-set*: $is\text{-}monic\text{-}set\ (set\ (comp\text{-}red\text{-}monic\text{-}basis\ xs))$
⟨*proof*⟩

lemma *pmdl-comp-red-monic-basis-subset*: $pmdl\ (set\ (comp\text{-}red\text{-}monic\text{-}basis\ xs))$

$\subseteq$ *pmdl* (*set xs*)
⟨*proof*⟩

lemma *comp-red-monic-basis-is-auto-reduced*: *is-auto-reduced* (*set* (*comp-red-monic-basis xs*))
⟨*proof*⟩

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*⟩

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*) $\longleftrightarrow$ *is-Groebner-basis G*
⟨*proof*⟩

13.1 Definition and Uniqueness of Reduced Gröbner Bases

context *ordered-term*
begin

definition *is-reduced-GB* :: $('t \Rightarrow_0 'b::field)$ *set* $\Rightarrow$ *bool* **where**
is-reduced-GB B $\equiv$ *is-Groebner-basis B* $\wedge$ *is-auto-reduced B* $\wedge$ *is-monic-set B* $\wedge$ $0 \notin B$

lemma *reduced-GB-D1*:
assumes *is-reduced-GB G*
shows *is-Groebner-basis G*
⟨*proof*⟩

lemma *reduced-GB-D2*:
assumes *is-reduced-GB G*
shows *is-auto-reduced G*
⟨*proof*⟩

lemma *reduced-GB-D3*:
assumes *is-reduced-GB G*
shows *is-monic-set G*
⟨*proof*⟩

lemma *reduced-GB-D4*:
assumes *is-reduced-GB G* **and** $g \in G$
shows $g \neq 0$
⟨*proof*⟩

lemma *reduced-GB-lc*:
assumes *major*: *is-reduced-GB G* **and** $g \in G$
shows *lc* $g = 1$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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 $B = G$
⟨*proof*⟩

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*) = -)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *comp-red-basis-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-Groebner-basis* (*set* (*comp-red-basis xs*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *comp-red-monic-basis-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-Groebner-basis* (*set* (*comp-red-monic-basis xs*))
⟨*proof*⟩

lemma *comp-red-monic-basis-is-reduced-GB*:
assumes *is-Groebner-basis* (*set xs*)
shows *is-reduced-GB* (*set* (*comp-red-monic-basis xs*))
⟨*proof*⟩

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*⟩

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 $\exists!G.\ G \subseteq$ *dgrad-p-set d m* $\wedge$ *finite G* $\wedge$ *is-reduced-GB G* $\wedge$ *pmdl G* $=$ *pmdl F*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reduced-GB-is-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-Groebner-basis* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-auto-reduced-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ` *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *is-auto-reduced* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-monic-set-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ` *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *is-monic-set* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-nonzero-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows $0 \notin$ *reduced-GB F*
⟨*proof*⟩

lemma *reduced-GB-pmdl-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
shows *pmdl* (*reduced-GB F*) = *pmdl F*
⟨*proof*⟩

lemma *reduced-GB-unique-dgrad-p-set*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *Keys F*) **and** $F \subseteq$ *dgrad-p-set d m*
and *is-reduced-GB G* **and** *pmdl G* = *pmdl F*
shows *reduced-GB F* = *G*
⟨*proof*⟩

lemma *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 *reduced-GB F* $\subseteq$ *dgrad-p-set d m*
⟨*proof*⟩

lemma *reduced-GB-unique*:
assumes *finite G* **and** *is-reduced-GB G* **and** *pmdl G* = *pmdl F*
shows *reduced-GB F* = *G*
⟨*proof*⟩

lemma *is-reduced-GB-empty*: *is-reduced-GB* {}
⟨*proof*⟩

lemma *is-reduced-GB-singleton*: *is-reduced-GB* {*f*} $\longleftrightarrow$ *lc f* = *1*
⟨*proof*⟩

lemma *reduced-GB-empty*: *reduced-GB* {} = {}
⟨*proof*⟩

lemma *reduced-GB-singleton*: *reduced-GB* {*f*} = (*if f* = *0 then* {} *else* {*monic f*})
⟨*proof*⟩

lemma *ex-unique-reduced-GB-finite*: *finite F* $\Longrightarrow$ ($\exists!$*G*. *finite G* $\wedge$ *is-reduced-GB G* $\wedge$ *pmdl G* = *pmdl F*)
⟨*proof*⟩

lemma *finite-reduced-GB-finite*: *finite F* $\Longrightarrow$ *finite* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-reduced-GB-finite*: *finite F* $\Longrightarrow$ *is-reduced-GB* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-GB-finite*: *finite F* $\Longrightarrow$ *is-Groebner-basis* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-auto-reduced-finite*: *finite F* $\Longrightarrow$ *is-auto-reduced* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-is-monic-set-finite*: *finite F* $\Longrightarrow$ *is-monic-set* (*reduced-GB F*)
⟨*proof*⟩

lemma *reduced-GB-nonzero-finite*: *finite F* $\Longrightarrow$ *0* $\notin$ *reduced-GB F*
⟨*proof*⟩

lemma *reduced-GB-pmdl-finite*: *finite F* $\Longrightarrow$ *pmdl* (*reduced-GB F*) = *pmdl F*
⟨*proof*⟩

lemma *reduced-GB-unique-finite*: *finite F* $\Longrightarrow$ *is-reduced-GB G* $\Longrightarrow$ *pmdl G* = *pmdl F* $\Longrightarrow$ *reduced-GB F* = *G*
⟨*proof*⟩

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* $\subseteq$ *punit.dgrad-p-set d m*
shows *ideal F* = *UNIV* $\longleftrightarrow$ *punit.reduced-GB F* = {*1*}
⟨*proof*⟩

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* ⇒ ($'t \Rightarrow_0 'b$::*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* ⇒ ($'a \Rightarrow_0 'b$::*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*
)
⟨*proof*⟩

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* = ($\lambda x.\ x$)
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*
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*

⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *compute-trd-punit* [*code*]: *trd-punit to fs p* = *trd-aux-punit to fs p* (*change-ord to 0*)
⟨*proof*⟩

experiment begin interpretation $trivariate_0$*-rat* ⟨*proof*⟩

lemma
rgb-punit DRLEX
[
X ^ *3* − $X * Y * Z^2$,
$Y^2 * Z$ − *1*
] =
[
X ^ *3* ∗ $Y - X * Z$,
− (X ^ *3*) + $X * Y * Z^2$,
$Y^2 * Z$ − *1*,
− ($X * Z$ ^ *3*) + X ^ *5*
]
⟨*proof*⟩

lemma
rgb-punit DRLEX
[
$X^2 + Y^2 + Z^2$ − *1*,
$X * Y - Z$ − *1*,
$Y^2 + X$,
$Z^2 + X$
] =
[
1
]
⟨*proof*⟩

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*: $\textit{vec-of-list}\ xs = \textit{vec}\ (\textit{length}\ xs)\ (\textit{nth}\ xs)$
⟨*proof*⟩

lemma *vec-cong*:
assumes $n = m$ **and** $\bigwedge i.\ i < m \Longrightarrow f\ i = g\ i$
shows $\textit{vec}\ n\ f = \textit{vec}\ m\ g$
⟨*proof*⟩

lemma *scalar-prod-comm*:
assumes $\textit{dim-vec}\ v = \textit{dim-vec}\ w$
shows $v \cdot w = w \cdot (v{::}'a{::}\textit{comm-semiring-0 vec})$
⟨*proof*⟩

lemma *vec-scalar-mult-fun*: $\textit{vec}\ n\ (\lambda x.\ c * f\ x) = c \cdot_v \textit{vec}\ n\ f$
⟨*proof*⟩

definition *mult-vec-mat* :: $'a\ \textit{vec} \Rightarrow 'a :: \textit{semiring-0 mat} \Rightarrow 'a\ \textit{vec}$ (**infixl** ‹$_v*$› *70*)
where $v\ _v{*}\ A \equiv \textit{vec}\ (\textit{dim-col}\ A)\ (\lambda j.\ v \cdot \textit{col}\ A\ j)$

definition *resize-vec* :: $\textit{nat} \Rightarrow 'a\ \textit{vec} \Rightarrow 'a\ \textit{vec}$
where $\textit{resize-vec}\ n\ v = \textit{vec}\ n\ (\textit{vec-index}\ v)$

lemma *dim-resize-vec*[*simp*]: $\textit{dim-vec}\ (\textit{resize-vec}\ n\ v) = n$
⟨*proof*⟩

lemma *resize-vec-carrier*: $\textit{resize-vec}\ n\ v \in \textit{carrier-vec}\ n$
⟨*proof*⟩

lemma *resize-vec-dim*[*simp*]: $\textit{resize-vec}\ (\textit{dim-vec}\ v)\ v = v$
⟨*proof*⟩

lemma *resize-vec-index*:
assumes $i < n$
shows $\textit{resize-vec}\ n\ v\ \$\ i = v\ \$\ i$
⟨*proof*⟩

lemma *mult-mat-vec-resize*:
  $v \ {}_v* \ A = (\textit{resize-vec} \ (\textit{dim-row} \ A) \ v) \ {}_v* \ A$
  ⟨*proof*⟩

lemma *assoc-mult-vec-mat*:
  assumes $v \in \textit{carrier-vec} \ n1$ **and** $A \in \textit{carrier-mat} \ n1 \ n2$ **and** $B \in \textit{carrier-mat}$ $n2 \ n3$
  shows $v \ {}_v* \ (A * B) = (v \ {}_v* \ A) \ {}_v* \ B$
  ⟨*proof*⟩

lemma *mult-vec-mat-transpose*:
  assumes $\textit{dim-vec} \ v = \textit{dim-row} \ A$
  shows $v \ {}_v* \ A = (\textit{transpose-mat} \ A) *_v (v::'a::\textit{comm-semiring-0 vec})$
⟨*proof*⟩

