Signature-Based Gröbner Basis Algorithms

Alexander Maletzky[*]

March 19, 2025

Abstract

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

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

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

Contents

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

1 Introduction

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

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

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

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

2 Preliminaries

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

2.1 Lists

2.1.1 Sequences of Lists

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

corollary *list-seq-length-mono-weak*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x\ \#\ seq\ i)$ **and** $i \le j$
shows $length\ (seq\ i) \le length\ (seq\ j)$
$\langle proof \rangle$

lemma *list-seq-indexE-length*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x\ \#\ seq\ i)$
obtains j **where** $i < length\ (seq\ j)$
$\langle proof \rangle$

lemma *list-seq-nth*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x\ \#\ seq\ i)$ **and** $i < length\ (seq\ j)$ **and** $j \le k$
shows $rev\ (seq\ k)\ !\ i = rev\ (seq\ j)\ !\ i$
$\langle proof \rangle$

corollary *list-seq-nth'*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x\ \#\ seq\ i)$ **and** $i < length\ (seq\ j)$ **and** $i < length\ (seq\ k)$
shows $rev\ (seq\ k)\ !\ i = rev\ (seq\ j)\ !\ i$
$\langle proof \rangle$

2.1.2 *filter*

lemma *filter-merge-wrt-1*:
assumes $\bigwedge y.\ y \in set\ ys \Longrightarrow P\ y \Longrightarrow False$
shows $filter\ P\ (merge\text{-}wrt\ rel\ xs\ ys) = filter\ P\ xs$
$\langle proof \rangle$

lemma *filter-merge-wrt-2*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x \Longrightarrow False$
shows $filter\ P\ (merge\text{-}wrt\ rel\ xs\ ys) = filter\ P\ ys$
$\langle proof \rangle$

lemma *length-filter-le-1*:
assumes $length\ (filter\ P\ xs) \le 1$ **and** $i < length\ xs$ **and** $j < length\ xs$
and $P\ (xs\ !\ i)$ **and** $P\ (xs\ !\ j)$
shows $i = j$
$\langle proof \rangle$

lemma *length-filter-eq* [*simp*]: $length\ (filter\ ((=)\ x)\ xs) = count\text{-}list\ xs\ x$
$\langle proof \rangle$

2.1.3 *drop*

lemma *nth-in-set-dropI*:

assumes $j \leq i$ **and** $i < length\ xs$
shows $xs\ !\ i \in set\ (drop\ j\ xs)$
⟨*proof*⟩

2.1.4 *count-list*

lemma *count-list-upt* [*simp*]: $count\text{-}list\ [a..<b]\ x = (if\ a \leq x \wedge x < b\ then\ 1\ else\ 0)$
⟨*proof*⟩

2.1.5 *sorted-wrt*

lemma *sorted-wrt-upt-iff*: $sorted\text{-}wrt\ rel\ [a..<b] \longleftrightarrow (\forall\ i\ j.\ a \leq i \longrightarrow i < j \longrightarrow j < b \longrightarrow rel\ i\ j)$
⟨*proof*⟩

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

lemma *map-insort-wrt*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow r2\ (f\ y)\ (f\ x) \longleftrightarrow r1\ y\ x$
shows $map\ f\ (insort\text{-}wrt\ r1\ y\ xs) = insort\text{-}wrt\ r2\ (f\ y)\ (map\ f\ xs)$
⟨*proof*⟩

lemma *map-merge-wrt*:
assumes $f\ `\ set\ xs \cap f\ `\ set\ ys = \{\}$
and $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ ys \Longrightarrow r2\ (f\ x)\ (f\ y) \longleftrightarrow r1\ x\ y$
shows $map\ f\ (merge\text{-}wrt\ r1\ xs\ ys) = merge\text{-}wrt\ r2\ (map\ f\ xs)\ (map\ f\ ys)$
⟨*proof*⟩

2.2 Recursive Functions

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

context
fixes $Q :: 'a \Rightarrow bool$
fixes $g :: 'a \Rightarrow 'b$
fixes $h :: 'a \Rightarrow 'a$
begin

function (*domintros*) $recfun\text{-}aux :: 'a \Rightarrow 'b$ **where**
$recfun\text{-}aux\ x = (if\ Q\ x\ then\ g\ x\ else\ h'\ (recfun\text{-}aux\ (h\ x)))$
⟨*proof*⟩

lemmas [*induct del*] = *recfun-aux.pinduct*

definition $dom :: 'a \Rightarrow bool$

where $dom\ x \longleftrightarrow (\exists\, k.\ Q\ ((h\ \widehat{\ }\widehat{\ }\ k)\ x))$

lemma *domI*:
assumes $\neg\ Q\ x \Longrightarrow dom\ (h\ x)$
shows $dom\ x$
⟨*proof*⟩

lemma *domD*:
assumes $dom\ x$ **and** $\neg\ Q\ x$
shows $dom\ (h\ x)$
⟨*proof*⟩

lemma *recfun-aux-domI*:
assumes $dom\ x$
shows *recfun-aux-dom x*
⟨*proof*⟩

lemma *recfun-aux-domD*:
assumes *recfun-aux-dom x*
shows $dom\ x$
⟨*proof*⟩

corollary *recfun-aux-dom-alt*: *recfun-aux-dom* = *dom*
⟨*proof*⟩

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

lemma *simps*: $fun\ x = (if\ Q\ x\ then\ g\ x\ else\ h'\ (fun\ (h\ x)))$
⟨*proof*⟩

lemma *eq-fixpointI*: $\neg\ dom\ x \Longrightarrow fun\ x = b$
⟨*proof*⟩

lemma *pinduct*: $dom\ x \Longrightarrow (\bigwedge x.\ dom\ x \Longrightarrow (\neg\ Q\ x \Longrightarrow P\ (h\ x)) \Longrightarrow P\ x) \Longrightarrow P\ x$
⟨*proof*⟩

end

end

interpretation *tailrec*: *recursive* $\lambda x.\ x$ *undefined*
⟨*proof*⟩

2.3 Binary Relations

lemma *almost-full-on-Int*:
assumes *almost-full-on P1 A1* **and** *almost-full-on P2 A2*

shows *almost-full-on* ($\lambda x\ y.\ P1\ x\ y \wedge P2\ x\ y$) ($A1 \cap A2$) (**is** *almost-full-on ?P ?A*)
⟨*proof*⟩

corollary *almost-full-on-same*:
assumes *almost-full-on P1 A* **and** *almost-full-on P2 A*
shows *almost-full-on* ($\lambda x\ y.\ P1\ x\ y \wedge P2\ x\ y$) *A*
⟨*proof*⟩

context *ord*
begin

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

lemma *is-le-relI* [*simp*]: *is-le-rel* (=) *is-le-rel* ($\leq$) *is-le-rel* (<)
⟨*proof*⟩

lemma *is-le-relE*:
assumes *is-le-rel rel*
obtains *rel* = (=) | *rel* = ($\leq$) | *rel* = (<)
⟨*proof*⟩

end

context *preorder*
begin

lemma *is-le-rel-le*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *is-le-rel-trans*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow rel\ y\ z \Longrightarrow rel\ x\ z$
⟨*proof*⟩

lemma *is-le-rel-trans-le-left*:
assumes *is-le-rel rel*
shows $x \leq y \Longrightarrow rel\ y\ z \Longrightarrow x \leq z$
⟨*proof*⟩

lemma *is-le-rel-trans-le-right*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow y \leq z \Longrightarrow x \leq z$
⟨*proof*⟩

lemma *is-le-rel-trans-less-left*:

assumes *is-le-rel rel*
shows $x < y \Longrightarrow$ *rel* $y\ z \Longrightarrow x < z$
⟨*proof*⟩

lemma *is-le-rel-trans-less-right*:
assumes *is-le-rel rel*
shows *rel* $x\ y \Longrightarrow y < z \Longrightarrow x < z$
⟨*proof*⟩

end

context *order*
begin

lemma *is-le-rel-distinct*:
assumes *is-le-rel rel*
shows *rel* $x\ y \Longrightarrow x \neq y \Longrightarrow x < y$
⟨*proof*⟩

lemma *is-le-rel-antisym*:
assumes *is-le-rel rel*
shows *rel* $x\ y \Longrightarrow$ *rel* $y\ x \Longrightarrow x = y$
⟨*proof*⟩

end

end

3 More Properties of Power-Products and Multivariate Polynomials

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

3.1 Power-Products

lemma (**in** *comm-powerprod*) *minus-plus'*: s *adds* $t \Longrightarrow u + (t - s) = (u + t) - s$
⟨*proof*⟩

context *ulcs-powerprod*
begin

lemma *lcs-alt-2*:
assumes $a + x = b + y$
shows *lcs* $x\ y = (b + y) -$ *gcs* $a\ b$
⟨*proof*⟩

corollary *lcs-alt-1*:
assumes $a + x = b + y$
shows $lcs\ x\ y = (a + x) - gcs\ a\ b$
⟨*proof*⟩

corollary *lcs-minus-1*:
assumes $a + x = b + y$
shows $lcs\ x\ y - x = a - gcs\ a\ b$
⟨*proof*⟩

corollary *lcs-minus-2*:
assumes $a + x = b + y$
shows $lcs\ x\ y - y = b - gcs\ a\ b$
⟨*proof*⟩

lemma *gcs-minus*:
assumes *u adds s* **and** *u adds t*
shows $gcs\ (s - u)\ (t - u) = gcs\ s\ t - u$
⟨*proof*⟩

corollary *gcs-minus-gcs*: $gcs\ (s - (gcs\ s\ t))\ (t - (gcs\ s\ t)) = 0$
⟨*proof*⟩

end

3.2 Miscellaneous

lemma *poly-mapping-rangeE*:
assumes $c \in$ *Poly-Mapping.range p*
obtains *k* **where** $k \in$ *keys p* **and** $c =$ *lookup p k*
⟨*proof*⟩

lemma *poly-mapping-range-nonzero*: $0 \notin$ *Poly-Mapping.range p*
⟨*proof*⟩

lemma (**in** *term-powerprod*) *Keys-range-vectorize-poly*: *Keys* (*Poly-Mapping.range* (*vectorize-poly p*)) = *pp-of-term* ‘ *keys p*
⟨*proof*⟩

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

context *ordered-term*
begin

lemma *lt-lookup-vectorize*: *punit.lt* (*lookup* (*vectorize-poly p*) (*component-of-term* (*lt p*))) = *lp p*
⟨*proof*⟩

lemma *lower-higher-zeroI*: $u \preceq_t v \Longrightarrow$ *lower* (*higher p v*) $u = 0$

⟨*proof*⟩

lemma *lookup-minus-higher*: *lookup* $(p - \textit{higher } p\ v)\ u = (\textit{lookup } p\ u \textit{ when } u \preceq_t v)$
⟨*proof*⟩

lemma *keys-minus-higher*: *keys* $(p - \textit{higher } p\ v) = \{u \in \textit{keys } p.\ u \preceq_t v\}$
⟨*proof*⟩

lemma *lt-minus-higher*: $v \in \textit{keys } p \Longrightarrow \textit{lt } (p - \textit{higher } p\ v) = v$
⟨*proof*⟩

lemma *lc-minus-higher*: $v \in \textit{keys } p \Longrightarrow \textit{lc } (p - \textit{higher } p\ v) = \textit{lookup } p\ v$
⟨*proof*⟩

lemma *tail-minus-higher*: $v \in \textit{keys } p \Longrightarrow \textit{tail } (p - \textit{higher } p\ v) = \textit{lower } p\ v$
⟨*proof*⟩

end

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

lemma (**in** *gd-term*) *dgrad-p-set-closed-mult-scalar*:
assumes *dickson-grading d* **and** $p \in$ *punit.dgrad-p-set d m* **and** $r \in$ *dgrad-p-set d m*
shows $p \odot r \in$ *dgrad-p-set d m*
⟨*proof*⟩

3.5 Regular Sequences

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

lemma *is-regular-sequenceD*:
is-regular-sequence fs $\Longrightarrow j <$ *length fs* $\Longrightarrow q * fs\ !\ j \in$ *ideal* (*set* (*take j fs*)) $\Longrightarrow$
$q \in$ *ideal* (*set* (*take j fs*))
⟨*proof*⟩

lemma *is-regular-sequence-Nil*: *is-regular-sequence* []
⟨*proof*⟩

lemma *is-regular-sequence-snocI*:
assumes $\bigwedge q.\ q * f \in$ *ideal* (*set fs*) $\Longrightarrow q \in$ *ideal* (*set fs*) **and** *is-regular-sequence fs*
shows *is-regular-sequence* (*fs* @ [*f*])
⟨*proof*⟩

lemma *is-regular-sequence-snocD*:
  assumes *is-regular-sequence* (*fs* @ [*f*])
  shows $\bigwedge q.\ q * f \in ideal\ (set\ fs) \Longrightarrow q \in ideal\ (set\ fs)$
    and *is-regular-sequence fs*
⟨*proof*⟩

lemma *is-regular-sequence-removeAll-zero*:
  assumes *is-regular-sequence fs*
  shows *is-regular-sequence* (*removeAll 0 fs*)
  ⟨*proof*⟩

lemma *is-regular-sequence-remdups*:
  assumes *is-regular-sequence fs*
  shows *is-regular-sequence* (*rev* (*remdups* (*rev fs*)))
  ⟨*proof*⟩

end

4 Signature-Based Algorithms for Computing Gröbner Bases

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

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

4.1 More Preliminaries

lemma (**in** *gd-term*) *lt-spoly-less-lcs*:
  assumes $p \neq 0$ **and** $q \neq 0$ **and** *spoly p q* $\neq 0$
  shows *lt* (*spoly p q*) $\prec_t$ *term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*))
⟨*proof*⟩

4.2 Module Polynomials

locale *qpm-inf-term* =
    gd-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
      for *pair-of-term*::$'t \Rightarrow ('a$::*quasi-pm-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

lemma *in-idealE-rep-dgrad-p-set*:
assumes *hom-grading d* **and** $B \subseteq$ *punit.dgrad-p-set d m* **and** $p \in$ *punit.dgrad-p-set d m* **and** $p \in$ *ideal B*
obtains *r* **where** *keys r* $\subseteq$ *B* **and** *Poly-Mapping.range r* $\subseteq$ *punit.dgrad-p-set d m* **and** $p =$ *ideal.rep r*
⟨*proof*⟩

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

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

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

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

lemma *sig-inv-setI*: *keys* (*vectorize-poly r*) $\subseteq$ {*0..<j*} $\Longrightarrow$ *r* $\in$ *sig-inv-set' j*
⟨*proof*⟩

lemma *sig-inv-setD*: *r* $\in$ *sig-inv-set' j* $\Longrightarrow$ *keys* (*vectorize-poly r*) $\subseteq$ {*0..<j*}
⟨*proof*⟩

lemma *sig-inv-setI'*:
assumes $\bigwedge v.\ v \in$ *keys r* $\Longrightarrow$ *component-of-term v* $< j$
shows *r* $\in$ *sig-inv-set' j*
⟨*proof*⟩

lemma *sig-inv-setD'*:
assumes *r* $\in$ *sig-inv-set' j* **and** *v* $\in$ *keys r*
shows *component-of-term v* $< j$
⟨*proof*⟩

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

lemma *sig-inv-set-mono*:
assumes $i \leq j$
shows *sig-inv-set' i* $\subseteq$ *sig-inv-set' j*
⟨*proof*⟩

lemma *sig-inv-set-zero*: *0* $\in$ *sig-inv-set' j*
⟨*proof*⟩

lemma *sig-inv-set-closed-uminus*: *r* $\in$ *sig-inv-set' j* $\Longrightarrow$ $-$ *r* $\in$ *sig-inv-set' j*