15.2 More about Matrices

definition $\textit{nzrows} :: 'a::\textit{zero mat} \Rightarrow 'a \ \textit{vec list}$
  where $\textit{nzrows} \ A = \textit{filter} \ (\lambda r. \ r \neq 0_v \ (\textit{dim-col} \ A)) \ (\textit{rows} \ A)$

definition $\textit{row-space} :: 'a \ \textit{mat} \Rightarrow 'a::\textit{semiring-0 vec set}$
  where $\textit{row-space} \ A = (\lambda v. \ \textit{mult-vec-mat} \ v \ A) \ ` \ (\textit{carrier-vec} \ (\textit{dim-row} \ A))$

definition $\textit{row-echelon} :: 'a \ \textit{mat} \Rightarrow 'a::\textit{field mat}$
  where $\textit{row-echelon} \ A = \textit{fst} \ (\textit{gauss-jordan} \ A \ (1_m \ (\textit{dim-row} \ A)))$

15.2.1 *nzrows*

lemma *length-nzrows*: $\textit{length} \ (\textit{nzrows} \ A) \leq \textit{dim-row} \ A$
  ⟨*proof*⟩

lemma *set-nzrows*: $\textit{set} \ (\textit{nzrows} \ A) = \textit{set} \ (\textit{rows} \ A) - \{0_v \ (\textit{dim-col} \ A)\}$
  ⟨*proof*⟩

lemma *nzrows-nth-not-zero*:
  assumes $i < \textit{length} \ (\textit{nzrows} \ A)$
  shows $\textit{nzrows} \ A \ ! \ i \neq 0_v \ (\textit{dim-col} \ A)$
  ⟨*proof*⟩

15.2.2 *row-space*

lemma *row-spaceI*:
  assumes $x = v \ {}_v* \ A$
  shows $x \in \textit{row-space} \ A$
  ⟨*proof*⟩

lemma *row-spaceE*:
  assumes $x \in \textit{row-space} \ A$
  obtains v **where** $v \in \textit{carrier-vec} \ (\textit{dim-row} \ A)$ **and** $x = v \ {}_v* \ A$
  ⟨*proof*⟩

lemma *row-space-alt*: *row-space* A = *range* (λv. *mult-vec-mat* v A)
⟨*proof*⟩

lemma *row-space-mult*:
  assumes $A \in$ *carrier-mat nr nc* **and** $B \in$ *carrier-mat nr nr*
  shows *row-space* ($B * A$) $\subseteq$ *row-space* A
⟨*proof*⟩

lemma *row-space-mult-unit*:
  assumes $P \in$ *Units* (*ring-mat TYPE*('*a*::*semiring-1*) (*dim-row* A) b)
  shows *row-space* ($P * A$) = *row-space* A
⟨*proof*⟩

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)) $\longleftrightarrow$ (f i = *dim-col* A)
⟨*proof*⟩

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 $i \neq 0_v$ (*dim-col* A)) $\longleftrightarrow$ (f $i <$ *dim-col* A)
⟨*proof*⟩

lemma *pivot-fun-stabilizes*:
  assumes *pivot-fun* A f *nc* **and** *i1* $\leq$ *i2* **and** *i2* $<$ *dim-row* A **and** *nc* $\leq f$ *i1*
  shows f *i2* = *nc*
⟨*proof*⟩

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*⟩

lemma *pivot-fun-mono*:
  assumes *pivot-fun* A f *nc* **and** *i1* $\leq$ *i2* **and** *i2* $<$ *dim-row* A
  shows f *i1* $\leq f$ *i2*
⟨*proof*⟩

lemma *row-echelon-carrier*:
  assumes $A \in$ *carrier-mat nr nc*
  shows *row-echelon* $A \in$ *carrier-mat nr nc*
⟨*proof*⟩

lemma *dim-row-echelon*[*simp*]:
  shows *dim-row* (*row-echelon* A) = *dim-row* A **and** *dim-col* (*row-echelon* A) = *dim-col* A

⟨*proof*⟩

lemma *row-echelon-transform*:
obtains P **where** $P \in$ *Units* (*ring-mat TYPE*('*a*::*field*) (*dim-row A*) *b*) **and** *row-echelon* $A = P * A$
⟨*proof*⟩

lemma *row-space-row-echelon*[*simp*]: *row-space* (*row-echelon A*) = *row-space A*
⟨*proof*⟩

lemma *row-echelon-pivot-fun*:
obtains *f* **where** *pivot-fun* (*row-echelon A*) *f* (*dim-col* (*row-echelon A*))
⟨*proof*⟩

lemma *distinct-nzrows-row-echelon*: *distinct* (*nzrows* (*row-echelon A*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *poly-to-row-index*:
assumes $i <$ *length ts*
shows *poly-to-row ts p* \$ *i* = *lookup p* (*ts* ! *i*)
⟨*proof*⟩

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*)) = $c \cdot p$
⟨*proof*⟩

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*⟩

lemma *lookup-list-to-poly*: *lookup* (*list-to-poly ts cs*) = *list-to-fun ts cs*
  ⟨*proof*⟩

lemma *list-to-fun-Nil* [*simp*]: *list-to-fun* [] *cs* = *0*
  ⟨*proof*⟩

lemma *list-to-poly-Nil* [*simp*]: *list-to-poly* [] *cs* = *0*
  ⟨*proof*⟩

lemma *row-to-poly-Nil* [*simp*]: *row-to-poly* [] *r* = *0*
  ⟨*proof*⟩

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*⟩

lemma *keys-row-to-poly*: *keys* (*row-to-poly ts r*) ⊆ *set ts*
⟨*proof*⟩

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*⟩

lemma *row-to-poly-zero* [*simp*]: *row-to-poly ts* (0_v (*length ts*)) = (0::$'t \Rightarrow_0 'b$::*zero*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *row-to-poly-vec-sum*:
assumes *distinct ts* **and** *length ts* = *n*
shows *row-to-poly ts* (*vec n* ($\lambda j.\ \sum i \in I.\ f\ i\ j$)) = (($\sum i \in I.$ *row-to-poly ts* (*vec n* (*f i*)))::$'t \Rightarrow_0$ $'b$::*comm-monoid-add*)
⟨*proof*⟩

lemma *row-to-poly-smult*:
assumes *distinct ts* **and** *dim-vec r* = *length ts*
shows *row-to-poly ts* ($c \cdot_v r$) = $c \cdot$ (*row-to-poly ts r*)
⟨*proof*⟩

lemma *poly-to-row-Nil* [*simp*]: *poly-to-row* [] *p* = *vec 0 f*
⟨*proof*⟩

lemma *polys-to-mat-Nil* [*simp*]: *polys-to-mat ts* [] = *mat 0* (*length ts*) *f*
⟨*proof*⟩

lemma *dim-row-polys-to-mat*[*simp*]: *dim-row* (*polys-to-mat ts ps*) = *length ps*
⟨*proof*⟩

lemma *dim-col-polys-to-mat*[*simp*]: *dim-col* (*polys-to-mat ts ps*) = *length ts*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *row-polys-to-mat*:
assumes *i* < *length ps*
shows *row* (*polys-to-mat ts ps*) *i* = *poly-to-row ts* (*ps* ! *i*)
⟨*proof*⟩

lemma *col-polys-to-mat*:
assumes *j* < *length ts*
shows *col* (*polys-to-mat ts ps*) *j* = *vec-of-list* (*map* ($\lambda p.$ *lookup p* (*ts* ! *j*)) *ps*)
⟨*proof*⟩

lemma *length-mat-to-polys*[*simp*]: *length* (*mat-to-polys ts A*) = *dim-row A*
⟨*proof*⟩

lemma *mat-to-polys-nth*:
assumes *i* < *dim-row A*

shows $(\textit{mat-to-polys ts A})\ !\ i = \textit{row-to-poly ts}\ (\textit{row A i})$
⟨*proof*⟩

lemma *Keys-mat-to-polys*: $\textit{Keys}\ (\textit{set}\ (\textit{mat-to-polys ts A})) \subseteq \textit{set ts}$
⟨*proof*⟩

lemma *polys-to-mat-to-polys*:
assumes $\textit{Keys}\ (\textit{set ps}) \subseteq \textit{set ts}$
shows $\textit{mat-to-polys ts}\ (\textit{polys-to-mat ts ps}) = (\textit{ps}::('t \Rightarrow_0 'b::\textit{semiring-1})\ \textit{list})$
⟨*proof*⟩

lemma *mat-to-polys-to-mat*:
assumes *distinct ts* **and** $\textit{length ts} = \textit{dim-col A}$
shows $(\textit{polys-to-mat ts}\ (\textit{mat-to-polys ts A})) = A$
⟨*proof*⟩