⟨*proof*⟩

lemma *sig-inv-set-closed-plus*:
assumes $r \in$ *sig-inv-set'* j **and** $s \in$ *sig-inv-set'* j
shows $r + s \in$ *sig-inv-set'* j
⟨*proof*⟩

lemma *sig-inv-set-closed-minus*:
assumes $r \in$ *sig-inv-set'* j **and** $s \in$ *sig-inv-set'* j
shows $r - s \in$ *sig-inv-set'* j
⟨*proof*⟩

lemma *sig-inv-set-closed-monom-mult*:
assumes $r \in$ *sig-inv-set'* j
shows *monom-mult* c t $r \in$ *sig-inv-set'* j
⟨*proof*⟩

lemma *sig-inv-set-closed-mult-scalar*:
assumes $r \in$ *sig-inv-set'* j
shows $p \odot r \in$ *sig-inv-set'* j
⟨*proof*⟩

lemma *rep-list-zero*: *rep-list* $0 = 0$
⟨*proof*⟩

lemma *rep-list-uminus*: *rep-list* $(- r) = -$ *rep-list* r
⟨*proof*⟩

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

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

lemma *vectorize-mult-scalar*:
vectorize-poly $(p \odot q) =$ *MPoly-Type-Class.punit.monom-mult* p 0 (*vectorize-poly* q)
⟨*proof*⟩

lemma *rep-list-mult-scalar*: *rep-list* $(c \odot r) = c *$ *rep-list* r
⟨*proof*⟩

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

lemma *rep-list-monomial*:
assumes *distinct fs*
shows *rep-list* (*monomial* c u) =

(*punit.monom-mult c* (*pp-of-term u*) (*fs* ! (*component-of-term u*))
when component-of-term u < *length fs*)
⟨*proof*⟩

lemma *rep-list-in-ideal-sig-inv-set*:
assumes *r* ∈ *sig-inv-set′ j*
shows *rep-list r* ∈ *ideal* (*set* (*take j fs*))
⟨*proof*⟩

corollary *rep-list-subset-ideal-sig-inv-set*:
B ⊆ *sig-inv-set′ j* ⟹ *rep-list* ‘ *B* ⊆ *ideal* (*set* (*take j fs*))
⟨*proof*⟩

lemma *rep-list-in-ideal*: *rep-list r* ∈ *ideal* (*set fs*)
⟨*proof*⟩

corollary *rep-list-subset-ideal*: *rep-list* ‘ *B* ⊆ *ideal* (*set fs*)
⟨*proof*⟩

lemma *in-idealE-rep-list*:
assumes *p* ∈ *ideal* (*set fs*)
obtains *r* **where** *p* = *rep-list r* **and** *r* ∈ *sig-inv-set*
⟨*proof*⟩

lemma *keys-rep-list-subset*:
assumes *t* ∈ *keys* (*rep-list r*)
obtains *v s* **where** *v* ∈ *keys r* **and** *s* ∈ *Keys* (*set fs*) **and** *t* = *pp-of-term v* + *s*
⟨*proof*⟩

lemma *dgrad-p-set-le-rep-list*:
assumes *dickson-grading d* **and** *dgrad-set-le d* (*pp-of-term* ‘ *keys r*) (*Keys* (*set fs*))
shows *punit.dgrad-p-set-le d* {*rep-list r*} (*set fs*)
⟨*proof*⟩

corollary *dgrad-p-set-le-rep-list-image*:
assumes *dickson-grading d* **and** *dgrad-set-le d* (*pp-of-term* ‘ *Keys F*) (*Keys* (*set fs*))
shows *punit.dgrad-p-set-le d* (*rep-list* ‘ *F*) (*set fs*)
⟨*proof*⟩
term *Max*

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

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

lemma *dgrad-max-0*: *d 0* ≤ *dgrad-max d*

⟨*proof*⟩

lemma *dgrad-max-1*: *set fs* $\subseteq$ *punit-dgrad-max-set d*
⟨*proof*⟩

lemma *dgrad-max-2*:
assumes *dickson-grading d* **and** *r* $\in$ *dgrad-max-set d*
shows *rep-list r* $\in$ *punit-dgrad-max-set d*
⟨*proof*⟩

corollary *dgrad-max-3*:
assumes *dickson-grading d* **and** *F* $\subseteq$ *dgrad-max-set d*
shows *rep-list* ‘ *F* $\subseteq$ *punit-dgrad-max-set d*
⟨*proof*⟩

lemma *punit-dgrad-max-set-subset-dgrad-p-set*:
assumes *dickson-grading d* **and** *set fs* $\subseteq$ *punit.dgrad-p-set d m* **and** $\neg$ *set fs* $\subseteq$ $\{0\}$
shows *punit-dgrad-max-set d* $\subseteq$ *punit.dgrad-p-set d m*
⟨*proof*⟩

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

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

lemma *dgrad-sig-set-set-mono*: *i* $\leq$ *j* $\Longrightarrow$ *dgrad-sig-set' i d* $\subseteq$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-uminus*: *r* $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ $-$ *r* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-plus*:
r $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *s* $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *r* + *s* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-minus*:
r $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *s* $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *r* $-$ *s* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-monom-mult*:
assumes *dickson-grading d* **and** *d t* $\leq$ *dgrad-max d*
shows *p* $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *monom-mult c t p* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-monom-mult-zero*:
p $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *monom-mult c 0 p* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-mult-scalar*:
dickson-grading d $\Longrightarrow$ *p* $\in$ *punit-dgrad-max-set d* $\Longrightarrow$ *r* $\in$ *dgrad-sig-set' j d* $\Longrightarrow$ *p* $\odot$ *r* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-monomial*:
assumes *d* (*pp-of-term u*) $\leq$ *dgrad-max d* **and** *component-of-term u* $<$ *j*
shows *monomial c u* $\in$ *dgrad-sig-set' j d*
⟨*proof*⟩

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

lemma *in-idealE-rep-list-dgrad-sig-set-take*:
assumes *hom-grading d* **and** *p* $\in$ *punit-dgrad-max-set d* **and** *p* $\in$ *ideal* (*set* (*take j fs*))
obtains *r* **where** *r* $\in$ *dgrad-sig-set d* **and** *r* $\in$ *dgrad-sig-set' j d* **and** *p* = *rep-list r*
⟨*proof*⟩

corollary *in-idealE-rep-list-dgrad-sig-set*:
assumes *hom-grading d* **and** *p* $\in$ *punit-dgrad-max-set d* **and** *p* $\in$ *ideal* (*set fs*)
obtains *r* **where** *r* $\in$ *dgrad-sig-set d* **and** *p* = *rep-list r*
⟨*proof*⟩

lemma *dgrad-sig-setD-lp*:
assumes *p* $\in$ *dgrad-sig-set' j d*
shows *d* (*lp p*) $\leq$ *dgrad-max d*
⟨*proof*⟩

lemma *dgrad-sig-setD-lt*:
assumes *p* $\in$ *dgrad-sig-set' j d* **and** *p* $\neq$ *0*
shows *component-of-term* (*lt p*) $<$ *j*
⟨*proof*⟩

lemma *dgrad-sig-setD-rep-list-lt*:
assumes *dickson-grading d* **and** *p* $\in$ *dgrad-sig-set' j d*
shows *d* (*punit.lt* (*rep-list p*)) $\leq$ *dgrad-max d*
⟨*proof*⟩

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

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

lemma *fst-spp-of*: *fst* (*spp-of r*) = *lt r*
⟨*proof*⟩

lemma *snd-spp-of*: $snd\ (spp\text{-}of\ r) = rep\text{-}list\ r$
⟨*proof*⟩

4.2.1 Signature Reduction

lemma *term-is-le-rel-canc-left*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ rel$
shows $rel\ (t \oplus u)\ (t \oplus v) \longleftrightarrow rel\ u\ v$
⟨*proof*⟩

lemma *term-is-le-rel-minus*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ rel$ **and** $s\ adds\ t$
shows $rel\ ((t - s) \oplus u)\ v \longleftrightarrow rel\ (t \oplus u)\ (s \oplus v)$
⟨*proof*⟩

lemma *term-is-le-rel-minus-minus*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ rel$ **and** $a\ adds\ t$ **and** $b\ adds\ t$
shows $rel\ ((t - a) \oplus u)\ ((t - b) \oplus v) \longleftrightarrow rel\ (b \oplus u)\ (a \oplus v)$
⟨*proof*⟩

lemma *pp-is-le-rel-canc-right*:
assumes $ordered\text{-}powerprod\text{-}lin.is\text{-}le\text{-}rel\ rel$
shows $rel\ (s + u)\ (t + u) \longleftrightarrow rel\ s\ t$
⟨*proof*⟩

lemma *pp-is-le-rel-canc-left*: $ordered\text{-}powerprod\text{-}lin.is\text{-}le\text{-}rel\ rel \Longrightarrow rel\ (t + u)\ (t + v) \longleftrightarrow rel\ u\ v$
⟨*proof*⟩

definition *sig-red-single* :: $('t \Rightarrow 't \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow bool$
where $sig\text{-}red\text{-}single\ sing\text{-}reg\ top\text{-}tail\ p\ q\ f\ t \longleftrightarrow$
$(rep\text{-}list\ f \neq 0 \wedge lookup\ (rep\text{-}list\ p)\ (t + punit.lt\ (rep\text{-}list\ f)) \neq 0 \wedge$
$q = p - monom\text{-}mult\ ((lookup\ (rep\text{-}list\ p)\ (t + punit.lt\ (rep\text{-}list\ f)))\ /\ punit.lc\ (rep\text{-}list\ f))\ t\ f \wedge$
$ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ sing\text{-}reg \wedge ordered\text{-}powerprod\text{-}lin.is\text{-}le\text{-}rel\ top\text{-}tail \wedge$
$sing\text{-}reg\ (t \oplus lt\ f)\ (lt\ p) \wedge top\text{-}tail\ (t + punit.lt\ (rep\text{-}list\ f))\ (punit.lt\ (rep\text{-}list\ p)))$

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

- If *sing-reg* is $(=)$, the reduction is singular.
- If *sing-reg* is $(\prec_t)$, the reduction is regular.
- If *sing-reg* is $(\preceq_t)$, the reduction is an arbitrary signature-reduction.

- If *top-tail* is $(=)$, it is a top reduction.
- If *top-tail* is $(\prec)$, it is a tail reduction.
- If *top-tail* is $(\preceq)$, the reduction is an arbitrary signature-reduction.

definition *sig-red* :: $('t \Rightarrow 't \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
where *sig-red sing-reg top-tail F p q* $\longleftrightarrow$ $(\exists f \in F.\ \exists t.$ *sig-red-single sing-reg top-tail p q f t*$)$

definition *is-sig-red* :: $('t \Rightarrow 't \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
where *is-sig-red sing-reg top-tail F p* $\longleftrightarrow$ $(\exists q.$ *sig-red sing-reg top-tail F p q*$)$

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

lemma *sig-red-singleD1*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *rep-list f* $\neq 0$
⟨*proof*⟩

lemma *sig-red-singleD2*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows t + *punit.lt* (*rep-list f*) $\in$ *keys* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-singleD3*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows $q = p -$ *monom-mult* ((*lookup* (*rep-list p*) (t + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
⟨*proof*⟩

lemma *sig-red-singleD4*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *ord-term-lin.is-le-rel sing-reg*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *sig-red-singleD6*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *sing-reg* ($t \oplus lt\ f$) ($lt\ p$)
⟨*proof*⟩

lemma *sig-red-singleD7*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *top-tail* (t + *punit.lt* (*rep-list* f)) (*punit.lt* (*rep-list* p))
⟨*proof*⟩

lemma *sig-red-singleD8*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows $t \oplus lt\ f \preceq_t lt\ p$
⟨*proof*⟩

lemma *sig-red-singleD9*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows t + *punit.lt* (*rep-list* f) $\preceq$ *punit.lt* (*rep-list* p)
⟨*proof*⟩

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

lemma *sig-red-single-red-single*:
sig-red-single sing-reg top-tail p q f t $\Longrightarrow$ *punit.red-single* (*rep-list* p) (*rep-list* q) (*rep-list* f) t
⟨*proof*⟩

lemma *sig-red-single-regular-lt*:
assumes *sig-red-single* ($\prec_t$) *top-tail p q f t*
shows $lt\ q = lt\ p$
⟨*proof*⟩

lemma *sig-red-single-regular-lc*:
assumes *sig-red-single* ($\prec_t$) *top-tail p q f t*
shows $lc\ q = lc\ p$
⟨*proof*⟩

lemma *sig-red-single-lt*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows $lt\ q \preceq_t lt\ p$
⟨*proof*⟩

lemma *sig-red-single-lt-rep-list*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *punit.lt* (*rep-list* q) $\preceq$ *punit.lt* (*rep-list* p)

⟨*proof*⟩

lemma *sig-red-single-tail-lt-in-keys-rep-list*:
  assumes *sig-red-single sing-reg* $(\prec)$ *p q f t*
  shows *punit.lt* (*rep-list p*) $\in$ *keys* (*rep-list q*)
⟨*proof*⟩

corollary *sig-red-single-tail-lt-rep-list*:
  assumes *sig-red-single sing-reg* $(\prec)$ *p q f t*
  shows *punit.lt* (*rep-list q*) = *punit.lt* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-single-tail-lc-rep-list*:
  assumes *sig-red-single sing-reg* $(\prec)$ *p q f t*
  shows *punit.lc* (*rep-list q*) = *punit.lc* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-single-top-lt-rep-list*:
  assumes *sig-red-single sing-reg* $(=)$ *p q f t* **and** *rep-list q* $\neq$ *0*
  shows *punit.lt* (*rep-list q*) $\prec$ *punit.lt* (*rep-list p*)
⟨*proof*⟩

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

lemma *sig-red-single-sing-reg-cases*:
  sig-red-single $(\preceq_t)$ *top-tail p q f t* = (*sig-red-single* $(=)$ *top-tail p q f t* $\vee$ *sig-red-single* $(\prec_t)$ *top-tail p q f t*)
  ⟨*proof*⟩

corollary *sig-red-single-sing-regI*:
  assumes *sig-red-single sing-reg top-tail p q f t*
  shows *sig-red-single* $(\preceq_t)$ *top-tail p q f t*
⟨*proof*⟩

lemma *sig-red-single-top-tail-cases*:
  sig-red-single sing-reg $(\preceq)$ *p q f t* = (*sig-red-single sing-reg* $(=)$ *p q f t* $\vee$ *sig-red-single sing-reg* $(\prec)$ *p q f t*)
  ⟨*proof*⟩

corollary *sig-red-single-top-tailI*:
  assumes *sig-red-single sing-reg top-tail p q f t*
  shows *sig-red-single sing-reg* $(\preceq)$ *p q f t*
⟨*proof*⟩