15.4 Properties of Macaulay Matrices

lemma *row-to-poly-vec-times*:
assumes *distinct ts* **and** $\textit{length ts} = \textit{dim-col A}$
shows $\textit{row-to-poly ts}\ (v\ {}_v{*}\ A) = ((\sum i=0..<\textit{dim-row A}.\ (v\ \$\ i) \cdot (\textit{row-to-poly ts}\ (\textit{row A i})))::'t \Rightarrow_0 'b::\textit{comm-semiring-0})$
⟨*proof*⟩

lemma *vec-times-polys-to-mat*:
assumes $\textit{Keys}\ (\textit{set ps}) \subseteq \textit{set ts}$ **and** $v \in \textit{carrier-vec}\ (\textit{length ps})$
shows $\textit{row-to-poly ts}\ (v\ {}_v{*}\ (\textit{polys-to-mat ts ps})) = (\sum (c,\ p) \leftarrow \textit{zip}\ (\textit{list-of-vec}\ v)\ \textit{ps}.\ c \cdot p)$
(**is** $?l = ?r$)
⟨*proof*⟩

lemma *row-space-subset-phull*:
assumes $\textit{Keys}\ (\textit{set ps}) \subseteq \textit{set ts}$
shows $\textit{row-to-poly ts}\ \text{‘}\ \textit{row-space}\ (\textit{polys-to-mat ts ps}) \subseteq \textit{phull}\ (\textit{set ps})$
(**is** $?r \subseteq ?h$)
⟨*proof*⟩

lemma *phull-subset-row-space*:
assumes $\textit{Keys}\ (\textit{set ps}) \subseteq \textit{set ts}$
shows $\textit{phull}\ (\textit{set ps}) \subseteq \textit{row-to-poly ts}\ \text{‘}\ \textit{row-space}\ (\textit{polys-to-mat ts ps})$
(**is** $?h \subseteq ?r$)
⟨*proof*⟩

lemma *row-space-eq-phull*:
assumes $\textit{Keys}\ (\textit{set ps}) \subseteq \textit{set ts}$
shows $\textit{row-to-poly ts}\ \text{‘}\ \textit{row-space}\ (\textit{polys-to-mat ts ps}) = \textit{phull}\ (\textit{set ps})$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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 $(pps\text{-}to\text{-}list\ S)\ !\ (f\ i1) = (pps\text{-}to\text{-}list\ S)\ !\ (f\ i2)$
shows $i1 = i2$
⟨*proof*⟩

lemma *lt-row-to-poly-pivot-in-keysD*:
assumes $card\ S = dim\text{-}col\ (A::'b::semiring\text{-}1\ mat)$ **and** $pivot\text{-}fun\ A\ f\ (dim\text{-}col\ A)$
and $i1 < dim\text{-}row\ A$ **and** $i2 < dim\text{-}row\ A$ **and** $f\ i1 < dim\text{-}col\ A$
and $(pps\text{-}to\text{-}list\ S)\ !\ (f\ i1) \in keys\ ((mat\text{-}to\text{-}polys\ (pps\text{-}to\text{-}list\ S)\ A)\ !\ i2)$
shows $i1 = i2$
⟨*proof*⟩

lemma *lt-row-space-pivot-fun*:
assumes $card\ S = dim\text{-}col\ (A::'b::\{comm\text{-}semiring\text{-}0,semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors\}\ mat)$
and $pivot\text{-}fun\ A\ f\ (dim\text{-}col\ A)$ **and** $p \in row\text{-}to\text{-}poly\ (pps\text{-}to\text{-}list\ S)$ ‘ $row\text{-}space\ A$ **and** $p \neq 0$
shows $lt\ p \in lt\text{-}set\ (set\ (mat\text{-}to\text{-}polys\ (pps\text{-}to\text{-}list\ S)\ A))$
⟨*proof*⟩

15.5 Functions *Macaulay-mat* and *Macaulay-list*

definition $Macaulay\text{-}mat :: ('t \Rightarrow_0 'b)\ list \Rightarrow 'b::field\ mat$
where $Macaulay\text{-}mat\ ps = polys\text{-}to\text{-}mat\ (Keys\text{-}to\text{-}list\ ps)\ ps$

definition $Macaulay\text{-}list :: ('t \Rightarrow_0 'b)\ list \Rightarrow ('t \Rightarrow_0 'b::field)\ list$
where $Macaulay\text{-}list\ ps =$
$filter\ (\lambda p.\ p \neq 0)\ (mat\text{-}to\text{-}polys\ (Keys\text{-}to\text{-}list\ ps)\ (row\text{-}echelon\ (Macaulay\text{-}mat\ ps)))$

lemma *dim-Macaulay-mat*[*simp*]:
$dim\text{-}row\ (Macaulay\text{-}mat\ ps) = length\ ps$
$dim\text{-}col\ (Macaulay\text{-}mat\ ps) = card\ (Keys\ (set\ ps))$
⟨*proof*⟩

lemma *Macaulay-list-Nil* [*simp*]: $Macaulay\text{-}list\ [] = ([]::('t \Rightarrow_0 'b::field)\ list)$ (**is** *?l* = -)
⟨*proof*⟩

lemma *set-Macaulay-list*:
$set\ (Macaulay\text{-}list\ ps) =$
$set\ (mat\text{-}to\text{-}polys\ (Keys\text{-}to\text{-}list\ ps)\ (row\text{-}echelon\ (Macaulay\text{-}mat\ ps))) - \{0\}$
⟨*proof*⟩

lemma *Keys-Macaulay-list*: $Keys\ (set\ (Macaulay\text{-}list\ ps)) \subseteq Keys\ (set\ ps)$
⟨*proof*⟩

lemma *in-Macaulay-listE*:
assumes $p \in set\ (Macaulay\text{-}list\ ps)$

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*⟩

lemma *phull-Macaulay-list*: *phull* (*set* (*Macaulay-list ps*)) = *phull* (*set ps*)
⟨*proof*⟩

lemma *pmdl-Macaulay-list*: *pmdl* (*set* (*Macaulay-list ps*)) = *pmdl* (*set ps*)
⟨*proof*⟩

lemma *Macaulay-list-is-monic-set*: *is-monic-set* (*set* (*Macaulay-list ps*))
⟨*proof*⟩

lemma *Macaulay-list-not-zero*: *0* $\notin$ *set* (*Macaulay-list ps*)
⟨*proof*⟩

lemma *Macaulay-list-distinct-lt*:
assumes $x \in$ *set* (*Macaulay-list ps*) **and** $y \in$ *set* (*Macaulay-list ps*)
and $x \neq y$
shows *lt x* $\neq$ *lt y*
⟨*proof*⟩

lemma *Macaulay-list-lt*:
assumes $p \in$ *phull* (*set ps*) **and** $p \neq 0$
obtains *g* **where** $g \in$ *set* (*Macaulay-list ps*) **and** $g \neq 0$ **and** *lt p* = *lt g*
⟨*proof*⟩

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* :: $('a \Rightarrow nat) \Rightarrow ((('t \Rightarrow_0 'b)\ list \times 't\ list \times 't\ list \times ('t \Rightarrow_0 'b)\ list) \times$

$((\prime t \Rightarrow_0 \prime b)$ *list* $\times$ $\prime t$ *list* $\times$ $\prime t$ *list* $\times$ $(\prime t \Rightarrow_0$ $\prime b)$ *list*)) *set*
where *sym-preproc-aux-term1 d* =
$\{((gs1, ks1, ts1, fs1), (gs2::(\prime t \Rightarrow_0 \prime b)\ list, ks2, ts2, fs2)).\ \exists\, t2 \in set\ ts2.\ \forall\, t1 \in set\ ts1.\ t1 \prec_t t2\}$

definition *sym-preproc-aux-term2* :: $(\prime a \Rightarrow nat) \Rightarrow (((\prime t \Rightarrow_0 \prime b{::}zero)\ list \times \prime t\ list \times \prime t\ list \times (\prime t \Rightarrow_0 \prime b)\ list) \times ((\prime t \Rightarrow_0 \prime b)\ list \times \prime t\ list \times \prime t\ list \times (\prime t \Rightarrow_0 \prime b)\ list))\ set$
where *sym-preproc-aux-term2 d* =
$\{((gs1, ks1, ts1, fs1), (gs2::(\prime t \Rightarrow_0 \prime b)\ list, ks2, ts2, fs2)).\ gs1 = gs2 \wedge dgrad\text{-}set\text{-}le\ d\ (pp\text{-}of\text{-}term\ `\ set\ ts1)\ (pp\text{-}of\text{-}term\ `\ (Keys\ (set\ gs2) \cup set\ ts2))\}$

definition *sym-preproc-aux-term*
where *sym-preproc-aux-term d* = *sym-preproc-aux-term1 d* $\cap$ *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*⟩

lemma *sym-preproc-aux-term1-wf-on*:
assumes *dickson-grading d*
shows *wfp-on* $(\lambda x\ y.\ (x, y) \in sym\text{-}preproc\text{-}aux\text{-}term1\ d)\ \{x.\ set\ (fst\ (snd\ (snd\ x))) \subseteq pp\text{-}of\text{-}term\ -`\ dgrad\text{-}set\ d\ m\}$
⟨*proof*⟩

lemma *sym-preproc-aux-term-wf*:
assumes *dickson-grading d*
shows *wf* (*sym-preproc-aux-term d*)
⟨*proof*⟩

primrec *sym-preproc-addnew* :: $(\prime t \Rightarrow_0 \prime b{::}semiring\text{-}1)\ list \Rightarrow \prime t\ list \Rightarrow (\prime t \Rightarrow_0 \prime b)\ list \Rightarrow \prime t \Rightarrow (\prime t\ list \times (\prime t \Rightarrow_0 \prime b)\ list)$ **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 $u \in set\ (fst\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$
shows $u \prec_t v$
⟨*proof*⟩