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

assumes *dickson-grading d* **and** $p \in$ *dgrad-max-set d* **and** $f \in$ *dgrad-max-set d*
and *sig-red-single sing-red top-tail p q f t*
shows $q \in$ *dgrad-max-set d*
⟨*proof*⟩

lemma *sig-inv-set-closed-sig-red-single*:
assumes $p \in$ *sig-inv-set* **and** $f \in$ *sig-inv-set* **and** *sig-red-single sing-red top-tail p q f t*
shows $q \in$ *sig-inv-set*
⟨*proof*⟩

corollary *dgrad-sig-set-closed-sig-red-single*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-sig-set d* **and** $f \in$ *dgrad-sig-set d*
and *sig-red-single sing-red top-tail p q f t*
shows $q \in$ *dgrad-sig-set d*
⟨*proof*⟩

lemma *sig-red-regular-lt*: *sig-red* $(\prec_t)$ *top-tail F p q* $\Longrightarrow$ *lt q* $=$ *lt p*
⟨*proof*⟩

lemma *sig-red-regular-lc*: *sig-red* $(\prec_t)$ *top-tail F p q* $\Longrightarrow$ *lc q* $=$ *lc p*
⟨*proof*⟩

lemma *sig-red-lt*: *sig-red sing-reg top-tail F p q* $\Longrightarrow$ *lt q* $\preceq_t$ *lt p*
⟨*proof*⟩

lemma *sig-red-tail-lt-rep-list*: *sig-red sing-reg* $(\prec)$ *F p q* $\Longrightarrow$ *punit.lt* (*rep-list q*) $=$ *punit.lt* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-tail-lc-rep-list*: *sig-red sing-reg* $(\prec)$ *F p q* $\Longrightarrow$ *punit.lc* (*rep-list q*) $=$ *punit.lc* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-top-lt-rep-list*:
sig-red sing-reg $(=)$ *F p q* $\Longrightarrow$ *rep-list q* $\neq$ *0* $\Longrightarrow$ *punit.lt* (*rep-list q*) $\prec$ *punit.lt* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-lt-rep-list*: *sig-red sing-reg top-tail F p q* $\Longrightarrow$ *punit.lt* (*rep-list q*) $\preceq$ *punit.lt* (*rep-list p*)
⟨*proof*⟩

lemma *sig-red-red*: *sig-red sing-reg top-tail F p q* $\Longrightarrow$ *punit.red* (*rep-list* ‘ *F*) (*rep-list p*) (*rep-list q*)
⟨*proof*⟩

lemma *sig-red-monom-mult*:
sig-red sing-reg top-tail F p q $\Longrightarrow$ $c \neq 0$ $\Longrightarrow$ *sig-red sing-reg top-tail F* (*monom-mult*

c s p) (*monom-mult c s q*)
⟨*proof*⟩

lemma *sig-red-sing-reg-cases*:
sig-red ($\preceq_t$) *top-tail F p q* = (*sig-red* (=) *top-tail F p q* ∨ *sig-red* ($\prec_t$) *top-tail F p q*)
⟨*proof*⟩

corollary *sig-red-sing-regI*: *sig-red sing-reg top-tail F p q* ⟹ *sig-red* ($\preceq_t$) *top-tail F p q*
⟨*proof*⟩

lemma *sig-red-top-tail-cases*:
sig-red sing-reg (⪯) *F p q* = (*sig-red sing-reg* (=) *F p q* ∨ *sig-red sing-reg* (≺) *F p q*)
⟨*proof*⟩

corollary *sig-red-top-tailI*: *sig-red sing-reg top-tail F p q* ⟹ *sig-red sing-reg* (⪯) *F p q*
⟨*proof*⟩

lemma *sig-red-wf-dgrad-max-set*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-max-set d*
shows *wfP* (*sig-red sing-reg top-tail F*)$^{-1-1}$
⟨*proof*⟩

lemma *dgrad-sig-set-closed-sig-red*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-sig-set d* **and** *p* ∈ *dgrad-sig-set d*
and *sig-red sing-red top-tail F p q*
shows *q* ∈ *dgrad-sig-set d*
⟨*proof*⟩

lemma *sig-red-mono*: *sig-red sing-reg top-tail F p q* ⟹ *F* ⊆ *F'* ⟹ *sig-red sing-reg top-tail F' p q*
⟨*proof*⟩

lemma *sig-red-Un*:
sig-red sing-reg top-tail (*A* ∪ *B*) *p q* ⟷ (*sig-red sing-reg top-tail A p q* ∨ *sig-red sing-reg top-tail B p q*)
⟨*proof*⟩

lemma *sig-red-subset*:
assumes *sig-red sing-reg top-tail F p q* **and** *sing-reg* = ($\preceq_t$) ∨ *sing-reg* = ($\prec_t$)
shows *sig-red sing-reg top-tail* {*f*∈*F*. *sing-reg* (*lt f*) (*lt p*)} *p q*
⟨*proof*⟩

lemma *sig-red-regular-rtrancl-lt*:
assumes (*sig-red* ($\prec_t$) *top-tail F*)** *p q*
shows *lt q* = *lt p*

⟨*proof*⟩

lemma *sig-red-regular-rtrancl-lc*:
assumes $(\textit{sig-red}\ (\prec_t)\ \textit{top-tail}\ F)^{**}\ p\ q$
shows $\textit{lc}\ q = \textit{lc}\ p$
⟨*proof*⟩

lemma *sig-red-rtrancl-lt*:
assumes $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q$
shows $\textit{lt}\ q \preceq_t \textit{lt}\ p$
⟨*proof*⟩

lemma *sig-red-tail-rtrancl-lt-rep-list*:
assumes $(\textit{sig-red sing-reg}\ (\prec)\ F)^{**}\ p\ q$
shows $\textit{punit.lt}\ (\textit{rep-list}\ q) = \textit{punit.lt}\ (\textit{rep-list}\ p)$
⟨*proof*⟩

lemma *sig-red-tail-rtrancl-lc-rep-list*:
assumes $(\textit{sig-red sing-reg}\ (\prec)\ F)^{**}\ p\ q$
shows $\textit{punit.lc}\ (\textit{rep-list}\ q) = \textit{punit.lc}\ (\textit{rep-list}\ p)$
⟨*proof*⟩

lemma *sig-red-rtrancl-lt-rep-list*:
assumes $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q$
shows $\textit{punit.lt}\ (\textit{rep-list}\ q) \preceq \textit{punit.lt}\ (\textit{rep-list}\ p)$
⟨*proof*⟩

lemma *sig-red-red-rtrancl*:
assumes $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q$
shows $(\textit{punit.red}\ (\textit{rep-list}\ \text{`}\ F))^{**}\ (\textit{rep-list}\ p)\ (\textit{rep-list}\ q)$
⟨*proof*⟩

lemma *sig-red-rtrancl-monom-mult*:
assumes $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q$
shows $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ (\textit{monom-mult}\ c\ s\ p)\ (\textit{monom-mult}\ c\ s\ q)$
⟨*proof*⟩

lemma *sig-red-rtrancl-sing-regI*: $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q \Longrightarrow (\textit{sig-red}\ (\preceq_t)\ \textit{top-tail}\ F)^{**}\ p\ q$
⟨*proof*⟩

lemma *sig-red-rtrancl-top-tailI*: $(\textit{sig-red sing-reg top-tail}\ F)^{**}\ p\ q \Longrightarrow (\textit{sig-red sing-reg}\ (\preceq)\ F)^{**}\ p\ q$
⟨*proof*⟩

lemma *dgrad-sig-set-closed-sig-red-rtrancl*:
assumes *dickson-grading* d **and** $F \subseteq \textit{dgrad-sig-set}\ d$ **and** $p \in \textit{dgrad-sig-set}\ d$
and $(\textit{sig-red sing-red top-tail}\ F)^{**}\ p\ q$
shows $q \in \textit{dgrad-sig-set}\ d$

⟨*proof*⟩

lemma *sig-red-rtrancl-mono*:
assumes (*sig-red sing-reg top-tail F*)** *p q* **and** $F \subseteq F'$
shows (*sig-red sing-reg top-tail F'*)** *p q*
⟨*proof*⟩

lemma *sig-red-rtrancl-subset*:
assumes (*sig-red sing-reg top-tail F*)** *p q* **and** *sing-reg* = $(\preceq_t)$ ∨ *sing-reg* = $(\prec_t)$
shows (*sig-red sing-reg top-tail* {*f*∈*F*. *sing-reg* (*lt f*) (*lt p*)})** *p q*
⟨*proof*⟩

lemma *is-sig-red-is-red*: *is-sig-red sing-reg top-tail F p* ⟹ *punit.is-red* (*rep-list* ‘ *F*) (*rep-list p*)
⟨*proof*⟩

lemma *is-sig-red-monom-mult*:
assumes *is-sig-red sing-reg top-tail F p* **and** $c \neq 0$
shows *is-sig-red sing-reg top-tail F* (*monom-mult c s p*)
⟨*proof*⟩

lemma *is-sig-red-sing-reg-cases*:
is-sig-red $(\preceq_t)$ *top-tail F p* = (*is-sig-red* (=) *top-tail F p* ∨ *is-sig-red* $(\prec_t)$ *top-tail F p*)
⟨*proof*⟩

corollary *is-sig-red-sing-regI*: *is-sig-red sing-reg top-tail F p* ⟹ *is-sig-red* $(\preceq_t)$ *top-tail F p*
⟨*proof*⟩

lemma *is-sig-red-top-tail-cases*:
is-sig-red sing-reg $(\preceq)$ *F p* = (*is-sig-red sing-reg* (=) *F p* ∨ *is-sig-red sing-reg* $(\prec)$ *F p*)
⟨*proof*⟩

corollary *is-sig-red-top-tailI*: *is-sig-red sing-reg top-tail F p* ⟹ *is-sig-red sing-reg* $(\preceq)$ *F p*
⟨*proof*⟩

lemma *is-sig-red-singletonI*:
assumes *is-sig-red sing-reg top-tail F r*
obtains *f* **where** $f \in F$ **and** *is-sig-red sing-reg top-tail* {*f*} *r*
⟨*proof*⟩

lemma *is-sig-red-singletonD*:
assumes *is-sig-red sing-reg top-tail* {*f*} *r* **and** $f \in F$
shows *is-sig-red sing-reg top-tail F r*
⟨*proof*⟩

lemma *is-sig-redD1*:
assumes *is-sig-red sing-reg top-tail F p*
shows *ord-term-lin.is-le-rel sing-reg*
⟨*proof*⟩

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

lemma *is-sig-red-addsI*:
assumes $f \in F$ **and** $t \in$ *keys* (*rep-list p*) **and** *rep-list f* $\neq$ *0* **and** *punit.lt* (*rep-list f*) *adds t*
and *ord-term-lin.is-le-rel sing-reg* **and** *ordered-powerprod-lin.is-le-rel top-tail*
and *sing-reg* ($t \oplus$ *lt f*) (*punit.lt* (*rep-list f*) $\oplus$ *lt p*) **and** *top-tail t* (*punit.lt* (*rep-list p*))
shows *is-sig-red sing-reg top-tail F p*
⟨*proof*⟩

lemma *is-sig-red-addsE*:
assumes *is-sig-red sing-reg top-tail F p*
obtains *f t* **where** $f \in F$ **and** $t \in$ *keys* (*rep-list p*) **and** *rep-list f* $\neq$ *0*
and *punit.lt* (*rep-list f*) *adds t*
and *sing-reg* ($t \oplus$ *lt f*) (*punit.lt* (*rep-list f*) $\oplus$ *lt p*)
and *top-tail t* (*punit.lt* (*rep-list p*))
⟨*proof*⟩

lemma *is-sig-red-top-addsI*:
assumes $f \in F$ **and** *rep-list f* $\neq$ *0* **and** *rep-list p* $\neq$ *0*
and *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*) **and** *ord-term-lin.is-le-rel sing-reg*
and *sing-reg* (*punit.lt* (*rep-list p*) $\oplus$ *lt f*) (*punit.lt* (*rep-list f*) $\oplus$ *lt p*)
shows *is-sig-red sing-reg* (=) *F p*
⟨*proof*⟩

lemma *is-sig-red-top-addsE*:
assumes *is-sig-red sing-reg* (=) *F p*
obtains *f* **where** $f \in F$ **and** *rep-list f* $\neq$ *0* **and** *rep-list p* $\neq$ *0*
and *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*)
and *sing-reg* (*punit.lt* (*rep-list p*) $\oplus$ *lt f*) (*punit.lt* (*rep-list f*) $\oplus$ *lt p*)
⟨*proof*⟩

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

punit.lc (*rep-list* q) $= 0 \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *is-sig-red-singleton-monom-multD*:
assumes *is-sig-red sing-reg top-tail* {*monom-mult c t f*} *p*
shows *is-sig-red sing-reg top-tail* {*f*} *p*
⟨*proof*⟩

lemma *is-sig-red-top-singleton-monom-multI*:
assumes *is-sig-red sing-reg* $(=)$ {*f*} *p* **and** $c \neq 0$
and *t adds punit.lt* (*rep-list p*) $-$ *punit.lt* (*rep-list f*)
shows *is-sig-red sing-reg* $(=)$ {*monom-mult c t f*} *p*
⟨*proof*⟩

lemma *is-sig-red-cong'*:
assumes *is-sig-red sing-reg top-tail F p* **and** *lt p* $=$ *lt q* **and** *rep-list p* $=$ *rep-list q*
shows *is-sig-red sing-reg top-tail F q*
⟨*proof*⟩

lemma *is-sig-red-cong*:
lt p $=$ *lt q* $\Longrightarrow$ *rep-list p* $=$ *rep-list q* $\Longrightarrow$
is-sig-red sing-reg top-tail F p $\longleftrightarrow$ *is-sig-red sing-reg top-tail F q*
⟨*proof*⟩

lemma *is-sig-red-top-cong*:
assumes *is-sig-red sing-reg* $(=)$ *F p* **and** *rep-list q* $\neq 0$ **and** *lt p* $=$ *lt q*
and *punit.lt* (*rep-list p*) $=$ *punit.lt* (*rep-list q*)
shows *is-sig-red sing-reg* $(=)$ *F q*
⟨*proof*⟩

lemma *sig-irredE-dgrad-max-set*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-max-set d*
obtains *q* **where** (*sig-red sing-reg top-tail F*)** *p q* **and** $\neg$ *is-sig-red sing-reg top-tail F q*
⟨*proof*⟩

lemma *is-sig-red-mono*:
is-sig-red sing-reg top-tail F p $\Longrightarrow F \subseteq F' \Longrightarrow$ *is-sig-red sing-reg top-tail F' p*
⟨*proof*⟩

lemma *is-sig-red-Un*:
is-sig-red sing-reg top-tail $(A \cup B)$ *p* $\longleftrightarrow$ (*is-sig-red sing-reg top-tail A p* $\vee$ *is-sig-red sing-reg top-tail B p*)
⟨*proof*⟩

lemma *is-sig-redD-lt*:
assumes *is-sig-red* $(\preceq_t)$ *top-tail* {*f*} *p*

shows $lt\ f \preceq_t lt\ p$
⟨*proof*⟩

lemma *is-sig-red-regularD-lt*:
assumes *is-sig-red* $(\prec_t)$ *top-tail* $\{f\}$ *p*
shows $lt\ f \prec_t lt\ p$
⟨*proof*⟩

lemma *sig-irred-regular-self*: $\neg$ *is-sig-red* $(\prec_t)$ *top-tail* $\{p\}$ *p*
⟨*proof*⟩