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*⟩

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*⟩

lemma *fst-sym-preproc-addnew-superset*: *set vs* ⊆ *set* (*fst* (*sym-preproc-addnew gs vs fs v*))
⟨*proof*⟩

lemma *snd-sym-preproc-addnew-superset*: *set fs* ⊆ *set* (*snd* (*sym-preproc-addnew gs vs fs v*))
⟨*proof*⟩

lemma *in-snd-sym-preproc-addnewE*:
assumes $p \in set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$
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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *sym-preproc-addnew-complete*:
assumes $g \in set\ gs$ **and** $lt\ g\ adds_t\ v$
shows $monom\text{-}mult\ 1\ (pp\text{-}of\text{-}term\ v - lp\ g)\ g \in set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$
⟨*proof*⟩

function *sym-preproc-aux* :: $('t \Rightarrow_0 'b::semiring\text{-}1)$ *list* $\Rightarrow$ $'t$ *list* $\Rightarrow$ $('t$ *list* $\times$ $('t \Rightarrow_0 'b)$ *list*$)$ $\Rightarrow$
$('t$ *list* $\times$ $('t \Rightarrow_0 'b)$ *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*)
)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *sym-preproc-aux-Nil*: *sym-preproc-aux gs ks* ([], *fs*) = (*ks*, *fs*)
⟨*proof*⟩

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*⟩

lemma *sym-preproc-aux-induct* [*consumes 0*, *case-names base rec*]:
assumes *base*: $\bigwedge$*ks fs*. *P ks* [] *fs* (*ks*, *fs*)
and *rec*: $\bigwedge$*ks vs fs v vs'*. *vs* $\neq$ [] $\Longrightarrow$ *v* = *ord-term-lin.Max* (*set vs*) $\Longrightarrow$ *vs'* = *removeAll v vs* $\Longrightarrow$
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*))
$\Longrightarrow$
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*⟩

lemma *fst-sym-preproc-aux-sorted-wrt*:
assumes *sorted-wrt* $(\succ_t)$ *ks* **and** $\bigwedge$*k v*. *k* $\in$ *set ks* $\Longrightarrow$ *v* $\in$ *set vs* $\Longrightarrow$ *v* $\prec_t$ *k*
shows *sorted-wrt* $(\succ_t)$ (*fst* (*sym-preproc-aux gs ks* (*vs*, *fs*)))
⟨*proof*⟩

lemma *fst-sym-preproc-aux-complete*:
assumes *Keys* (*set* (*fs*::$('t \Rightarrow_0 'b::semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)$ *list*)) = *set ks* $\cup$ *set vs*
shows *set* (*fst* (*sym-preproc-aux gs ks* (*vs*, *fs*))) = *Keys* (*set* (*snd* (*sym-preproc-aux gs ks* (*vs*, *fs*))))
⟨*proof*⟩

lemma *snd-sym-preproc-aux-superset*: *set fs* $\subseteq$ *set* (*snd* (*sym-preproc-aux gs ks* (*vs*,

fs)))
⟨*proof*⟩

lemma *in-snd-sym-preproc-auxE*:
assumes $p \in set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs)))$
assumes *1*: $p \in set\ fs \Longrightarrow thesis$
assumes *2*: $\bigwedge g\ t.\ g \in set\ gs \Longrightarrow p = monom\text{-}mult\ 1\ t\ g \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *snd-sym-preproc-aux-pmdl*:
$pmdl\ (set\ gs \cup set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (ts,\ fs)))) = pmdl\ (set\ gs \cup set\ fs)$
⟨*proof*⟩

lemma *snd-sym-preproc-aux-dgrad-set-le*:
assumes *dickson-grading d* **and** $set\ vs \subseteq Keys\ (set\ (fs{::}('t \Rightarrow_0 'b{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)\ list))$
shows $dgrad\text{-}set\text{-}le\ d\ (pp\text{-}of\text{-}term\ `\ Keys\ (set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs)))))\ (pp\text{-}of\text{-}term\ `\ Keys\ (set\ gs \cup set\ fs))$
⟨*proof*⟩

lemma *components-snd-sym-preproc-aux-subset*:
assumes $set\ vs \subseteq Keys\ (set\ (fs{::}('t \Rightarrow_0 'b{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)\ list))$
shows $component\text{-}of\text{-}term\ `\ Keys\ (set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs)))) \subseteq$
$component\text{-}of\text{-}term\ `\ Keys\ (set\ gs \cup set\ fs)$
⟨*proof*⟩

lemma *snd-sym-preproc-aux-complete*:
assumes $\bigwedge u'\ g'.\ u' \in Keys\ (set\ fs) \Longrightarrow u' \notin set\ vs \Longrightarrow g' \in set\ gs \Longrightarrow lt\ g'\ adds_t\ u' \Longrightarrow$
$monom\text{-}mult\ 1\ (pp\text{-}of\text{-}term\ u' - lp\ g')\ g' \in set\ fs$
assumes $u \in Keys\ (set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs))))$ **and** $g \in set\ gs$ **and** $lt\ g\ adds_t\ u$
shows $monom\text{-}mult\ (1{::}'b{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)\ (pp\text{-}of\text{-}term\ u - lp\ g)\ g \in$
$set\ (snd\ (sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs)))$
⟨*proof*⟩

definition $sym\text{-}preproc :: ('t \Rightarrow_0 'b{::}semiring\text{-}1)\ list \Rightarrow ('t \Rightarrow_0 'b)\ list \Rightarrow ('t\ list \times ('t \Rightarrow_0 'b)\ list)$
where $sym\text{-}preproc\ gs\ fs = sym\text{-}preproc\text{-}aux\ gs\ []\ (Keys\text{-}to\text{-}list\ fs,\ fs)$

lemma *sym-preproc-Nil* [*simp*]: $sym\text{-}preproc\ gs\ [] = ([],\ [])$
⟨*proof*⟩

lemma *fst-sym-preproc*:
$fst\ (sym\text{-}preproc\ gs\ fs) = Keys\text{-}to\text{-}list\ (snd\ (sym\text{-}preproc\ gs\ (fs{::}('t \Rightarrow_0 'b{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)\ list)))$

⟨*proof*⟩

lemma *snd-sym-preproc-superset*: *set fs* ⊆ *set* (*snd* (*sym-preproc gs fs*))
⟨*proof*⟩

lemma *in-snd-sym-preprocE*:
assumes $p \in$ *set* (*snd* (*sym-preproc gs fs*))
assumes *1*: $p \in$ *set fs* $\Longrightarrow$ *thesis*
assumes *2*: $\bigwedge g\ t.\ g \in$ *set gs* $\Longrightarrow p =$ *monom-mult 1 t g* $\Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *snd-sym-preproc-pmdl*: *pmdl* (*set gs* ∪ *set* (*snd* (*sym-preproc gs fs*))) = *pmdl* (*set gs* ∪ *set fs*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *snd-sym-preproc-complete*:
assumes $v \in$ *Keys* (*set* (*snd* (*sym-preproc gs fs*))) **and** $g \in$ *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*))
⟨*proof*⟩

end

16.2 *lin-red*

context *ordered-term*
begin

definition *lin-red* :: $('t \Rightarrow_0 'b::field)$ *set* $\Rightarrow$ $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow$ *bool*
where *lin-red F p q* $\equiv$ ($\exists f \in F$. *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 $f \in F$ **and** *red-single p q f 0*
shows *lin-red F p q*
⟨*proof*⟩

lemma *lin-redE*:
assumes *lin-red F p q*
obtains $f::'t \Rightarrow_0 'b::field$ **where** $f \in F$ **and** *red-single p q f 0*
⟨*proof*⟩

lemma *lin-red-imp-red*:
assumes *lin-red F p q*
shows *red F p q*
⟨*proof*⟩

lemma *lin-red-Un*: *lin-red* $(F \cup G)$ *p q* = (*lin-red F p q* $\vee$ *lin-red G p q*)
⟨*proof*⟩

lemma *lin-red-imp-red-rtrancl*:
assumes $(\textit{lin-red } F)^{**}$ *p q*
shows $(\textit{red } F)^{**}$ *p q*
⟨*proof*⟩

lemma *phull-closed-lin-red*:
assumes *phull B* $\subseteq$ *phull A* **and** $p \in$ *phull A* **and** *lin-red B p q*
shows $q \in$ *phull A*
⟨*proof*⟩