4.2.2 Signature Gröbner Bases

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

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

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

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

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

lemma *sig-red-zeroI*:
assumes (*sig-red sing-reg* $(\preceq)$ $F)^{**}$ *r s* **and** *rep-list* $s = 0$
shows *sig-red-zero sing-reg F r*
⟨*proof*⟩

lemma *sig-red-zeroE*:
assumes *sig-red-zero sing-reg F r*
obtains *s* **where** (*sig-red sing-reg* $(\preceq)$ $F)^{**}$ *r s* **and** *rep-list* $s = 0$
⟨*proof*⟩

lemma *sig-red-zero-monom-mult*:
assumes *sig-red-zero sing-reg F r*
shows *sig-red-zero sing-reg F* (*monom-mult c t r*)
⟨*proof*⟩

lemma *sig-red-zero-sing-regI*:
assumes *sig-red-zero sing-reg G p*
shows *sig-red-zero* ($\preceq_t$) *G p*
⟨*proof*⟩

lemma *sig-red-zero-nonzero*:
assumes *sig-red-zero sing-reg F r* **and** *rep-list r* $\neq$ *0* **and** *sing-reg* = ($\preceq_t$) $\vee$ *sing-reg* = ($\prec_t$)
shows *is-sig-red sing-reg* (=) *F r*
⟨*proof*⟩

lemma *sig-red-zero-mono*: *sig-red-zero sing-reg F p* $\Longrightarrow$ *F* $\subseteq$ *F'* $\Longrightarrow$ *sig-red-zero sing-reg F' p*
⟨*proof*⟩

lemma *sig-red-zero-subset*:
assumes *sig-red-zero sing-reg F p* **and** *sing-reg* = ($\preceq_t$) $\vee$ *sing-reg* = ($\prec_t$)
shows *sig-red-zero sing-reg* {*f*∈*F*. *sing-reg* (*lt f*) (*lt p*)} *p*
⟨*proof*⟩

lemma *sig-red-zero-idealI*:
assumes *sig-red-zero sing-reg F p*
shows *rep-list p* ∈ *ideal* (*rep-list* ‘ *F*)
⟨*proof*⟩

lemma *is-sig-GB-inI*:
assumes $\bigwedge$*r*. *lt r* = *u* $\Longrightarrow$ *r* ∈ *dgrad-sig-set d* $\Longrightarrow$ *sig-red-zero* ($\preceq_t$) *G r*
shows *is-sig-GB-in d G u*
⟨*proof*⟩

lemma *is-sig-GB-inD*:
assumes *is-sig-GB-in d G u* **and** *r* ∈ *dgrad-sig-set d* **and** *lt r* = *u*
shows *sig-red-zero* ($\preceq_t$) *G r*
⟨*proof*⟩

lemma *is-sig-GB-inI-triv*:
assumes ¬ *d* (*pp-of-term u*) ≤ *dgrad-max d* ∨ ¬ *component-of-term u* < *length fs*
shows *is-sig-GB-in d G u*
⟨*proof*⟩

lemma *is-sig-GB-in-mono*: *is-sig-GB-in d G u* $\Longrightarrow$ *G* $\subseteq$ *G'* $\Longrightarrow$ *is-sig-GB-in d G' u*
⟨*proof*⟩

lemma *is-sig-GB-uptI*:
  assumes $G \subseteq$ *dgrad-sig-set* d
    and $\bigwedge v.\ v \prec_t u \Longrightarrow d$ (*pp-of-term* v) $\leq$ *dgrad-max* $d \Longrightarrow$ *component-of-term* $v <$ *length fs* $\Longrightarrow$
          is-sig-GB-in d G v
  shows *is-sig-GB-upt* d G u
  ⟨*proof*⟩

lemma *is-sig-GB-uptD1*:
  assumes *is-sig-GB-upt* d G u
  shows $G \subseteq$ *dgrad-sig-set* d
  ⟨*proof*⟩

lemma *is-sig-GB-uptD2*:
  assumes *is-sig-GB-upt* d G u **and** $v \prec_t u$
  shows *is-sig-GB-in* d G v
  ⟨*proof*⟩

lemma *is-sig-GB-uptD3*:
  assumes *is-sig-GB-upt* d G u **and** $r \in$ *dgrad-sig-set* d **and** *lt* $r \prec_t u$
  shows *sig-red-zero* $(\preceq_t)$ G r
  ⟨*proof*⟩

lemma *is-sig-GB-upt-le*:
  assumes *is-sig-GB-upt* d G u **and** $v \preceq_t u$
  shows *is-sig-GB-upt* d G v
⟨*proof*⟩

lemma *is-sig-GB-upt-mono*:
  is-sig-GB-upt d G $u \Longrightarrow G \subseteq G' \Longrightarrow G' \subseteq$ *dgrad-sig-set* $d \Longrightarrow$ *is-sig-GB-upt* d G' u
  ⟨*proof*⟩

lemma *is-sig-GB-upt-is-Groebner-basis*:
  assumes *dickson-grading* d **and** *hom-grading* d **and** $G \subseteq$ *dgrad-sig-set'* j d
    and $\bigwedge u.$ *component-of-term* $u < j \Longrightarrow$ *is-sig-GB-in* d G u
  shows *punit.is-Groebner-basis* (*rep-list* ‘ G)
  ⟨*proof*⟩

lemma *is-sig-GB-is-Groebner-basis*:
  assumes *dickson-grading* d **and** *hom-grading* d **and** $G \subseteq$ *dgrad-max-set* d **and** $\bigwedge u.$ *is-sig-GB-in* d G u
  shows *punit.is-Groebner-basis* (*rep-list* ‘ G)
  ⟨*proof*⟩

lemma *sig-red-zero-is-red*:
  assumes *sig-red-zero sing-reg* F r **and** *rep-list* $r \neq 0$
  shows *is-sig-red sing-reg* $(\preceq)$ F r

⟨*proof*⟩

lemma *is-sig-red-sing-top-is-red-zero*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G u* **and** $a \in$ *dgrad-sig-set d* **and** *lt a* = *u*
and *is-sig-red* (=) (=) *G a* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G a*
shows *sig-red-zero* ($\preceq_t$) *G a*
⟨*proof*⟩

lemma *sig-regular-reduced-unique*:
assumes *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
and *lt p* = *lt q* **and** *lc p* = *lc q* **and** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) *G p* **and** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) *G q*
shows *rep-list p* = *rep-list q*
⟨*proof*⟩

corollary *sig-regular-reduced-unique'*:
assumes *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
and *lt p* = *lt q* **and** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) *G p* **and** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) *G q*
shows *punit.monom-mult* (*lc q*) *0* (*rep-list p*) = *punit.monom-mult* (*lc p*) *0* (*rep-list q*)
⟨*proof*⟩

lemma *sig-regular-top-reduced-lt-lc-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
and *lt p* = *lt q* **and** (p = *0*) $\longleftrightarrow$ (q = *0*) **and** ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) ∧ *lc q* ∗ *punit.lc* (*rep-list p*) = *lc p* ∗ *punit.lc* (*rep-list q*)
⟨*proof*⟩

corollary *sig-regular-top-reduced-lt-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d*
and $q \in$ *dgrad-sig-set d* **and** *lt p* = *lt q* **and** $p \neq$ *0* **and** $q \neq$ *0*
and ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*)
⟨*proof*⟩

corollary *sig-regular-top-reduced-lc-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
and *lt p* = *lt q* **and** *lc p* = *lc q* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lc* (*rep-list p*) = *punit.lc* (*rep-list q*)
⟨*proof*⟩

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

lemma *is-min-sig-GB-minimal*:
assumes *is-min-sig-GB d G* **and** $G' \subseteq$ *dgrad-sig-set d*
and $\bigwedge u$. *d* (*pp-of-term u*) $\leq$ *dgrad-max d* $\Longrightarrow$ *component-of-term u* $<$ *length fs* $\Longrightarrow$ *is-sig-GB-in d* G' *u*
and $g \in G$ **and** *rep-list g* $\neq$ *0*
obtains g' **where** $g' \in G'$ **and** *rep-list* $g' \neq$ *0* **and** *lt* g' = *lt g*
and *punit.lt* (*rep-list* g') = *punit.lt* (*rep-list g*)
⟨*proof*⟩

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

lemma *is-syz-sigI*:
assumes $s \neq$ *0* **and** *lt s* = *u* **and** $s \in$ *dgrad-sig-set d* **and** *rep-list s* = *0*
shows *is-syz-sig d u*
⟨*proof*⟩

lemma *is-syz-sigE*:
assumes *is-syz-sig d u*
obtains *r* **where** $r \neq$ *0* **and** *lt r* = *u* **and** $r \in$ *dgrad-sig-set d* **and** *rep-list r* = *0*
⟨*proof*⟩

lemma *is-syz-sig-adds*:
assumes *dickson-grading d* **and** *is-syz-sig d u* **and** *u* $adds_t$ *v*
and *d* (*pp-of-term v*) $\leq$ *dgrad-max d*
shows *is-syz-sig d v*
⟨*proof*⟩

lemma *syzygy-crit*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G u* **and** *is-syz-sig d u*
and $p \in$ *dgrad-sig-set d* **and** *lt p* = *u*
shows *sig-red-zero* ($\prec_t$) *G p*
⟨*proof*⟩

lemma *lemma-21*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt p*) **and** $p \in$ *dgrad-sig-set d* **and** $g \in G$
and *rep-list p* $\neq$ *0* **and** *rep-list g* $\neq$ *0* **and** *lt g* $adds_t$ *lt p*
and *punit.lt* (*rep-list g*) *adds punit.lt* (*rep-list p*)
shows *is-sig-red* ($\preceq_t$) (=) *G p*
⟨*proof*⟩

4.2.3 Rewrite Bases

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

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

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

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

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

lemma *is-rewrite-ordD1*: $\mathit{is\text{-}rewrite\text{-}ord}\ rword \Longrightarrow rword\ a\ a$
⟨*proof*⟩

lemma *is-rewrite-ordD2*: $\mathit{is\text{-}rewrite\text{-}ord}\ rword \Longrightarrow rword\ a\ b \Longrightarrow rword\ b\ c \Longrightarrow rword\ a\ c$
⟨*proof*⟩

lemma *is-rewrite-ordD3*:
assumes *is-rewrite-ord rword*
and *rword a b* $\Longrightarrow$ *thesis*
and $\neg$ *rword a b* $\Longrightarrow$ *rword b a* $\Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *is-rewrite-ordD4*:
assumes *is-rewrite-ord rword* **and** *rword a b* **and** *rword b a*
shows *fst a* = *fst b*
⟨*proof*⟩

lemma *is-rewrite-ordD4'*:
assumes *is-rewrite-ord rword* **and** *rword* (*spp-of a*) (*spp-of b*) **and** *rword* (*spp-of b*) (*spp-of a*)
shows *lt a* = *lt b*
⟨*proof*⟩

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

lemma *is-canon-rewriterI*:
assumes $p \in A$ **and** $p \neq 0$ **and** *lt p* $adds_t$ *u*
and $\bigwedge a.\ a \in A \Longrightarrow a \neq 0 \Longrightarrow$ *lt a* $adds_t$ *u* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of p*)
shows *is-canon-rewriter rword A u p*
⟨*proof*⟩

lemma *is-canon-rewriterD1*: *is-canon-rewriter rword A u p* $\Longrightarrow p \in A$
⟨*proof*⟩

lemma *is-canon-rewriterD2*: *is-canon-rewriter rword A u p* $\Longrightarrow p \neq 0$
⟨*proof*⟩

lemma *is-canon-rewriterD3*: *is-canon-rewriter rword A u p* $\Longrightarrow$ *lt p* $adds_t$ *u*
⟨*proof*⟩

lemma *is-canon-rewriterD4*:
is-canon-rewriter rword A u p $\Longrightarrow a \in A \Longrightarrow a \neq 0 \Longrightarrow$ *lt a* $adds_t$ *u* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of p*)
⟨*proof*⟩

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

lemma *is-rewrite-ord-finite-canon-rewriterE*:
assumes *is-rewrite-ord rword* **and** *finite A* **and** $a \in A$ **and** $a \neq 0$ **and** *lt a* $adds_t$ *u*
obtains *p* **where** *is-canon-rewriter rword A u p*
⟨*proof*⟩

lemma *is-rewrite-ord-canon-rewriterD1*:
assumes *is-rewrite-ord rword* **and** *is-canon-rewriter rword A u p* **and** *is-canon-rewriter rword A v q*
and *lt p* $adds_t$ *v* **and** *lt q* $adds_t$ *u*
shows *lt p = lt q*
⟨*proof*⟩

corollary *is-rewrite-ord-canon-rewriterD2*:
assumes *is-rewrite-ord rword* **and** *is-canon-rewriter rword A u p* **and** *is-canon-rewriter rword A u q*
shows *lt p = lt q*
⟨*proof*⟩

lemma *is-rewrite-ord-canon-rewriterD3*:
assumes *is-rewrite-ord rword* **and** *dickson-grading d* **and** *is-canon-rewriter rword A u p*
and $a \in A$ **and** $a \neq 0$ **and** *lt a* $adds_t$ *u* **and** *is-sig-GB-upt d A (lt a)*
and *lt p* $adds_t$ *lt a* **and** ¬ *is-sig-red* $(\prec_t)$ (=) *A a*
shows *lt p = lt a*
⟨*proof*⟩

lemma *is-RB-inI1*:
assumes *is-canon-rewriter rword G u g* **and** ¬ *is-sig-red* $(\prec_t)$ (=) *G* (*monom-mult 1* (*pp-of-term u* − *lp g*) *g*)
shows *is-RB-in d rword G u*
⟨*proof*⟩

lemma *is-RB-inI2*:
assumes *is-syz-sig d u*
shows *is-RB-in d rword G u*
⟨*proof*⟩

lemma *is-RB-inE*:
assumes *is-RB-in d rword G u*
and *is-syz-sig d u* ⟹ *thesis*
and ⋀*g*. ¬ *is-syz-sig d u* ⟹ *is-canon-rewriter rword G u g* ⟹
¬ *is-sig-red* $(\prec_t)$ (=) *G* (*monom-mult 1* (*pp-of-term u* − *lp g*) *g*) ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *is-RB-inD*:

assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-sig-set d* **and** *is-RB-in d rword G u*
and $\neg$ *is-syz-sig d u* **and** *d (pp-of-term u)* $\leq$ *dgrad-max d*
and *is-canon-rewriter rword G u g*
shows *rep-list g* $\neq$ *0*
⟨*proof*⟩

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

lemma *is-RB-uptD1*:
assumes *is-RB-upt d canon G u*
shows $G \subseteq$ *dgrad-sig-set d*
⟨*proof*⟩

lemma *is-RB-uptD2*:
assumes *is-RB-upt d canon G u* **and** $v \prec_t u$ **and** *d (pp-of-term v)* $\leq$ *dgrad-max d*
and *component-of-term v* $<$ *length fs*
shows *is-RB-in d canon G v*
⟨*proof*⟩

lemma *is-RB-in-UnI*:
assumes *is-RB-in d rword G u* **and** $\bigwedge h.\ h \in H \Longrightarrow u \prec_t$ *lt h*
shows *is-RB-in d rword* $(H \cup G)$ *u*
⟨*proof*⟩

corollary *is-RB-in-insertI*:
assumes *is-RB-in d rword G u* **and** $u \prec_t$ *lt g*
shows *is-RB-in d rword (insert g G) u*
⟨*proof*⟩

corollary *is-RB-upt-UnI*:
assumes *is-RB-upt d rword G u* **and** $H \subseteq$ *dgrad-sig-set d* **and** $\bigwedge h.\ h \in H \Longrightarrow u \preceq_t$ *lt h*
shows *is-RB-upt d rword* $(H \cup G)$ *u*
⟨*proof*⟩

corollary *is-RB-upt-insertI*:
assumes *is-RB-upt d rword G u* **and** $g \in$ *dgrad-sig-set d* **and** $u \preceq_t$ *lt g*
shows *is-RB-upt d rword (insert g G) u*
⟨*proof*⟩

lemma *is-RB-upt-is-sig-GB-upt*:
assumes *dickson-grading d* **and** *is-RB-upt d rword G u*

shows *is-sig-GB-upt d G u*
⟨*proof*⟩

corollary *is-RB-upt-is-syz-sigD*:
assumes *dickson-grading d* **and** *is-RB-upt d rword G u*
and *is-syz-sig d u* **and** $p \in$ *dgrad-sig-set d* **and** *lt* $p = u$
shows *sig-red-zero* $(\prec_t)$ *G p*
⟨*proof*⟩

4.2.4 S-Pairs

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

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

lemma *rep-list-spair*: *rep-list* (*spair p q*) = *punit.spoly* (*rep-list p*) (*rep-list q*)
⟨*proof*⟩

lemma *spair-comm*: *spair p q* = − *spair q p*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-spair*:
assumes *dickson-grading d* **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
shows *spair p q* ∈ *dgrad-sig-set d*
⟨*proof*⟩

lemma *lt-spair*:
assumes *rep-list p* ≠ *0* **and** *punit.lt* (*rep-list p*) ⊕ *lt q* $\prec_t$ *punit.lt* (*rep-list q*) ⊕ *lt p*
shows *lt* (*spair p q*) = (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*
⟨*proof*⟩