16.3 Reduction

definition *Macaulay-red* :: *'t list* $\Rightarrow$ $('t \Rightarrow_0 'b)$ *list* $\Rightarrow$ $('t \Rightarrow_0 'b::field)$ *list*
where *Macaulay-red vs fs* =
(*let lts = map lt* (*filter* ($\lambda p.\ p \neq 0$) *fs*) *in*
filter ($\lambda p.\ p \neq 0 \wedge lt\ p \notin set\ lts$) (*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* (*λp. lt p* ∉ *lt-set* (*set fs*)) (*Macaulay-list fs*)
⟨*proof*⟩

lemma *set-Macaulay-red*:
set (*Macaulay-red* (*Keys-to-list fs*) *fs*) = *set* (*Macaulay-list fs*) − {*p. lt p* ∈ *lt-set* (*set fs*)}
⟨*proof*⟩

lemma *Keys-Macaulay-red*: *Keys* (*set* (*Macaulay-red* (*Keys-to-list fs*) *fs*)) ⊆ *Keys* (*set fs*)
⟨*proof*⟩

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*⟩

primrec *pdata-pairs-to-list* :: (*'t*, *'b::field*, *'c*) *pdata-pair list* ⇒ (*'t* $\Rightarrow_0$ *'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* ∈ -)
⟨*proof*⟩

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* ∈ -)
⟨*proof*⟩

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*)

⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *pmdl-f4-red-aux*: *set* (*f4-red-aux bs ps*) ⊆ *pmdl* (*args-to-set* ([], *bs*, *ps*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

definition *f4-red* :: (*'t*, *'b*::*field*, *'c*::*default*, *'d*) *complT*
where *f4-red gs bs ps sps data* = (*map* (λ*h*. (*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*)

⟨*proof*⟩

lemma *rcp-spec-f4-red*: *rcp-spec f4-red*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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*
⟨*proof*⟩

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* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'a\ list$ **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*: $v \in set\ vs \Longrightarrow v \notin f$ ‘ *set* (*remdups-wrt-rev f xs vs*)
⟨*proof*⟩

lemma *distinct-remdups-wrt-rev*: *distinct* (*map f* (*remdups-wrt-rev f xs vs*))
⟨*proof*⟩

lemma *map-of-remdups-wrt-rev'*:
map-of (*remdups-wrt-rev fst xs vs*) *k* = *map-of* (*filter* ($\lambda x.$ *fst* $x \notin$ *set vs*) *xs*) *k*
⟨*proof*⟩

corollary *map-of-remdups-wrt-rev*: *map-of* (*remdups-wrt-rev fst xs* []) = *map-of xs*
⟨*proof*⟩

lemma (**in** *term-powerprod*) *compute-list-to-poly* [*code*]:
list-to-poly ts cs = $distr_0$ *DRLEX* (*remdups-wrt-rev fst* (*zip ts cs*) [])
⟨*proof*⟩

lemma (**in** *ordered-term*) *compute-Macaulay-list* [*code*]:
Macaulay-list ps =
(*let ts* = *Keys-to-list ps in*
filter ($\lambda p.\ p \neq 0$) (*mat-to-polys ts* (*row-echelon* (*polys-to-mat ts ps*)))
)
⟨*proof*⟩

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*
⟨*proof*⟩

17.2 Computations

experiment begin interpretation $trivariate_0$*-rat* ⟨*proof*⟩

lemma
lt-punit DRLEX $(X^2 * Z \mathbin{\hat{}} 3 + 3 * X^2 * Y) = sparse_0\ [(0,\ 2),\ (2,\ 3)]$
⟨*proof*⟩

lemma
lc-punit DRLEX $(X^2 * Z \mathbin{\hat{}} 3 + 3 * X^2 * Y) = 1$
⟨*proof*⟩

lemma
tail-punit DRLEX $(X^2 * Z \mathbin{\hat{}} 3 + 3 * X^2 * Y) = 3 * X^2 * Y$
⟨*proof*⟩

lemma
ord-strict-p-punit DRLEX $(X^2 * Z \mathbin{\hat{}} 4 - 2 * Y \mathbin{\hat{}} 3 * Z^2)\ (X^2 * Z \mathbin{\hat{}} 7 + 2 * Y \mathbin{\hat{}} 3 * Z^2)$
⟨*proof*⟩

lemma
f4-punit DRLEX
[
$(X^2 * Z \mathbin{\hat{}} 4 - 2 * Y \mathbin{\hat{}} 3 * Z^2,\ ()),$
$(Y^2 * Z + 2 * Z \mathbin{\hat{}} 3,\ ())$
] () =
[
$(X^2 * Y^2 * Z^2 + 4 * Y \mathbin{\hat{}} 3 * Z^2,\ ()),$
$(X^2 * Z \mathbin{\hat{}} 4 - 2 * Y \mathbin{\hat{}} 3 * Z^2,\ ()),$
$(Y^2 * Z + 2 * Z \mathbin{\hat{}} 3,\ ()),$
$(X^2 * Y \mathbin{\hat{}} 4 * Z + 4 * Y \mathbin{\hat{}} 5 * Z,\ ())$
]
⟨*proof*⟩

lemma
f4-punit DRLEX
[
$(X^2 + Y^2 + Z^2 - 1,\ ()),$
$(X * Y - Z - 1,\ ()),$
$(Y^2 + X,\ ()),$
$(Z^2 + X,\ ())$
] () =
[
$(1,\ ())$
]

$\langle proof \rangle$

end

value [*code*] *length* (*f4-punit DRLEX* (*map* ($\lambda p.$ (*p*, ())) ((*cyclic DRLEX 4*)::(- $\Rightarrow_0$ *rat*) *list*)) ())

value [*code*] *length* (*f4-punit DRLEX* (*map* ($\lambda p.$ (*p*, ())) ((*katsura DRLEX 2*)::(- $\Rightarrow_0$ *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* $\langle proof \rangle$

18.1 Syzygy Modules Generated by Sets

context *module*
begin

definition *rep* :: $('b \Rightarrow_0 'a) \Rightarrow 'b$
where $rep\ r = (\sum v{\in}keys\ r.\ lookup\ r\ v *s\ v)$

definition *represents* :: $'b\ set \Rightarrow ('b \Rightarrow_0 'a) \Rightarrow 'b \Rightarrow bool$
where $represents\ B\ r\ x \longleftrightarrow (keys\ r \subseteq B \wedge local.rep\ r = x)$

definition *syzygy-module* :: $'b\ set \Rightarrow ('b \Rightarrow_0 'a)\ set$
where $syzygy\text{-}module\ B = \{s.\ local.represents\ B\ s\ 0\}$

end

hide-const (**open**) *real-vector.rep real-vector.represents real-vector.syzygy-module*

context *module*
begin

lemma *rep-monomial* [*simp*]: $rep\ (monomial\ c\ x) = c *s\ x$
$\langle proof \rangle$

lemma *rep-zero* [*simp*]: *rep 0 = 0*
⟨*proof*⟩

lemma *rep-uminus* [*simp*]: *rep* (− *r*) = − *rep r*
⟨*proof*⟩

lemma *rep-plus*: *rep* (*r* + *s*) = *rep r* + *rep s*
⟨*proof*⟩

lemma *rep-minus*: *rep* (*r* − *s*) = *rep r* − *rep s*
⟨*proof*⟩

lemma *rep-smult*: *rep* (*monomial c 0* ∗ *r*) = *c* ∗*s rep r*
⟨*proof*⟩

lemma *rep-in-span*: *rep r* ∈ *span* (*keys r*)
⟨*proof*⟩

lemma *spanE-rep*:
assumes *x* ∈ *span B*
obtains *r* **where** *keys r* ⊆ *B* **and** *x* = *rep r*
⟨*proof*⟩

lemma *representsI*:
assumes *keys r* ⊆ *B* **and** *rep r* = *x*
shows *represents B r x*
⟨*proof*⟩

lemma *representsD1*:
assumes *represents B r x*
shows *keys r* ⊆ *B*
⟨*proof*⟩

lemma *representsD2*:
assumes *represents B r x*
shows *x* = *rep r*
⟨*proof*⟩

lemma *represents-mono*:
assumes *represents B r x* **and** *B* ⊆ *A*
shows *represents A r x*
⟨*proof*⟩

lemma *represents-self*: *represents* {*x*} (*monomial 1 x*) *x*
⟨*proof*⟩

lemma *represents-zero*: *represents B 0 0*
⟨*proof*⟩

lemma *represents-plus*:
assumes *represents A r x* **and** *represents B s y*
shows *represents* $(A \cup B)$ $(r + s)$ $(x + y)$
⟨*proof*⟩

lemma *represents-uminus*:
assumes *represents B r x*
shows *represents B* $(- r)$ $(- x)$
⟨*proof*⟩

lemma *represents-minus*:
assumes *represents A r x* **and** *represents B s y*
shows *represents* $(A \cup B)$ $(r - s)$ $(x - y)$
⟨*proof*⟩

lemma *represents-scale*:
assumes *represents B r x*
shows *represents B* (*monomial c 0* $*$ *r*) (*c* $*$*s x*)
⟨*proof*⟩