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

lemma *lt-rep-list-spair*:

assumes *rep-list* $p \neq 0$ **and** *rep-list* $q \neq 0$ **and** *rep-list* (*spair* p q) $\neq 0$
and a + *punit.lt* (*rep-list* p) = b + *punit.lt* (*rep-list* q)
shows *punit.lt* (*rep-list* (*spair* p q)) $\prec$ (a − *gcs* a b) + *punit.lt* (*rep-list* p)
⟨*proof*⟩

lemma *is-regular-spair-sym*: *is-regular-spair* p q $\Longrightarrow$ *is-regular-spair* q p
⟨*proof*⟩

lemma *is-regular-spairI*:
assumes *rep-list* $p \neq 0$ **and** *rep-list* $q \neq 0$
and *punit.lt* (*rep-list* q) $\oplus$ *lt* p $\neq$ *punit.lt* (*rep-list* p) $\oplus$ *lt* q
shows *is-regular-spair* p q
⟨*proof*⟩

lemma *is-regular-spairI'*:
assumes *rep-list* $p \neq 0$ **and** *rep-list* $q \neq 0$
and a + *punit.lt* (*rep-list* p) = b + *punit.lt* (*rep-list* q) **and** $a \oplus$ *lt* $p \neq b \oplus$ *lt* q
shows *is-regular-spair* p q
⟨*proof*⟩

lemma *is-regular-spairD1*: *is-regular-spair* p q $\Longrightarrow$ *rep-list* $p \neq 0$
⟨*proof*⟩

lemma *is-regular-spairD2*: *is-regular-spair* p q $\Longrightarrow$ *rep-list* $q \neq 0$
⟨*proof*⟩

lemma *is-regular-spairD3*:
fixes p q
defines *t1* $\equiv$ *punit.lt* (*rep-list* p)
defines *t2* $\equiv$ *punit.lt* (*rep-list* q)
assumes *is-regular-spair* p q
shows *t2* $\oplus$ *lt* p $\neq$ *t1* $\oplus$ *lt* q (**is** *?thesis1*)
and *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list* p)) (*lcs t1 t2* − *t1*) p) $\neq$
lt (*monom-mult* (*1* / *punit.lc* (*rep-list* q)) (*lcs t1 t2* − *t2*) q) (**is** *?l* $\neq$ *?r*)
⟨*proof*⟩

lemma *is-regular-spair-nonzero*: *is-regular-spair* p q $\Longrightarrow$ *spair* p q $\neq 0$
⟨*proof*⟩

lemma *is-regular-spair-lt*:
assumes *is-regular-spair* p q
shows *lt* (*spair* p q) = *ord-term-lin.max*
((*lcs* (*punit.lt* (*rep-list* p)) (*punit.lt* (*rep-list* q)) − *punit.lt* (*rep-list* p))
$\oplus$ *lt* p)
((*lcs* (*punit.lt* (*rep-list* p)) (*punit.lt* (*rep-list* q)) − *punit.lt* (*rep-list* q))
$\oplus$ *lt* q)
⟨*proof*⟩

lemma *is-regular-spair-lt-ge-1*:

assumes *is-regular-spair p q*
shows $lt\ p \preceq_t lt\ (spair\ p\ q)$
$\langle proof \rangle$

corollary *is-regular-spair-lt-ge-2*:
assumes *is-regular-spair p q*
shows $lt\ q \preceq_t lt\ (spair\ p\ q)$
$\langle proof \rangle$

lemma *is-regular-spair-component-lt-cases*:
assumes *is-regular-spair p q*
shows *component-of-term* (*lt* (*spair p q*)) = *component-of-term* (*lt p*) ∨
component-of-term (*lt* (*spair p q*)) = *component-of-term* (*lt q*)
$\langle proof \rangle$

lemma *lemma-9*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** *is-RB-upt d rword G u*
and *inj-on lt G* **and** ¬ *is-syz-sig d u* **and** *is-canon-rewriter rword G u g1* **and** $h \in G$
and *is-sig-red* ($\prec_t$) (=) {*h*} (*monom-mult 1* (*pp-of-term u* − *lp g1*) *g1*)
and *d* (*pp-of-term u*) ≤ *dgrad-max d*
shows *lcs* (*punit.lt* (*rep-list g1*)) (*punit.lt* (*rep-list h*)) − *punit.lt* (*rep-list g1*) =
pp-of-term u − *lp g1* (**is** *?thesis1*)
and *lcs* (*punit.lt* (*rep-list g1*)) (*punit.lt* (*rep-list h*)) − *punit.lt* (*rep-list h*) =
((*pp-of-term u* − *lp g1*) + *punit.lt* (*rep-list g1*)) − *punit.lt* (*rep-list h*) (**is** *?thesis2*)
and *is-regular-spair g1 h* (**is** *?thesis3*)
and *lt* (*spair g1 h*) = *u* (**is** *?thesis4*)
$\langle proof \rangle$

lemma *is-RB-upt-finite*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** $G \subseteq$ *dgrad-sig-set d* **and** *inj-on lt G*
and *finite G*
and $\bigwedge$*g1 g2*. $g1 \in G \Longrightarrow g2 \in G \Longrightarrow$ *is-regular-spair g1 g2* $\Longrightarrow$ *lt* (*spair g1 g2*) $\prec_t u \Longrightarrow$
is-RB-in d rword G (*lt* (*spair g1 g2*))
and $\bigwedge$*i*. *i* < *length fs* $\Longrightarrow$ *term-of-pair* (*0*, *i*) $\prec_t u \Longrightarrow$ *is-RB-in d rword G* (*term-of-pair* (*0*, *i*))
shows *is-RB-upt d rword G u*
$\langle proof \rangle$

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

lemma *lemma-11*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** *is-RB-upt d rword G* (*lt p*)
and $p \in$ *dgrad-sig-set d* **and** *is-sig-red* ($\prec_t$) (=) *G p*

obtains $u\ g$ **where** $u \prec_t lt\ p$ **and** $d\ (pp\text{-}of\text{-}term\ u) \le dgrad\text{-}max\ d$ **and** $component\text{-}of\text{-}term\ u < length\ fs$
and $\neg\ is\text{-}syz\text{-}sig\ d\ u$ **and** $is\text{-}canon\text{-}rewriter\ rword\ G\ u\ g$
and $u = (punit.lt\ (rep\text{-}list\ p) - punit.lt\ (rep\text{-}list\ g)) \oplus lt\ g$ **and** $is\text{-}sig\text{-}red\ (\prec_t)\ (=)\ \{g\}\ p$
$\langle proof \rangle$

4.2.5 Termination

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

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

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

lemma *term-pp-rel-refl*: $reflp\ r \Longrightarrow term\text{-}pp\text{-}rel\ r\ a\ a$
$\langle proof \rangle$

lemma *term-pp-rel-irrefl*: $irreflp\ r \Longrightarrow \neg\ term\text{-}pp\text{-}rel\ r\ a\ a$
$\langle proof \rangle$

lemma *term-pp-rel-sym*: $symp\ r \Longrightarrow term\text{-}pp\text{-}rel\ r\ a\ b \Longrightarrow term\text{-}pp\text{-}rel\ r\ b\ a$
$\langle proof \rangle$

lemma *term-pp-rel-trans*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ r$ **and** $term\text{-}pp\text{-}rel\ r\ a\ b$ **and** $term\text{-}pp\text{-}rel\ r\ b\ c$
shows $term\text{-}pp\text{-}rel\ r\ a\ c$
$\langle proof \rangle$

lemma *term-pp-rel-trans-eq-left*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ r$ **and** $term\text{-}pp\text{-}rel\ (=)\ a\ b$ **and** $term\text{-}pp\text{-}rel\ r\ b\ c$
shows $term\text{-}pp\text{-}rel\ r\ a\ c$
$\langle proof \rangle$

lemma *term-pp-rel-trans-eq-right*:
assumes $ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ r$ **and** $term\text{-}pp\text{-}rel\ r\ a\ b$ **and** $term\text{-}pp\text{-}rel\ (=)\ b\ c$
shows $term\text{-}pp\text{-}rel\ r\ a\ c$
$\langle proof \rangle$

lemma *canon-term-pp-cancel*: $canon\text{-}term\text{-}pp\text{-}pair\ (cancel\text{-}term\text{-}pp\text{-}pair\ a)$
$\langle proof \rangle$

lemma *term-pp-rel-cancel*:
assumes $reflp\ r$
shows $term\text{-}pp\text{-}rel\ r\ a\ (cancel\text{-}term\text{-}pp\text{-}pair\ a)$

$\langle proof \rangle$

lemma *canon-term-pp-rel-id*:
assumes *term-pp-rel* $(=)$ *a b* **and** *canon-term-pp-pair a* **and** *canon-term-pp-pair b*
shows $a = b$
$\langle proof \rangle$

lemma *min-set-finite*:
fixes *seq* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b::field)$
assumes *dickson-grading d* **and** *range seq* $\subseteq$ *dgrad-sig-set d* **and** $0 \notin$ *rep-list* ' *range seq*
and $\bigwedge i\ j.\ i < j \Longrightarrow lt\ (seq\ i) \prec_t lt\ (seq\ j)$
shows *finite* $\{i.\ \neg\ (\exists j<i.\ lt\ (seq\ j)\ adds_t\ lt\ (seq\ i)\ \wedge$
punit.lt (*rep-list* (*seq j*)) *adds punit.lt* (*rep-list* (*seq i*)))$\}$
$\langle proof \rangle$

lemma *rb-termination*:
fixes *seq* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b::field)$
assumes *dickson-grading d* **and** *range seq* $\subseteq$ *dgrad-sig-set d* **and** $0 \notin$ *rep-list* ' *range seq*
and $\bigwedge i\ j.\ i < j \Longrightarrow lt\ (seq\ i) \prec_t lt\ (seq\ j)$
and $\bigwedge i.\ \neg$ *is-sig-red* $(\prec_t)$ $(\preceq)$ (*seq* ' $\{0..<i\}$) (*seq i*)
and $\bigwedge i.\ (\exists j<length\ fs.\ lt\ (seq\ i) = lt\ (monomial\ (1::'b)\ (term\text{-}of\text{-}pair\ (0,\ j)))\ \wedge$
punit.lt (*rep-list* (*seq i*)) $\preceq$ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* $(0,\ j)$))))) $\vee$
$(\exists j\ k.$ *is-regular-spair* (*seq j*) (*seq k*) $\wedge$ *rep-list* (*spair* (*seq j*) (*seq k*)) $\neq 0\ \wedge$
lt (*seq i*) $=$ *lt* (*spair* (*seq j*) (*seq k*)) $\wedge$
punit.lt (*rep-list* (*seq i*)) $\preceq$ *punit.lt* (*rep-list* (*spair* (*seq j*) (*seq k*))))
and $\bigwedge i.$ *is-sig-GB-upt d* (*seq* ' $\{0..<i\}$) (*lt* (*seq i*))
shows *thesis*
$\langle proof \rangle$

4.2.6 Concrete Rewrite Orders

definition *is-strict-rewrite-ord* :: $(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow bool$
where *is-strict-rewrite-ord rel* $\longleftrightarrow$ *is-rewrite-ord* $(\lambda x\ y.\ \neg\ rel\ y\ x)$

lemma *is-strict-rewrite-ordI*: *is-rewrite-ord* $(\lambda x\ y.\ \neg\ rel\ y\ x) \Longrightarrow$ *is-strict-rewrite-ord rel*
$\langle proof \rangle$

lemma *is-strict-rewrite-ordD*: *is-strict-rewrite-ord rel* $\Longrightarrow$ *is-rewrite-ord* $(\lambda x\ y.\ \neg\ rel\ y\ x)$
$\langle proof \rangle$

lemma *is-strict-rewrite-ord-antisym*:
  assumes *is-strict-rewrite-ord rel* **and** $\neg$ *rel x y* **and** $\neg$ *rel y x*
  shows *fst x = fst y*
  ⟨*proof*⟩

lemma *is-strict-rewrite-ord-asym*:
  assumes *is-strict-rewrite-ord rel* **and** *rel x y*
  shows $\neg$ *rel y x*
⟨*proof*⟩

lemma *is-strict-rewrite-ord-irrefl*: *is-strict-rewrite-ord rel* $\Longrightarrow \neg$ *rel x x*
  ⟨*proof*⟩

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

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

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

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

lemma *rw-rat-alt*: *rw-rat* = (λ*p q*. $\neg$ *rw-rat-strict q p*)
  ⟨*proof*⟩

lemma *rw-rat-is-rewrite-ord*: *is-rewrite-ord rw-rat*
⟨*proof*⟩

lemma *rw-rat-strict-is-strict-rewrite-ord*: *is-strict-rewrite-ord rw-rat-strict*
⟨*proof*⟩

lemma *rw-add-alt*: *rw-add* = (λ*p q*. $\neg$ *rw-add-strict q p*)
  ⟨*proof*⟩

lemma *rw-add-is-rewrite-ord*: *is-rewrite-ord rw-add*
⟨*proof*⟩

lemma *rw-add-strict-is-strict-rewrite-ord*: *is-strict-rewrite-ord rw-add-strict*
⟨*proof*⟩

4.2.7 Preparations for Sig-Poly-Pairs

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

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

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

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

lemma *spp-inv-spp-of*:
  assumes $r \neq 0$ **and** $r \in$ *dgrad-sig-set dgrad*
  shows *spp-inv* (*spp-of r*)
  ⟨*proof*⟩

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

lemma *sig-poly-rel-vec-of*: *spp-rel sp* (*vec-of sp*)
⟨*proof*⟩

lemma *vec-of-nonzero*: *vec-of sp* $\neq$ *0*
  ⟨*proof*⟩

lemma *lt-vec-of*: *lt* (*vec-of sp*) = *fst sp*
  ⟨*proof*⟩

lemma *rep-list-vec-of*: *rep-list* (*vec-of sp*) = *snd sp*
  ⟨*proof*⟩

lemma *spp-of-vec-of*: *spp-of* (*vec-of sp*) = *sp*
  ⟨*proof*⟩

end

lemma *map-spp-of-vec-of*:
  assumes *list-all spp-inv sps*
  shows *map* (*spp-of* $\circ$ *vec-of*) *sps* = *sps*
⟨*proof*⟩

lemma *vec-of-dgrad-sig-set*: *vec-of sp* $\in$ *dgrad-sig-set dgrad*
⟨*proof*⟩

lemma *spp-invD-fst*:
assumes *spp-inv sp*
shows *dgrad* (*pp-of-term* (*fst sp*)) $\leq$ *dgrad-max dgrad* **and** *component-of-term* (*fst sp*) $<$ *length fs*
⟨*proof*⟩

lemma *spp-invD-snd*:
assumes *dickson-grading dgrad* **and** *spp-inv sp*
shows *snd sp* $\in$ *punit-dgrad-max-set dgrad*
⟨*proof*⟩

lemma *vec-of-inj*:
assumes *spp-inv sp* **and** *vec-of sp* = *vec-of sp'*
shows *sp* = *sp'*
⟨*proof*⟩

lemma *spp-inv-alt*: *spp-inv sp* $\longleftrightarrow$ (*vec-of sp* $\neq$ *0*)
⟨*proof*⟩

lemma *spp-of-vec-of-spp-of*:
assumes *p* $\in$ *dgrad-sig-set dgrad*
shows *spp-of* (*vec-of* (*spp-of p*)) = *spp-of p*
⟨*proof*⟩