lemma *represents-in-span*:
assumes *represents B r x*
shows $x \in$ *span B*
⟨*proof*⟩

lemma *syzygy-module-iff*: $s \in$ *syzygy-module B* $\longleftrightarrow$ *represents B s 0*
⟨*proof*⟩

lemma *syzygy-moduleI*:
assumes *represents B s 0*
shows $s \in$ *syzygy-module B*
⟨*proof*⟩

lemma *syzygy-moduleD*:
assumes $s \in$ *syzygy-module B*
shows *represents B s 0*
⟨*proof*⟩

lemma *zero-in-syzygy-module*: *0* $\in$ *syzygy-module B*
⟨*proof*⟩

lemma *syzygy-module-closed-plus*:
assumes *s1* $\in$ *syzygy-module B* **and** *s2* $\in$ *syzygy-module B*
shows *s1* + *s2* $\in$ *syzygy-module B*
⟨*proof*⟩

lemma *syzygy-module-closed-minus*:
assumes *s1* $\in$ *syzygy-module B* **and** *s2* $\in$ *syzygy-module B*

shows $s1 - s2 \in$ *syzygy-module B*
⟨*proof*⟩

lemma *syzygy-module-closed-times-monomial*:
assumes $s \in$ *syzygy-module B*
shows *monomial c 0* $*$ $s \in$ *syzygy-module B*
⟨*proof*⟩

end

context *term-powerprod*
begin

lemma *keys-rep-subset*:
assumes $u \in$ *keys* (*pmdl.rep r*)
obtains *t v* **where** $t \in$ *Keys* (*Poly-Mapping.range r*) **and** $v \in$ *Keys* (*keys r*) **and** $u = t \oplus v$
⟨*proof*⟩

lemma *rep-mult-scalar*: *pmdl.rep* (*punit.monom-mult c 0 r*) $= c \odot$ *pmdl.rep r*
⟨*proof*⟩

lemma *represents-mult-scalar*:
assumes *pmdl.represents B r x*
shows *pmdl.represents B* (*punit.monom-mult c 0 r*) ($c \odot x$)
⟨*proof*⟩

lemma *syzygy-module-closed-map-scale*: $s \in$ *pmdl.syzygy-module B* $\Longrightarrow c \cdot s \in$ *pmdl.syzygy-module B*
⟨*proof*⟩

lemma *phull-syzygy-module*: *phull* (*pmdl.syzygy-module B*) = *pmdl.syzygy-module B*
⟨*proof*⟩

end

18.2 Polynomial Mappings on List-Indices

definition *pm-of-idx-pm* :: ($'a$ *list*) $\Rightarrow$ (*nat* $\Rightarrow_0$ $'b$) $\Rightarrow$ $'a \Rightarrow_0$ $'b$::*zero*
where *pm-of-idx-pm xs f* = *Abs-poly-mapping* (λx. *lookup f* (*Min* $\{i.\ i <$ *length xs* $\wedge$ *xs* ! $i = x\}$) *when* $x \in$ *set xs*)

definition *idx-pm-of-pm* :: ($'a$ *list*) $\Rightarrow$ ($'a \Rightarrow_0$ $'b$) $\Rightarrow$ *nat* $\Rightarrow_0$ $'b$::*zero*
where *idx-pm-of-pm xs f* = *Abs-poly-mapping* (λi. *lookup f* (*xs* ! *i*) *when* $i <$ *length xs*)

lemma *lookup-pm-of-idx-pm*:
lookup (*pm-of-idx-pm xs f*) = (λx. *lookup f* (*Min* $\{i.\ i <$ *length xs* $\wedge$ *xs* ! $i = x\}$)

when $x \in$ *set xs*)
⟨*proof*⟩

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*⟩

lemma *keys-pm-of-idx-pm-subset*: *keys* (*pm-of-idx-pm xs f*) ⊆ *set xs*
⟨*proof*⟩

lemma *range-pm-of-idx-pm-subset*: *Poly-Mapping.range* (*pm-of-idx-pm xs f*) ⊆ *lookup f* ‘ {*0*..<*length xs*} − {*0*}
⟨*proof*⟩

corollary *range-pm-of-idx-pm-subset′*: *Poly-Mapping.range* (*pm-of-idx-pm xs f*) ⊆ *Poly-Mapping.range f*
⟨*proof*⟩

lemma *pm-of-idx-pm-zero* [*simp*]: *pm-of-idx-pm xs 0* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pm-of-idx-pm-uminus*: *pm-of-idx-pm xs* (− *f*) = − *pm-of-idx-pm xs f*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *lookup-idx-pm-of-pm*: *lookup* (*idx-pm-of-pm xs f*) = (λ*i*. *lookup f* (*xs* ! *i*)

when $i <$ *length xs*)
⟨*proof*⟩

lemma *keys-idx-pm-of-pm-subset*: *keys* (*idx-pm-of-pm xs f*) ⊆ {*0..<length xs*}
⟨*proof*⟩

lemma *idx-pm-of-pm-zero* [*simp*]: *idx-pm-of-pm xs 0* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *is-pot-ordD*:
assumes *is-pot-ord* **and** *component-of-term u* < *component-of-term v*
shows $u \prec_t v$
⟨*proof*⟩

lemma *is-pot-ordD2*:
assumes *is-pot-ord* **and** $u \preceq_t v$
shows *component-of-term u* ≤ *component-of-term v*

⟨*proof*⟩

lemma *is-pot-ord*:
assumes *is-pot-ord*
shows $u \preceq_t v \longleftrightarrow$ (*component-of-term* $u <$ *component-of-term* $v \lor$ (*component-of-term* $u =$ *component-of-term* $v \land$ *pp-of-term* $u \preceq$ *pp-of-term* v)) (**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

definition *map-component* :: $('k \Rightarrow 'k) \Rightarrow 't \Rightarrow 't$
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*))
⟨*proof*⟩

lemma *pp-of-map-component* [*term-simps*]: *pp-of-term* (*map-component f v*) = *pp-of-term v*
⟨*proof*⟩

lemma *component-of-map-component* [*term-simps*]:
component-of-term (*map-component f v*) = *f* (*component-of-term v*)
⟨*proof*⟩

lemma *map-component-term-of-pair* [*term-simps*]:
map-component f (*term-of-pair* (*t*, *k*)) = *term-of-pair* (*t*, *f k*)
⟨*proof*⟩

lemma *map-component-comp*: *map-component f* (*map-component g x*) = *map-component* ($\lambda k.$ *f* (*g k*)) *x*
⟨*proof*⟩

lemma *map-component-id* [*term-simps*]: *map-component* ($\lambda k.$ *k*) *x* = *x*
⟨*proof*⟩

lemma *map-component-inj*:
assumes *inj f* **and** *map-component f u* = *map-component f v*
shows *u* = *v*
⟨*proof*⟩

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*::$'t \Rightarrow ('a$::*graded-dickson-powerprod* $\times$ *nat*)
and *term-of-pair*::$('a \times nat) \Rightarrow 't$

and *ord*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\preceq$› *50*)
and *ord-strict* (**infixl** ‹$\prec$› *50*)
and *ord-term*::$'t \Rightarrow 't \Rightarrow$ *bool* (**infixl** ‹$\preceq_t$› *50*)
and *ord-term-strict*::$'t \Rightarrow 't \Rightarrow$ *bool* (**infixl** ‹$\prec_t$› *50*)
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* $\Rightarrow$ $('t \Rightarrow_0 'b) \Rightarrow$ *nat* $\Rightarrow$ $('t \Rightarrow_0 'b$::*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 $\leq$ *component-of-term x then lookup b* (*map-component* (λk. *k* $-$ *n*) *x*)
else 0)

definition *proj-poly-syz* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b$::*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* $\Rightarrow$ $('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b$::*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* $\Rightarrow ('t \Rightarrow_0 'b$::*semiring-1*) *list*
where *init-syzygy-list bs* = *map-idx* (*lift-poly-syz* (*length bs*)) *bs 0*

definition *proj-orig-basis* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b)$ *list* $\Rightarrow ('t \Rightarrow_0 'b$::*semiring-1*) *list*
where *proj-orig-basis n bs* = *map* (*proj-poly-syz n*) *bs*

definition *filter-syzygy-basis* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b)$ *list* $\Rightarrow ('t \Rightarrow_0 'b$::*semiring-1*) *list*
where *filter-syzygy-basis n bs* = [*b*$\leftarrow$*bs*. *component-of-term* ' *keys b* $\subseteq$ {*0*..<*n*}]

definition *syzygy-module-list* :: $('t \Rightarrow_0 'b)$ *list* $\Rightarrow ('t \Rightarrow_0 'b$::*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 $\leq$ *component-of-term x then lookup b* (*map-component* (λk. *k* $-$ *n*) *x*)
else 0) $\neq$ *0*} $\subseteq$ *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λk. *k* + *n*) ' *keys b*)

(**is** *?l* $\subseteq$ *?r*) **for** *b*::*'t* $\Rightarrow_0$ *'b*::*semiring-1*
⟨*proof*⟩

lemma *lookup-lift-poly-syz*:
lookup (*lift-poly-syz n b i*) =
(λx. *if pair-of-term x* = (*0*, *i*) *then 1 else if n* $\leq$ *component-of-term x then lookup b* (*map-component* (λk. *k* − *n*) *x*) *else 0*)
⟨*proof*⟩

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* $\leq$ *j then lookup b* (*term-of-pair* (*t*, *j* − *n*)) *else 0*)
⟨*proof*⟩

lemma *keys-lift-poly-syz*:
keys (*lift-poly-syz n b i*) = *insert* (*term-of-pair* (*0*, *i*)) (*map-component* (λk. *k* + *n*) ‘ *keys b*)
⟨*proof*⟩

18.4.2 *proj-poly-syz*

lemma *inj-map-component-plus*: *inj* (*map-component* (λk. *k* + *n*))
⟨*proof*⟩

lemma *lookup-proj-poly-syz*: *lookup* (*proj-poly-syz n p*) *x* = *lookup p* (*map-component* (λk. *k* + *n*) *x*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *keys-proj-poly-syz*: *keys* (*proj-poly-syz n p*) = *map-component* (λk. *k* + *n*) −‘ *keys p*
⟨*proof*⟩

lemma *proj-poly-syz-zero* [*simp*]: *proj-poly-syz n 0* = *0*
⟨*proof*⟩

lemma *proj-poly-syz-plus*: *proj-poly-syz n* (*p* + *q*) = *proj-poly-syz n p* + *proj-poly-syz n q*
⟨*proof*⟩

lemma *proj-poly-syz-sum*: *proj-poly-syz n* (*sum f A*) = ($\sum$ *a*$\in$*A*. *proj-poly-syz n* (*f a*))
⟨*proof*⟩

lemma *proj-poly-syz-sum-list*: $proj\text{-}poly\text{-}syz\ n\ (sum\text{-}list\ xs) = sum\text{-}list\ (map\ (proj\text{-}poly\text{-}syz\ n)\ xs)$
⟨*proof*⟩

lemma *proj-poly-syz-monom-mult*:
$proj\text{-}poly\text{-}syz\ n\ (monom\text{-}mult\ c\ t\ p) = monom\text{-}mult\ c\ t\ (proj\text{-}poly\text{-}syz\ n\ p)$
⟨*proof*⟩

lemma *proj-poly-syz-mult-scalar*:
$proj\text{-}poly\text{-}syz\ n\ (mult\text{-}scalar\ q\ p) = mult\text{-}scalar\ q\ (proj\text{-}poly\text{-}syz\ n\ p)$
⟨*proof*⟩

lemma *proj-poly-syz-lift-poly-syz*:
assumes $i < n$
shows $proj\text{-}poly\text{-}syz\ n\ (lift\text{-}poly\text{-}syz\ n\ p\ i) = p$
⟨*proof*⟩

lemma *proj-poly-syz-eq-zero-iff*: $proj\text{-}poly\text{-}syz\ n\ p = 0 \longleftrightarrow (component\text{-}of\text{-}term\ `\ keys\ p \subseteq \{0..<n\})$
⟨*proof*⟩

lemma *component-of-lt-ge*:
assumes *is-pot-ord* **and** $proj\text{-}poly\text{-}syz\ n\ p \neq 0$
shows $n \leq component\text{-}of\text{-}term\ (lt\ p)$
⟨*proof*⟩

lemma *lt-proj-poly-syz*:
assumes *is-pot-ord* **and** $proj\text{-}poly\text{-}syz\ n\ p \neq 0$
shows $lt\ (proj\text{-}poly\text{-}syz\ n\ p) = map\text{-}component\ (\lambda k.\ k - n)\ (lt\ p)$ (**is** - = *?l*)
⟨*proof*⟩

lemma *proj-proj-poly-syz*: $proj\text{-}poly\ k\ (proj\text{-}poly\text{-}syz\ n\ p) = proj\text{-}poly\ (k + n)\ p$
⟨*proof*⟩

lemma *poly-mapping-eqI-proj-syz*:
assumes $proj\text{-}poly\text{-}syz\ n\ p = proj\text{-}poly\text{-}syz\ n\ q$
and $\bigwedge k.\ k < n \Longrightarrow proj\text{-}poly\ k\ p = proj\text{-}poly\ k\ q$
shows $p = q$
⟨*proof*⟩

18.4.3 *cofactor-list-syz*

lemma *length-cofactor-list-syz* [*simp*]: $length\ (cofactor\text{-}list\text{-}syz\ n\ p) = n$
⟨*proof*⟩

lemma *cofactor-list-syz-nth*:
assumes $i < n$
shows $(cofactor\text{-}list\text{-}syz\ n\ p)\ !\ i = proj\text{-}poly\ i\ p$
⟨*proof*⟩

lemma *cofactor-list-syz-zero* [*simp*]: *cofactor-list-syz n 0* = *replicate n 0*
⟨*proof*⟩

lemma *cofactor-list-syz-plus*:
cofactor-list-syz n (*p* + *q*) = *map2* (+) (*cofactor-list-syz n p*) (*cofactor-list-syz n q*)
⟨*proof*⟩

18.4.4 *init-syzygy-list*

lemma *length-init-syzygy-list* [*simp*]: *length* (*init-syzygy-list bs*) = *length bs*
⟨*proof*⟩

lemma *init-syzygy-list-nth*:
assumes *i* < *length bs*
shows (*init-syzygy-list bs*) ! *i* = *lift-poly-syz* (*length bs*) (*bs* ! *i*) *i*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *pp-of-Keys-init-syzygy-list-superset*:
pp-of-term ‘ *Keys* (*set bs*) ⊆ *pp-of-term* ‘ *Keys* (*set* (*init-syzygy-list bs*))
⟨*proof*⟩

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*⟩

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*}
⟨*proof*⟩

lemma *proj-lift-poly-syz*:
assumes *j* < *n*
shows *proj-poly j* (*lift-poly-syz n p i*) = (*1 when j* = *i*)
⟨*proof*⟩

18.4.5 *proj-orig-basis*

lemma *length-proj-orig-basis* [*simp*]: *length* (*proj-orig-basis n bs*) = *length bs*
⟨*proof*⟩

lemma *proj-orig-basis-nth*:
assumes *i* < *length bs*
shows (*proj-orig-basis n bs*) ! *i* = *proj-poly-syz n* (*bs* ! *i*)
⟨*proof*⟩

lemma *proj-orig-basis-init-syzygy-list* [*simp*]:
proj-orig-basis (*length bs*) (*init-syzygy-list bs*) = *bs*
⟨*proof*⟩

lemma *set-proj-orig-basis*: *set* (*proj-orig-basis n bs*) = *proj-poly-syz n* ‘ *set bs*
⟨*proof*⟩

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*⟩

18.4.6 *filter-syzygy-basis*

lemma *filter-syzygy-basis-alt*: *filter-syzygy-basis n bs* = [*b*←*bs*. *proj-poly-syz n b* = *0*]
⟨*proof*⟩

lemma *set-filter-syzygy-basis*:
set (*filter-syzygy-basis n bs*) = {*b*∈*set bs*. *proj-poly-syz n b* = *0*}
⟨*proof*⟩

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*
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *monom-mult-atomize*:

monom-mult c t (*atomize-poly p*) = *atomize-poly* (*MPoly-Type-Class.punit.monom-mult* (*monomial c t*) *0 p*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *syzygy-module-list-closed-monom-mult*:
assumes *s* ∈ *syzygy-module-list bs*
shows *monom-mult c t s* ∈ *syzygy-module-list bs*
⟨*proof*⟩

lemma *pmdl-syzygy-module-list* [*simp*]: *pmdl* (*syzygy-module-list bs*) = *syzygy-module-list bs*
⟨*proof*⟩

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*⟩

lemma *component-of-syzygy-module-list*:
assumes *s* ∈ *syzygy-module-list bs*
shows *component-of-term* ' *keys s* ⊆ {*0*..<*length bs*}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map2-times-proj*:
map2 (∗) *xs* (*map* (*proj-poly k*) *ys*) = *map* (*proj-poly k* ∘ (λ(*x*, *y*). *x* ⊙ *y*)) (*zip xs ys*)
⟨*proof*⟩

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*⟩

18.4.8 Cofactors

lemma *map2-mult-scalar-plus*:
map2 (⊙) (*map2* (+) *xs ys*) *zs* = *map2* (+) (*map2* (⊙) *xs zs*) (*map2* (⊙) *ys zs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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** $f \neq 0$
shows ∃ *g*∈*set* (*filter-syzygy-basis* (*length bs*) *gs*). $g \neq 0 \wedge$ *lt g* $adds_t$ *lt f*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
)
⟨*proof*⟩

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*::($'a \times nat$) $\Rightarrow$ $'t$
and *cmp-term*
begin

sublocale *aux*: *gd-inf-term pair-of-term term-of-pair*
$\lambda s\ t.$ *le-of-nat-term-order cmp-term* (*term-of-pair* (s, *the-min*)) (*term-of-pair* (t, *the-min*))
$\lambda s\ t.$ *lt-of-nat-term-order cmp-term* (*term-of-pair* (s, *the-min*)) (*term-of-pair* (t, *the-min*))
le-of-nat-term-order cmp-term
lt-of-nat-term-order cmp-term ⟨*proof*⟩

definition *lift-keys* :: *nat* $\Rightarrow$ ($'t$, $'b$) *oalist-ntm* $\Rightarrow$ ($'t$, $'b$::*semiring-0*) *oalist-ntm*
where *lift-keys i xs* = *oalist-of-list-ntm* (*map-raw* ($\lambda kv.$ (*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* ($\lambda kv.$ (*map-component* ((+) *i*) (*fst kv*), *snd kv*)) (*list-of-oalist-ntm xs*))
⟨*proof*⟩