4.2.8 Total Reduction

primrec *find-sig-reducer* :: ('*t* $\times$ ('*a* $\Rightarrow_0$ '*b*)) *list* $\Rightarrow$ '*t* $\Rightarrow$ '*a* $\Rightarrow$ *nat* $\Rightarrow$ *nat option*
where
find-sig-reducer [] - - - = *None*|
find-sig-reducer (*b* # *bs*) *u t i* =
(*if snd b* $\neq$ *0* $\wedge$ *punit.lt* (*snd b*) *adds t* $\wedge$ (*t* $-$ *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u then Some i*
else find-sig-reducer bs u t (*Suc i*))

lemma *find-sig-reducer-SomeD-aux*:
assumes *find-sig-reducer bs u t i* = *Some j*
shows *i* $\leq$ *j* **and** *j* $-$ *i* $<$ *length bs*
⟨*proof*⟩

lemma *find-sig-reducer-SomeD'*:
assumes *find-sig-reducer bs u t i* = *Some j* **and** *b* = *bs* ! (*j* $-$ *i*)
shows *b* $\in$ *set bs* **and** *snd b* $\neq$ *0* **and** *punit.lt* (*snd b*) *adds t* **and** (*t* $-$ *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u*
⟨*proof*⟩

corollary *find-sig-reducer-SomeD*:
assumes *find-sig-reducer* (*map spp-of bs*) *u t 0* = *Some i*
shows *i* $<$ *length bs* **and** *rep-list* (*bs* ! *i*) $\neq$ *0* **and** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds*

t
  and $(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i))) \oplus lt\ (bs\ !\ i) \prec_t u$
⟨*proof*⟩

lemma *find-sig-reducer-NoneE*:
  assumes *find-sig-reducer bs u t i = None* **and** $b \in set\ bs$
  assumes $snd\ b = 0 \Longrightarrow thesis$ **and** $snd\ b \neq 0 \Longrightarrow \neg\ punit.lt\ (snd\ b)\ adds\ t \Longrightarrow thesis$
  and $snd\ b \neq 0 \Longrightarrow punit.lt\ (snd\ b)\ adds\ t \Longrightarrow \neg\ (t - punit.lt\ (snd\ b)) \oplus fst\ b \prec_t u \Longrightarrow thesis$
  shows *thesis*
  ⟨*proof*⟩

lemma *find-sig-reducer-SomeD-red-single*:
  assumes $t \in keys\ (rep\text{-}list\ p)$ **and** $find\text{-}sig\text{-}reducer\ (map\ spp\text{-}of\ bs)\ (lt\ p)\ t\ 0 = Some\ i$
  shows $sig\text{-}red\text{-}single\ (\prec_t)\ (\preceq)\ p\ (p - monom\text{-}mult\ (lookup\ (rep\text{-}list\ p)\ t\ /\ punit.lc\ (rep\text{-}list\ (bs\ !\ i)))$
      $(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))\ (bs\ !\ i))\ (bs\ !\ i)\ (t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))$
⟨*proof*⟩

corollary *find-sig-reducer-SomeD-red*:
  assumes $t \in keys\ (rep\text{-}list\ p)$ **and** $find\text{-}sig\text{-}reducer\ (map\ spp\text{-}of\ bs)\ (lt\ p)\ t\ 0 = Some\ i$
  shows $sig\text{-}red\ (\prec_t)\ (\preceq)\ (set\ bs)\ p\ (p - monom\text{-}mult\ (lookup\ (rep\text{-}list\ p)\ t\ /\ punit.lc\ (rep\text{-}list\ (bs\ !\ i)))$
      $(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))\ (bs\ !\ i))$
  ⟨*proof*⟩

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

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

lemma *sig-trd-term-wf*:
  assumes *dickson-grading d*
  shows *wf (sig-trd-term d)*
⟨*proof*⟩

function (*domintros*) $sig\text{-}trd\text{-}aux :: ('a \times ('t \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b)$ **where**
  $sig\text{-}trd\text{-}aux\ (t, p) =$

(*let* p' =
(*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*
None ⇒ *p*
| *Some i* ⇒ *p* − *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*));
p'' = *punit.lower* (*rep-list* p') *t in*
if p'' = *0 then* p' *else sig-trd-aux* (*punit.lt* p'', p'))
⟨*proof*⟩

lemma *sig-trd-aux-domI*:
assumes *fst args0* ∈ *keys* (*rep-list* (*snd args0*))
shows *sig-trd-aux-dom args0*
⟨*proof*⟩

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

lemma *sig-trd-aux-red-rtrancl*:
assumes *fst args0* ∈ *keys* (*rep-list* (*snd args0*))
shows (*sig-red* ($\prec_t$) (⪯) (*set bs*))** (*snd args0*) (*sig-trd-aux args0*)
⟨*proof*⟩

corollary *sig-trd-red-rtrancl*: (*sig-red* ($\prec_t$) (⪯) (*set bs*))** *p* (*sig-trd p*)
⟨*proof*⟩

lemma *sig-trd-aux-irred*:
assumes *fst args0* ∈ *keys* (*rep-list* (*snd args0*))
and ⋀*b s*. *b* ∈ *set bs* ⟹ *rep-list b* ≠ *0* ⟹ *fst args0* ≺ *s* + *punit.lt* (*rep-list b*) ⟹
s ⊕ *lt b* $\prec_t$ *lt* (*snd* (*args0*)) ⟹ *lookup* (*rep-list* (*snd args0*)) (*s* + *punit.lt* (*rep-list b*)) = *0*
shows ¬ *is-sig-red* ($\prec_t$) (⪯) (*set bs*) (*sig-trd-aux args0*)
⟨*proof*⟩

corollary *sig-trd-irred*: ¬ *is-sig-red* ($\prec_t$) (⪯) (*set bs*) (*sig-trd p*)
⟨*proof*⟩

end

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

context
fixes *v* :: $'t$
begin

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

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

lemma *sig-trd-spp-aux-simps* [*code*]:
sig-trd-spp-aux (*p*, *r*) = (*if p* = *0 then r else sig-trd-spp-aux* (*sig-trd-spp-body* (*p*, *r*)))
⟨*proof*⟩

end

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

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

end

4.2.9 Koszul Syzygies

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

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

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

fun *new-syz-sigs* :: *'t list* ⇒ $('t \Rightarrow_0 'b)$ *list* ⇒ $(('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b))$ + *nat* ⇒ *'t list*
where
new-syz-sigs ss bs (*Inl* (*a*, *b*)) = *ss* |
new-syz-sigs ss bs (*Inr j*) =

(if *is-pot-ord then*
filter-min-append ($adds_t$) *ss* (*filter-min* ($adds_t$) (*map* ($\lambda b.$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)) *bs*))
else ss)

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

lemma *Koszul-syz-sigs-auxI*:
assumes $i < j$ **and** $j <$ *length bs*
shows *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i*), $k + j$)) (*term-of-pair* (*punit.lt* (*bs* ! *j*), $k + i$)) $\in$
set (*Koszul-syz-sigs-aux bs k*)
⟨*proof*⟩

lemma *Koszul-syz-sigs-auxE*:
assumes $v \in$ *set* (*Koszul-syz-sigs-aux bs k*)
obtains *i j* **where** $i < j$ **and** $j <$ *length bs*
and $v =$ *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i*), $k + j$)) (*term-of-pair* (*punit.lt* (*bs* ! *j*), $k + i$))
⟨*proof*⟩

lemma *lt-Koszul-syz-comp*:
assumes $0 \notin$ *set fs* **and** $i <$ *length fs*
shows *lt* ((*fs* ! *i*) $\odot$ *monomial 1* (*term-of-pair* (*0*, *j*))) = *term-of-pair* (*punit.lt* (*fs* ! *i*), *j*)
⟨*proof*⟩

lemma *Koszul-syz-nonzero-lt*:
assumes *rep-list a* $\neq$ *0* **and** *rep-list b* $\neq$ *0* **and** *component-of-term* (*lt a*) < *component-of-term* (*lt b*)
shows *rep-list a* $\odot$ *b* $-$ *rep-list b* $\odot$ *a* $\neq$ *0* (**is** *?p* $-$ *?q* $\neq$ *0*)
and *lt* (*rep-list a* $\odot$ *b* $-$ *rep-list b* $\odot$ *a*) =
ord-term-lin.max (*punit.lt* (*rep-list a*) $\oplus$ *lt b*) (*punit.lt* (*rep-list b*) $\oplus$ *lt a*) (**is** - = *?r*)
⟨*proof*⟩

lemma *Koszul-syz-is-syz*: *rep-list* (*rep-list a* $\odot$ *b* $-$ *rep-list b* $\odot$ *a*) = *0*
⟨*proof*⟩

lemma *dgrad-sig-set-closed-Koszul-syz*:
assumes *dickson-grading dgrad* **and** $a \in$ *dgrad-sig-set dgrad* **and** $b \in$ *dgrad-sig-set*

dgrad
shows *rep-list a* ⊙ *b* − *rep-list b* ⊙ *a* ∈ *dgrad-sig-set dgrad*
⟨*proof*⟩

corollary *Koszul-syz-is-syz-sig*:
assumes *dickson-grading dgrad* **and** *a* ∈ *dgrad-sig-set dgrad* **and** *b* ∈ *dgrad-sig-set dgrad*
and *rep-list a* ≠ *0* **and** *rep-list b* ≠ *0* **and** *component-of-term* (*lt a*) < *component-of-term* (*lt b*)
shows *is-syz-sig dgrad* (*ord-term-lin.max* (*punit.lt* (*rep-list a*) ⊕ *lt b*) (*punit.lt* (*rep-list b*) ⊕ *lt a*))
⟨*proof*⟩

corollary *lt-Koszul-syz-in-Koszul-syz-sigs-aux*:
assumes *distinct fs* **and** *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length fs*
shows *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*))) ∈
set (*Koszul-syz-sigs-aux fs 0*) (**is** *?l* ∈ *?K*)
⟨*proof*⟩

corollary *lt-Koszul-syz-in-Koszul-syz-sigs*:
assumes ¬ *is-pot-ord* **and** *distinct fs* **and** *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length fs*
obtains *v* **where** *v* ∈ *set* (*Koszul-syz-sigs fs*)
and *v* $adds_t$ *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*)))
⟨*proof*⟩

lemma *lt-Koszul-syz-init*:
assumes *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length fs*
shows *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*))) =
ord-term-lin.max (*term-of-pair* (*punit.lt* (*fs* ! *i*), *j*)) (*term-of-pair* (*punit.lt* (*fs* ! *j*), *i*))
(**is** *lt* (*?p* − *?q*) = *?r*)
⟨*proof*⟩

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

corollary *Koszul-syz-sigs-is-syz-sig*:
assumes *dickson-grading dgrad* **and** *distinct fs* **and** *0* ∉ *set fs* **and** *v* ∈ *set* (*Koszul-syz-sigs fs*)
shows *is-syz-sig dgrad v*
⟨*proof*⟩

lemma *Koszul-syz-sigs-minimal*:
assumes $u \in set\ (Koszul\text{-}syz\text{-}sigs\ fs)$ **and** $v \in set\ (Koszul\text{-}syz\text{-}sigs\ fs)$ **and** $u\ adds_t\ v$
shows $u = v$
⟨*proof*⟩

lemma *Koszul-syz-sigs-distinct*: *distinct* (*Koszul-syz-sigs fs*)
⟨*proof*⟩

4.2.10 Algorithms

definition *spair-spp* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b))$
where $spair\text{-}spp\ p\ q = (let\ t1 = punit.lt\ (snd\ p);\ t2 = punit.lt\ (snd\ q);\ l = lcs\ t1\ t2\ in$
$(ord\text{-}term\text{-}lin.max\ ((l - t1) \oplus fst\ p)\ ((l - t2) \oplus fst\ q),$
$punit.monom\text{-}mult\ (1\ /\ punit.lc\ (snd\ p))\ (l - t1)\ (snd\ p) -$
$punit.monom\text{-}mult\ (1\ /\ punit.lc\ (snd\ q))\ (l - t2)\ (snd\ q)))$

definition *is-regular-spair-spp* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool$
where $is\text{-}regular\text{-}spair\text{-}spp\ p\ q \longleftrightarrow$
$(snd\ p \neq 0 \land snd\ q \neq 0 \land punit.lt\ (snd\ q) \oplus fst\ p \neq punit.lt\ (snd\ p) \oplus fst\ q)$

definition *spair-sigs* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \times 't)$
where $spair\text{-}sigs\ p\ q =$
$(let\ t1 = punit.lt\ (rep\text{-}list\ p);\ t2 = punit.lt\ (rep\text{-}list\ q);\ l = lcs\ t1\ t2\ in$
$((l - t1) \oplus lt\ p,\ (l - t2) \oplus lt\ q))$

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

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

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

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

sig-of-pair (*Inr j*) = *term-of-pair* (*0*, *j*)

fun *sig-of-pair-spp* :: ((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) ⇒ 't
where
sig-of-pair-spp (*Inl* (*p*, *q*)) = (*let* (*u*, *v*) = *spair-sigs-spp p q in ord-term-lin.max u v*) |
sig-of-pair-spp (*Inr j*) = *term-of-pair* (*0*, *j*)

definition *pair-ord* :: ((('t ⇒$_0$ 'b) × ('t ⇒$_0$ 'b)) + *nat*) ⇒ ((('t ⇒$_0$ 'b) × ('t ⇒$_0$ 'b)) + *nat*) ⇒ *bool*
where *pair-ord x y* ⟷ (*sig-of-pair x* $\preceq_t$ *sig-of-pair y*)

definition *pair-ord-spp* :: ((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) ⇒
((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) ⇒ *bool*
where *pair-ord-spp x y* ⟷ (*sig-of-pair-spp x* $\preceq_t$ *sig-of-pair-spp y*)

primrec *new-spairs* :: ('t ⇒$_0$ 'b) *list* ⇒ ('t ⇒$_0$ 'b) ⇒ ((('t ⇒$_0$ 'b) × ('t ⇒$_0$ 'b)) + *nat*) *list* **where**
new-spairs [] *p* = [] |
new-spairs (*b* # *bs*) *p* =
(*if is-regular-spair p b then insort-wrt pair-ord* (*Inl* (*p*, *b*)) (*new-spairs bs p*) *else new-spairs bs p*)

primrec *new-spairs-spp* :: ('t × ('a ⇒$_0$ 'b)) *list* ⇒ ('t × ('a ⇒$_0$ 'b)) ⇒
((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) *list* **where**
new-spairs-spp [] *p* = [] |
new-spairs-spp (*b* # *bs*) *p* =
(*if is-regular-spair-spp p b then*
insort-wrt pair-ord-spp (*Inl* (*p*, *b*)) (*new-spairs-spp bs p*)
else new-spairs-spp bs p)

definition *add-spairs* :: ((('t ⇒$_0$ 'b) × ('t ⇒$_0$ 'b)) + *nat*) *list* ⇒ ('t ⇒$_0$ 'b) *list* ⇒ ('t ⇒$_0$ 'b) ⇒
((('t ⇒$_0$ 'b) × ('t ⇒$_0$ 'b)) + *nat*) *list*
where *add-spairs ps bs p* = *merge-wrt pair-ord* (*new-spairs bs p*) *ps*

definition *add-spairs-spp* :: ((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) *list* ⇒
('t × ('a ⇒$_0$ 'b)) *list* ⇒ ('t × ('a ⇒$_0$ 'b)) ⇒
((('t × ('a ⇒$_0$ 'b)) × ('t × ('a ⇒$_0$ 'b))) + *nat*) *list*
where *add-spairs-spp ps bs p* = *merge-wrt pair-ord-spp* (*new-spairs-spp bs p*) *ps*

lemma *spair-alt-spair-sigs*:
spair p q = *monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*pp-of-term* (*fst* (*spair-sigs p q*)) − *lp p*) *p* −
monom-mult (*1* / *punit.lc* (*rep-list q*)) (*pp-of-term* (*snd* (*spair-sigs p q*)) − *lp q*) *q*
⟨*proof*⟩

lemma *sig-of-spair*:

assumes *is-regular-spair p q*
shows *sig-of-pair* (*Inl* (*p*, *q*)) = *lt* (*spair p q*)
⟨*proof*⟩

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

lemma *in-new-spairsI*:
assumes $b \in$ *set bs* **and** *is-regular-spair p b*
shows *Inl* (*p*, *b*) $\in$ *set* (*new-spairs bs p*)
⟨*proof*⟩

lemma *in-new-spairsD*:
assumes *Inl* (*a*, *b*) $\in$ *set* (*new-spairs bs p*)
shows $a = p$ **and** $b \in$ *set bs* **and** *is-regular-spair p b*
⟨*proof*⟩

corollary *in-new-spairs-iff*:
Inl (*p*, *b*) $\in$ *set* (*new-spairs bs p*) $\longleftrightarrow$ ($b \in$ *set bs* $\wedge$ *is-regular-spair p b*)
⟨*proof*⟩

lemma *Inr-not-in-new-spairs*: *Inr j* $\notin$ *set* (*new-spairs bs p*)
⟨*proof*⟩

lemma *sum-prodE*:
assumes $\bigwedge a\ b.\ p = Inl\ (a,\ b) \Longrightarrow thesis$ **and** $\bigwedge j.\ p = Inr\ j \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

corollary *in-new-spairsE*:
assumes $q \in$ *set* (*new-spairs bs p*)
obtains *b* **where** $b \in$ *set bs* **and** *is-regular-spair p b* **and** $q = Inl\ (p,\ b)$
⟨*proof*⟩

lemma *new-spairs-sorted*: *sorted-wrt pair-ord* (*new-spairs bs p*)
⟨*proof*⟩

lemma *sorted-add-spairs*:
assumes *sorted-wrt pair-ord ps*
shows *sorted-wrt pair-ord* (*add-spairs ps bs p*)
⟨*proof*⟩

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

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

definition *is-pred-syz* :: $'t$ *list* $\Rightarrow$ $'t \Rightarrow$ *bool*
  where *is-pred-syz ss u* = ($\exists$ *v*$\in$*set ss. v adds*$_t$ *u*)

definition *is-rewritable* :: ($'t \Rightarrow_0$ $'b$) *list* $\Rightarrow$ ($'t \Rightarrow_0$ $'b$) $\Rightarrow$ $'t \Rightarrow$ *bool*
  where *is-rewritable bs p u* = ($\exists$ *b*$\in$*set bs. b* $\neq$ *0* $\wedge$ *lt b adds*$_t$ *u* $\wedge$ *rword-strict* (*spp-of p*) (*spp-of b*))

definition *is-rewritable-spp* :: ($'t \times$ ($'a \Rightarrow_0$ $'b$)) *list* $\Rightarrow$ ($'t \times$ ($'a \Rightarrow_0$ $'b$)) $\Rightarrow$ $'t \Rightarrow$ *bool*
  where *is-rewritable-spp bs p u* = ($\exists$ *b*$\in$*set bs. fst b adds*$_t$ *u* $\wedge$ *rword-strict p b*)

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

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

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

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

fun *rb-spp-body* ::
      ((($'t \times$ ($'a \Rightarrow_0$ $'b$)) *list* $\times$ $'t$ *list* $\times$ ((($'t \times$ ($'a \Rightarrow_0$ $'b$)) $\times$ ($'t \times$ ($'a \Rightarrow_0$ $'b$))) + *nat*) *list*) $\times$ *nat*) $\Rightarrow$
       ((($'t \times$ ($'a \Rightarrow_0$ $'b$)) *list* $\times$ $'t$ *list* $\times$ ((($'t \times$ ($'a \Rightarrow_0$ $'b$)) $\times$ ($'t \times$ ($'a \Rightarrow_0$ $'b$))) + *nat*) *list*) $\times$ *nat*)
  where
  rb-spp-body ((*bs*, *ss*, []), *z*) = ((*bs*, *ss*, []), *z*) |
  rb-spp-body ((*bs*, *ss*, *p* # *ps*), *z*) =
    (*let ss'* = *new-syz-sigs-spp ss bs p in*
      if sig-crit-spp bs ss' p then
          ((*bs*, *ss'*, *ps*), *z*)

else
let $p' = $ *sig-trd-spp bs* (*spp-of-pair p*) *in*
if snd $p' = 0$ *then*
((*bs*, *fst* p' # ss', *ps*), *Suc z*)
else
((p' # *bs*, ss', *add-spairs-spp ps bs* p'), *z*))

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

lemma *rb-spp-aux-Nil* [*code*]: *rb-spp-aux* ((*bs*, *ss*, []), *z*) = ((*bs*, *ss*, []), *z*)
⟨*proof*⟩

lemma *rb-spp-aux-Cons* [*code*]:
rb-spp-aux ((*bs*, *ss*, *p* # *ps*), *z*) = *rb-spp-aux* (*rb-spp-body* ((*bs*, *ss*, *p* # *ps*), *z*))
⟨*proof*⟩

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

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

lemma *rword*: *is-rewrite-ord rword*
⟨*proof*⟩

lemma *sig-crit'-sym*: *sig-crit' bs* (*Inl* (*p*, *q*)) $\Longrightarrow$ *sig-crit' bs* (*Inl* (*q*, *p*))
⟨*proof*⟩

lemma *is-rewritable-ConsD*:
assumes *is-rewritable* (*b* # *bs*) *p u* **and** $u \prec_t$ *lt b*
shows *is-rewritable bs p u*
⟨*proof*⟩

lemma *sig-crit'-ConsD*:
assumes *sig-crit'* (*b* # *bs*) *p* **and** *sig-of-pair p* $\prec_t$ *lt b*
shows *sig-crit' bs p*
⟨*proof*⟩

definition *rb-aux-inv1* :: $('t \Rightarrow_0 'b)$ *list* $\Rightarrow$ *bool*
where *rb-aux-inv1 bs* =
(*set bs* $\subseteq$ *dgrad-sig-set dgrad* $\wedge$ $0 \notin$ *rep-list* ‘ *set bs* $\wedge$

sorted-wrt $(\lambda x\ y.\ lt\ y \prec_t lt\ x)$ *bs* $\wedge$
$(\forall\, i{<}length\ bs.\ \neg\ is\text{-}sig\text{-}red\ (\prec_t)\ (\preceq)\ (set\ (drop\ (Suc\ i)\ bs))\ (bs\ !\ i))\ \wedge$
$(\forall\, i{<}length\ bs.$
$((\exists\, j{<}length\ fs.\ lt\ (bs\ !\ i) = lt\ (monomial\ (1{::}'b)\ (term\text{-}of\text{-}pair\ (0,\ j)))\ \wedge$
$punit.lt\ (rep\text{-}list\ (bs\ !\ i)) \preceq punit.lt\ (rep\text{-}list\ (monomial\ 1\ (term\text{-}of\text{-}pair\ (0,\ j))))) \vee$
$(\exists\, p{\in}set\ bs.\ \exists\, q{\in}set\ bs.\ is\text{-}regular\text{-}spair\ p\ q \wedge rep\text{-}list\ (spair\ p\ q) \neq 0\ \wedge$
$lt\ (bs\ !\ i) = lt\ (spair\ p\ q) \wedge punit.lt\ (rep\text{-}list\ (bs\ !\ i)) \preceq punit.lt\ (rep\text{-}list\ (spair\ p\ q))))) \wedge$
$(\forall\, i{<}length\ bs.\ is\text{-}RB\text{-}upt\ dgrad\ rword\ (set\ (drop\ (Suc\ i)\ bs))\ (lt\ (bs\ !\ i))))$

fun *rb-aux-inv* $:: (('t \Rightarrow_0 'b)\ list \times 't\ list \times ((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat)\ list) \Rightarrow bool$
where $rb\text{-}aux\text{-}inv\ (bs,\ ss,\ ps) =$
$(rb\text{-}aux\text{-}inv1\ bs\ \wedge$
$(\forall\, u{\in}set\ ss.\ is\text{-}syz\text{-}sig\ dgrad\ u) \wedge$
$(\forall\, p\ q.\ Inl\ (p,\ q) \in set\ ps \longrightarrow (is\text{-}regular\text{-}spair\ p\ q \wedge p \in set\ bs \wedge q \in set\ bs)) \wedge$
$(\forall\, j.\ Inr\ j \in set\ ps \longrightarrow (j < length\ fs \wedge (\forall\, b{\in}set\ bs.\ lt\ b \prec_t term\text{-}of\text{-}pair\ (0,\ j)) \wedge$
$length\ (filter\ (\lambda q.\ sig\text{-}of\text{-}pair\ q = term\text{-}of\text{-}pair\ (0,\ j))\ ps) \leq 1) \wedge$
$(sorted\text{-}wrt\ pair\text{-}ord\ ps) \wedge$
$(\forall\, p{\in}set\ ps.\ (\forall\, b1{\in}set\ bs.\ \forall\, b2{\in}set\ bs.\ is\text{-}regular\text{-}spair\ b1\ b2 \longrightarrow$
$sig\text{-}of\text{-}pair\ p \prec_t lt\ (spair\ b1\ b2) \longrightarrow (Inl\ (b1,\ b2) \in set\ ps \vee Inl\ (b2,\ b1) \in set\ ps)) \wedge$
$(\forall\, j{<}length\ fs.\ sig\text{-}of\text{-}pair\ p \prec_t term\text{-}of\text{-}pair\ (0,\ j) \longrightarrow Inr\ j \in set\ ps)) \wedge$
$(\forall\, b{\in}set\ bs.\ \forall\, p{\in}set\ ps.\ lt\ b \preceq_t sig\text{-}of\text{-}pair\ p) \wedge$
$(\forall\, a{\in}set\ bs.\ \forall\, b{\in}set\ bs.\ is\text{-}regular\text{-}spair\ a\ b \longrightarrow Inl\ (a,\ b) \notin set\ ps \longrightarrow Inl\ (b,\ a) \notin set\ ps \longrightarrow$
$\neg\ is\text{-}RB\text{-}in\ dgrad\ rword\ (set\ bs)\ (lt\ (spair\ a\ b)) \longrightarrow$
$(\exists\, p{\in}set\ ps.\ sig\text{-}of\text{-}pair\ p = lt\ (spair\ a\ b) \wedge \neg\ sig\text{-}crit'\ bs\ p)) \wedge$
$(\forall\, j{<}length\ fs.\ Inr\ j \notin set\ ps \longrightarrow (is\text{-}RB\text{-}in\ dgrad\ rword\ (set\ bs)\ (term\text{-}of\text{-}pair\ (0,\ j)) \wedge$
$rep\text{-}list\ (monomial\ (1{::}'b)\ (term\text{-}of\text{-}pair\ (0,\ j))) \in ideal\ (rep\text{-}list\ `\ set\ bs))))$

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

lemma *rb-aux-inv1-D1*: $rb\text{-}aux\text{-}inv1\ bs \Longrightarrow set\ bs \subseteq dgrad\text{-}sig\text{-}set\ dgrad$
⟨*proof*⟩

lemma *rb-aux-inv1-D2*: $rb\text{-}aux\text{-}inv1\ bs \Longrightarrow 0 \notin rep\text{-}list\ `\ set\ bs$
⟨*proof*⟩

lemma *rb-aux-inv1-D3*: $rb\text{-}aux\text{-}inv1\ bs \Longrightarrow sorted\text{-}wrt\ (\lambda x\ y.\ lt\ y \prec_t lt\ x)\ bs$
⟨*proof*⟩

lemma *rb-aux-inv1-D4*:
rb-aux-inv1 bs $\Longrightarrow$ *i* $<$ *length bs* $\Longrightarrow$ $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i*) *bs*)) (*bs* ! *i*)
⟨*proof*⟩

lemma *rb-aux-inv1-D5*:
rb-aux-inv1 bs $\Longrightarrow$ *i* $<$ *length bs* $\Longrightarrow$ *is-RB-upt dgrad rword* (*set* (*drop* (*Suc i*) *bs*)) (*lt* (*bs* ! *i*))
⟨*proof*⟩

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

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

lemma *rb-aux-inv1-distinct-lt*:
assumes *rb-aux-inv1 bs*
shows *distinct* (*map lt bs*)
⟨*proof*⟩

corollary *rb-aux-inv1-lt-inj-on*:
assumes *rb-aux-inv1 bs*
shows *inj-on lt* (*set bs*)
⟨*proof*⟩

lemma *canon-rewriter-unique*:
assumes *rb-aux-inv1 bs* **and** *is-canon-rewriter rword* (*set bs*) *u a*
and *is-canon-rewriter rword* (*set bs*) *u b*
shows *a* = *b*
⟨*proof*⟩

lemma *rb-aux-inv-D1*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *rb-aux-inv1 bs*
⟨*proof*⟩

lemma *rb-aux-inv-D2*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *u* $\in$ *set ss* $\Longrightarrow$ *is-syz-sig dgrad u*
⟨*proof*⟩

lemma *rb-aux-inv-D3*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *Inl* (*p*, *q*) $\in$ *set ps*
  shows *p* $\in$ *set bs* **and** *q* $\in$ *set bs* **and** *is-regular-spair p q*
  ⟨*proof*⟩

lemma *rb-aux-inv-D4*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *Inr j* $\in$ *set ps*
  shows *j* < *length fs* **and** $\bigwedge$*b*. *b* $\in$ *set bs* $\Longrightarrow$ *lt b* $\prec_t$ *term-of-pair* (*0*, *j*)
    and *length* (*filter* (λ*q*. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) *ps*) $\leq$ *1*
  ⟨*proof*⟩

lemma *rb-aux-inv-D5*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *sorted-wrt pair-ord ps*
  ⟨*proof*⟩

lemma *rb-aux-inv-D6-1*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *p* $\in$ *set ps* **and** *b1* $\in$ *set bs* **and** *b2* $\in$ *set bs*
    and *is-regular-spair b1 b2* **and** *sig-of-pair p* $\prec_t$ *lt* (*spair b1 b2*)
  obtains *Inl* (*b1*, *b2*) $\in$ *set ps* | *Inl* (*b2*, *b1*) $\in$ *set ps*
  ⟨*proof*⟩

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

lemma *rb-aux-inv-D7*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *b* $\in$ *set bs* $\Longrightarrow$ *p* $\in$ *set ps* $\Longrightarrow$ *lt b* $\preceq_t$ *sig-of-pair p*
  ⟨*proof*⟩

lemma *rb-aux-inv-D8*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *a* $\in$ *set bs* **and** *b* $\in$ *set bs* **and** *is-regular-spair a b*
    and *Inl* (*a*, *b*) $\notin$ *set ps* **and** *Inl* (*b*, *a*) $\notin$ *set ps* **and** $\neg$ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
  obtains *p* **where** *p* $\in$ *set ps* **and** *sig-of-pair p* = *lt* (*spair a b*) **and** $\neg$ *sig-crit′ bs p*
  ⟨*proof*⟩

lemma *rb-aux-inv-D9*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *j* < *length fs* **and** *Inr j* $\notin$ *set ps*
  shows *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*))
    and *rep-list* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ‘ *set bs*)
  ⟨*proof*⟩

lemma *rb-aux-inv-is-RB-upt*:
  assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** $\bigwedge$*p*. *p* $\in$ *set ps* $\Longrightarrow$ *u* $\preceq_t$ *sig-of-pair p*
  shows *is-RB-upt dgrad rword* (*set bs*) *u*