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*⟩

lemma *keys-lift-keys-subset*:
keys (*MP-oalist* (*lift-keys i xs*)) $\subseteq$ (*map-component* ((+) *i*)) ‘ *keys* (*MP-oalist xs*)
(**is** *?l* $\subseteq$ *?r*)
⟨*proof*⟩

end

global-interpretation *pprod*$'$: *gd-nat-inf-term* λx::($'a$, $'b$) *pp* $\times$ *nat*. x $\lambda x.$ 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* $\times$ *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*
⟨*proof*⟩

lemma *compute-adds-term-pprod* [*code*]:
adds-term-pprod u v = (*snd u* = *snd v* ∧ *adds-pp-add-linorder* (*fst u*) (*fst v*))
⟨*proof*⟩

lemma *compute-splus-pprod* [*code*]: *splus-pprod t* (*s*, *i*) = (*t* + *s*, *i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compute-trd-pprod* [*code*]: *trd-pprod to fs p* = *trd-aux-pprod to fs p* (*change-ord to 0*)
⟨*proof*⟩

lemmas [*code*] = *conversep-iff*

lemma *POT-is-pot-ord*: *pprod'.is-pot-ord* (*TYPE*('*a*::*nat*)) (*TYPE*('*b*::*nat*)) (*POT to*)
⟨*proof*⟩

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*⟩

19.2 Computations

experiment begin interpretation $trivariate_0$-*rat* ⟨*proof*⟩

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*)]
⟨*proof*⟩

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* $(Y \,\hat{}\, 3 + 2 * X * Z) - Vec_0\ 1\ (X \,\hat{}\, 4 + 3 * X^2 * Y)$,
$Vec_0\ 2\ 1 + Vec_0\ 3\ (Z^2 - X - Y)$,
$Vec_0\ 1\ (Z^2 - X - Y) - Vec_0\ 2\ (Y \,\hat{}\, 3 + 2 * X * Z)$,
$Vec_0\ 0\ (Z^2 - X - Y) - Vec_0\ 2\ (X \,\hat{}\, 4 + 3 * X^2 * Y)$,
$Vec_0\ 0\ (-\ (Y \,\hat{}\, 3 * Z^2) + Y \,\hat{}\, 4 + X * Y \,\hat{}\, 3 + 2 * X^2 * Z + 2 * X * Y * Z - 2 * X * Z \,\hat{}\, 3) +$
$Vec_0\ 1\ (X \,\hat{}\, 4 * Z^2 - X \,\hat{}\, 5 - X \,\hat{}\, 4 * Y - 3 * X \,\hat{}\, 3 * Y - 3 * X^2 * Y^2 + 3 * X^2 * Y * Z^2)$
]
⟨*proof*⟩

lemma
syzygy-basis DRLEX $[Vec_0\ 0\ (X \,\hat{}\, 4 + 3 * X^2 * Y),\ Vec_0\ 0\ (Y \,\hat{}\, 3 + 2 * X * Z),\ Vec_0\ 0\ (Z^2 - X - Y)] =$
[
$Vec_0\ 0\ (Y \,\hat{}\, 3 + 2 * X * Z) - Vec_0\ 1\ (X \,\hat{}\, 4 + 3 * X^2 * Y)$,
$Vec_0\ 1\ (Z^2 - X - Y) - Vec_0\ 2\ (Y \,\hat{}\, 3 + 2 * X * Z)$,
$Vec_0\ 0\ (Z^2 - X - Y) - Vec_0\ 2\ (X \,\hat{}\, 4 + 3 * X^2 * Y)$,
$Vec_0\ 0\ (-\ (Y \,\hat{}\, 3 * Z^2) + Y \,\hat{}\, 4 + X * Y \,\hat{}\, 3 + 2 * X^2 * Z + 2 * X * Y * Z - 2 * X * Z \,\hat{}\, 3) +$
$Vec_0\ 1\ (X \,\hat{}\, 4 * Z^2 - X \,\hat{}\, 5 - X \,\hat{}\, 4 * Y - 3 * X \,\hat{}\, 3 * Y - 3 * X^2 * Y^2 + 3 * X^2 * Y * Z^2)$
]
⟨*proof*⟩

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* ($\lambda p.$ (*p*, ())) (*init-syzygy-list-pprod*
$[Vec_0\ 0\ (X * Y - Z),\ Vec_0\ 0\ (X * Z - Y),\ Vec_0\ 0\ (Y * Z - X)]$)) ()) =
[
$Vec_0\ 0\ 1 + Vec_0\ 3\ (X * Y - Z)$,
$Vec_0\ 1\ 1 + Vec_0\ 3\ (X * Z - Y)$,
$Vec_0\ 2\ 1 + Vec_0\ 3\ (Y * Z - X)$,
$Vec_0\ 0\ (-\ X * Z + Y) + Vec_0\ 1\ (X * Y - Z)$,
$Vec_0\ 0\ (-\ Y * Z + X) + Vec_0\ 2\ (X * Y - Z)$,
$Vec_0\ 1\ (-\ Y * Z + X) + Vec_0\ 2\ (X * Z - Y)$,
$Vec_0\ 1\ (-Y) + Vec_0\ 2\ (X) + Vec_0\ 3\ (Y \,\hat{}\, 2 - X \,\hat{}\, 2)$,
$Vec_0\ 0\ (Z) + Vec_0\ 2\ (-X) + Vec_0\ 3\ (X \,\hat{}\, 2 - Z \,\hat{}\, 2)$,
$Vec_0\ 0\ (Y - Y * Z \,\hat{}\, 2) + Vec_0\ 1\ (Y \,\hat{}\, 2 * Z - Z) + Vec_0\ 2\ (Y \,\hat{}\, 2 - Z \,\hat{}\, 2)$,
$Vec_0\ 0\ (-\ Y) + Vec_0\ 1\ (-\ (X * Y)) + Vec_0\ 2\ (X \,\hat{}\, 2 - 1) + Vec_0\ 3\ (X - X \,\hat{}\, 3)$
]
⟨*proof*⟩

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* $(Y - Y * Z \mathbin{\char`\^} 2) + Vec_0$ *1* $(Y \mathbin{\char`\^} 2 * Z - Z) + Vec_0$ *2* $(Y \mathbin{\char`\^} 2 - Z \mathbin{\char`\^}$
$2)$
]
⟨*proof*⟩

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** $s \in .[X]$ **and** $t \in .[X]$
obtains *s adds t* | *t adds s*
⟨*proof*⟩

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*⟩

lemma *adds-iff-deg-le-univariate*:
assumes $s \in .[X]$ **and** $t \in .[X]$
shows *s adds t* $\longleftrightarrow$ *deg-pm s* $\leq$ *deg-pm t*
⟨*proof*⟩

corollary *ord-iff-deg-le-univariate*: $s \in .[X] \Longrightarrow t \in .[X] \Longrightarrow s \preceq t \longleftrightarrow$ *deg-pm s* $\leq$ *deg-pm t*
⟨*proof*⟩

lemma *poly-deg-univariate*:
assumes $p \in P[X]$
shows *poly-deg p* = *deg-pm* (*lpp p*)
⟨*proof*⟩

lemma *reduced-GB-univariate-cases*:
assumes $F \subseteq P[X]$
obtains *g* **where** $g \in P[X]$ **and** $g \neq 0$ **and** *lcf g* $= 1$ **and** *punit.reduced-GB F* $= \{g\}$ |
punit.reduced-GB F $= \{\}$
⟨*proof*⟩

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* $\leq$ *poly-deg f*
⟨*proof*⟩

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*⟩

lemma *lp-dehomogenize*:
assumes *is-hom-ord x* **and** *homogeneous p*
shows *lpp* (*dehomogenize x p*) = *except* (*lpp p*) $\{x\}$
⟨*proof*⟩

lemma *isGB-dehomogenize*:
assumes *is-hom-ord x* **and** *finite X* **and** $G \subseteq P[X]$ **and** *punit.is-Groebner-basis G*
and $\bigwedge g.\ g \in G \Longrightarrow$ *homogeneous g*
shows *punit.is-Groebner-basis* (*dehomogenize x* ‘ *G*)
⟨*proof*⟩

end

context *extended-ord-pm-powerprod*
begin

lemma *extended-ord-lp*:
assumes *None* $\notin$ *indets p*
shows *restrict-indets-pp* (*extended-ord.lpp p*) = *lpp* (*restrict-indets p*)
⟨*proof*⟩

lemma *restrict-indets-reduced-GB*:
assumes *finite X* **and** $F \subseteq P[X]$
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*) $\subseteq P[X]$
(**is** *?thesis3*)
⟨*proof*⟩

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. Kerber, 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.