⟨*proof*⟩

lemma *rb-aux-inv-is-RB-upt-Cons*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
shows *is-RB-upt dgrad rword* (*set bs*) (*sig-of-pair p*)
⟨*proof*⟩

lemma *Inr-in-tailD*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** *Inr j* $\in$ *set ps*
shows *sig-of-pair p* $\neq$ *term-of-pair* (*0*, *j*)
⟨*proof*⟩

lemma *pair-list-aux*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *p* $\in$ *set ps*
shows *sig-of-pair p* = *lt* (*poly-of-pair p*) $\wedge$ *poly-of-pair p* $\neq$ *0* $\wedge$ *poly-of-pair p* $\in$ *dgrad-sig-set dgrad*
⟨*proof*⟩

corollary *pair-list-sig-of-pair*:
rb-aux-inv (*bs*, *ss*, *ps*) $\Longrightarrow$ *p* $\in$ *set ps* $\Longrightarrow$ *sig-of-pair p* = *lt* (*poly-of-pair p*)
⟨*proof*⟩

corollary *pair-list-nonzero*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *p* $\in$ *set ps* $\Longrightarrow$ *poly-of-pair p* $\neq$ *0*
⟨*proof*⟩

corollary *pair-list-dgrad-sig-set*:
rb-aux-inv (*bs*, *ss*, *ps*) $\Longrightarrow$ *p* $\in$ *set ps* $\Longrightarrow$ *poly-of-pair p* $\in$ *dgrad-sig-set dgrad*
⟨*proof*⟩

lemma *is-rewritableI-is-canon-rewriter*:
assumes *rb-aux-inv1 bs* **and** *b* $\in$ *set bs* **and** *b* $\neq$ *0* **and** *lt b* $adds_t$ *u*
and $\neg$ *is-canon-rewriter rword* (*set bs*) *u b*
shows *is-rewritable bs b u*
⟨*proof*⟩

lemma *is-rewritableD-is-canon-rewriter*:
assumes *rb-aux-inv1 bs* **and** *is-rewritable bs b u*
shows $\neg$ *is-canon-rewriter rword* (*set bs*) *u b*
⟨*proof*⟩

lemma *lemma-12*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *is-RB-upt dgrad rword* (*set bs*) *u*
and *dgrad* (*pp-of-term u*) $\leq$ *dgrad-max dgrad* **and** *is-canon-rewriter rword* (*set bs*) *u a*
and $\neg$ *is-syz-sig dgrad u* **and** *is-sig-red* ($\prec_t$) (=) (*set bs*) (*monom-mult 1* (*pp-of-term u* $-$ *lp a*) *a*)
obtains *p q* **where** *p* $\in$ *set bs* **and** *q* $\in$ *set bs* **and** *is-regular-spair p q* **and** *lt* (*spair p q*) = *u*

and $\neg$ *sig-crit′ bs* (*Inl* (*p*, *q*))
⟨*proof*⟩

lemma *is-canon-rewriterI-eq-sig*:
assumes *rb-aux-inv1 bs* **and** $b \in$ *set bs*
shows *is-canon-rewriter rword* (*set bs*) (*lt b*) *b*
⟨*proof*⟩

lemma *not-sig-crit*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** $\neg$ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** $b \in$ *set bs*
shows *lt b* $\prec_t$ *sig-of-pair p*
⟨*proof*⟩

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

lemma *rep-list-monomial′*: *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) = ((*fs* ! *j*) *when* $j <$ *length fs*)
⟨*proof*⟩

lemma *new-syz-sigs-is-syz-sig*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** $v \in$ *set* (*new-syz-sigs ss bs p*)
shows *is-syz-sig dgrad v*
⟨*proof*⟩

lemma *new-syz-sigs-minimal*:
assumes $\bigwedge u'\ v'.\ u' \in$ *set ss* $\Longrightarrow v' \in$ *set ss* $\Longrightarrow u'$ $adds_t$ $v' \Longrightarrow u' = v'$
assumes $u \in$ *set* (*new-syz-sigs ss bs p*) **and** $v \in$ *set* (*new-syz-sigs ss bs p*) **and** u $adds_t$ v
shows $u = v$
⟨*proof*⟩

lemma *new-syz-sigs-distinct*:
assumes *distinct ss*
shows *distinct* (*new-syz-sigs ss bs p*)
⟨*proof*⟩

lemma *sig-crit′I-sig-crit*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows *sig-crit′ bs p*
⟨*proof*⟩

lemma *rb-aux-inv-preserved-0*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
and $\bigwedge s.\ s \in$ *set ss′* $\Longrightarrow$ *is-syz-sig dgrad s*
and $\bigwedge a\ b.\ a \in$ *set bs* $\Longrightarrow b \in$ *set bs* $\Longrightarrow$ *is-regular-spair a b* $\Longrightarrow$ *Inl* (*a*, *b*) $\notin$

set ps ⟹
Inl (b, a) ∉ set ps ⟹ ¬ is-RB-in dgrad rword (set bs) (lt (spair a b)) ⟹
∃ q∈set ps. sig-of-pair q = lt (spair a b) ∧ ¬ sig-crit′ bs q
and ⋀j. j < length fs ⟹ p = Inr j ⟹ Inr j ∉ set ps ⟹ is-RB-in dgrad rword (set bs) (term-of-pair (0, j)) ∧
rep-list (monomial 1 (term-of-pair (0, j))) ∈ ideal (rep-list ‘ set bs)
shows rb-aux-inv (bs, ss′, ps)
⟨proof⟩

lemma rb-aux-inv-preserved-1:
assumes rb-aux-inv (bs, ss, p # ps) **and** sig-crit bs (new-syz-sigs ss bs p) p
shows rb-aux-inv (bs, new-syz-sigs ss bs p, ps)
⟨proof⟩

lemma rb-aux-inv-preserved-2:
assumes rb-aux-inv (bs, ss, p # ps) **and** rep-list (sig-trd bs (poly-of-pair p)) = 0
shows rb-aux-inv (bs, lt (sig-trd bs (poly-of-pair p)) # new-syz-sigs ss bs p, ps)
⟨proof⟩

lemma rb-aux-inv-preserved-3:
assumes rb-aux-inv (bs, ss, p # ps) **and** ¬ sig-crit bs (new-syz-sigs ss bs p) p
and rep-list (sig-trd bs (poly-of-pair p)) ≠ 0
shows rb-aux-inv ((sig-trd bs (poly-of-pair p)) # bs, new-syz-sigs ss bs p,
add-spairs ps bs (sig-trd bs (poly-of-pair p)))
and lt (sig-trd bs (poly-of-pair p)) ∉ lt ‘ set bs
⟨proof⟩

lemma rb-aux-inv-init: rb-aux-inv ([], Koszul-syz-sigs fs, map Inr [0..<length fs])
⟨proof⟩

corollary rb-aux-inv-init-fst:
rb-aux-inv (fst (([], Koszul-syz-sigs fs, map Inr [0..<length fs]), z))
⟨proof⟩

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

rb-aux $((p' \# bs, ss', \textit{add-spairs ps bs } p'), z))$
⟨*proof*⟩

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

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

Termination

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

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

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

lemma *wfp-on-rb-aux-term1*: *wfp-on* (λ*x y*. (*x*, *y*) ∈ *rb-aux-term1*) (*Collect rb-aux-inv1*)
⟨*proof*⟩

lemma *wfp-on-rb-aux-term2*: *wfp-on* (λ*x y*. (*x*, *y*) ∈ *rb-aux-term2*) (*Collect rb-aux-inv*)
⟨*proof*⟩

corollary *wf-rb-aux-term*: *wf rb-aux-term*
⟨*proof*⟩

lemma *rb-aux-domI*:
assumes *rb-aux-inv* (*fst args*)
shows *rb-aux-dom args*
⟨*proof*⟩

Invariant

lemma *rb-aux-inv-invariant*:
assumes *rb-aux-inv* (*fst args*)
shows *rb-aux-inv* (*fst* (*rb-aux args*))
⟨*proof*⟩

lemma *rb-aux-inv-last-Nil*:
assumes *rb-aux-dom args*
shows *snd* (*snd* (*fst* (*rb-aux args*))) = []
⟨*proof*⟩

corollary *rb-aux-shape*:
assumes *rb-aux-dom args*

obtains *bs ss z* **where** *rb-aux args* = ((*bs*, *ss*, []), *z*)
⟨*proof*⟩

lemma *rb-aux-is-RB-upt*:
is-RB-upt dgrad rword (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) *u*
⟨*proof*⟩

corollary *rb-is-RB-upt*: *is-RB-upt dgrad rword* (*set* (*fst rb*)) *u*
⟨*proof*⟩

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

corollary *rb-aux-is-sig-GB-in*:
is-sig-GB-in dgrad (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) *u*
⟨*proof*⟩

corollary *rb-aux-is-Groebner-basis*:
assumes *hom-grading dgrad*
shows *punit.is-Groebner-basis* (*set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))))
⟨*proof*⟩

lemma *ideal-rb-aux*:
ideal (*set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))) =
ideal (*set fs*) (**is** *ideal ?l* = *ideal ?r*)
⟨*proof*⟩

corollary *ideal-rb*: *ideal* (*rep-list* ‘ *set* (*fst rb*)) = *ideal* (*set fs*)
⟨*proof*⟩

lemma
shows *dgrad-max-set-closed-rb-aux*:
set (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) ⊆
punit-dgrad-max-set dgrad (**is** *?thesis1*)
and *rb-aux-nonzero*:
0 ∉ *set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))
(**is** *?thesis2*)
⟨*proof*⟩

4.2.11 Minimality of the Computed Basis

lemma *rb-aux-top-irred'*:
assumes $rword\text{-}strict = rw\text{-}rat\text{-}strict$ **and** $rb\text{-}aux\text{-}inv\ (bs,\ ss,\ p\ \#\ ps)$
and $\neg\ sig\text{-}crit\ bs\ (new\text{-}syz\text{-}sigs\ ss\ bs\ p)\ p$
shows $\neg\ is\text{-}sig\text{-}red\ (\preceq_t)\ (=)\ (set\ bs)\ (sig\text{-}trd\ bs\ (poly\text{-}of\text{-}pair\ p))$
$\langle proof \rangle$

lemma *rb-aux-top-irred*:
assumes $rword\text{-}strict = rw\text{-}rat\text{-}strict$ **and** $rb\text{-}aux\text{-}inv\ (fst\ args)$ **and** $b \in set\ (fst\ (fst\ (rb\text{-}aux\ args)))$
and $\bigwedge b0.\ b0 \in set\ (fst\ (fst\ args)) \Longrightarrow \neg\ is\text{-}sig\text{-}red\ (\preceq_t)\ (=)\ (set\ (fst\ (fst\ args)) - \{b0\})\ b0$
shows $\neg\ is\text{-}sig\text{-}red\ (\preceq_t)\ (=)\ (set\ (fst\ (fst\ (rb\text{-}aux\ args))) - \{b\})\ b$
$\langle proof \rangle$

corollary *rb-aux-is-min-sig-GB*:
assumes $rword\text{-}strict = rw\text{-}rat\text{-}strict$
shows $is\text{-}min\text{-}sig\text{-}GB\ dgrad\ (set\ (fst\ (fst\ (rb\text{-}aux\ (([],\ Koszul\text{-}syz\text{-}sigs\ fs,\ map\ Inr\ [0..<length\ fs]),\ z)))))$
(**is** $is\text{-}min\text{-}sig\text{-}GB$ - $(set\ (fst\ (fst\ (rb\text{-}aux\ ?args)))))$)
$\langle proof \rangle$

corollary *rb-is-min-sig-GB*:
assumes $rword\text{-}strict = rw\text{-}rat\text{-}strict$
shows $is\text{-}min\text{-}sig\text{-}GB\ dgrad\ (set\ (fst\ rb))$
$\langle proof \rangle$

4.2.12 No Zero-Reductions

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

lemma *rb-aux-inv2-D1*: $rb\text{-}aux\text{-}inv2\ args \Longrightarrow rb\text{-}aux\text{-}inv\ args$
$\langle proof \rangle$

lemma *rb-aux-inv2-D2*:
$rb\text{-}aux\text{-}inv2\ (bs,\ ss,\ ps) \Longrightarrow j < length\ fs \Longrightarrow Inr\ j \notin set\ ps \Longrightarrow$
$fs\ !\ j \in ideal\ (rep\text{-}list\ `\ set\ (filter\ (\lambda b.\ component\text{-}of\text{-}term\ (lt\ b) < Suc\ j)\ bs))$
$\langle proof \rangle$

lemma *rb-aux-inv2-E*:
assumes $rb\text{-}aux\text{-}inv2\ (bs,\ ss,\ ps)$ **and** $j < length\ fs$ **and** $Inr\ j \notin set\ ps$ **and** $b \in$

set bs
  and *component-of-term* (*lt b*) < *j*
 obtains *s* **where** $s \in$ *set ss* **and** *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
 ⟨*proof*⟩

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

lemma *sig-red-zero-filter*:
 assumes *sig-red-zero* ($\preceq_t$) (*set bs*) *r* **and** *component-of-term* (*lt r*) < *j*
 shows *sig-red-zero* ($\preceq_t$) (*set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*)) *r*
⟨*proof*⟩

lemma *rb-aux-inv2-preserved-0*:
 assumes *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **and** *j* < *length fs* **and** *Inr j* ∉ *set ps*
  and *b* ∈ *set bs* **and** *component-of-term* (*lt b*) < *j*
 shows ∃ *s*∈*set* (*new-syz-sigs ss bs p*). *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
⟨*proof*⟩

lemma *rb-aux-inv2-preserved-1*:
 assumes *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **and** *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
 shows *rb-aux-inv2* (*bs*, *new-syz-sigs ss bs p*, *ps*)
 ⟨*proof*⟩

lemma *rb-aux-inv2-preserved-3*:
 assumes *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **and** ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
  and *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
 shows *rb-aux-inv2* (*sig-trd bs* (*poly-of-pair p*) # *bs*, *new-syz-sigs ss bs p*,
   add-spairs ps bs (*sig-trd bs* (*poly-of-pair p*)))
⟨*proof*⟩

lemma *rb-aux-inv2-ideal-subset*:
 assumes *rb-aux-inv2* (*bs*, *ss*, *ps*) **and** ⋀*p0*. *p0* ∈ *set ps* ⟹ *j* ≤ *component-of-term* (*sig-of-pair p0*)
 shows *ideal* (*set* (*take j fs*)) ⊆ *ideal* (*rep-list* ‘ *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*))
   (**is** *ideal ?B* ⊆ *ideal ?A*)
⟨*proof*⟩

lemma *rb-aux-inv-is-Groebner-basis*:
 assumes *hom-grading dgrad* **and** *rb-aux-inv* (*bs*, *ss*, *ps*)
  and ⋀*p0*. *p0* ∈ *set ps* ⟹ *j* ≤ *component-of-term* (*sig-of-pair p0*)
 shows *punit.is-Groebner-basis* (*rep-list* ‘ *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*))
   (**is** *punit.is-Groebner-basis* (*rep-list* ‘ *set ?bs*))
 ⟨*proof*⟩

lemma *rb-aux-inv2-no-zero-red*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs* **and** *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*)
and ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows *rep-list* (*sig-trd bs* (*poly-of-pair p*)) $\neq$ *0*
⟨*proof*⟩

corollary *rb-aux-no-zero-red'*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs* **and** *rb-aux-inv2* (*fst args*)
shows *snd* (*rb-aux args*) = *snd args*
⟨*proof*⟩

corollary *rb-aux-no-zero-red*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs*
shows *snd* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)) = *z*
⟨*proof*⟩

corollary *rb-no-zero-red*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs*
shows *snd rb* = *0*
⟨*proof*⟩

end

4.3 Sig-Poly-Pairs

We now prove that the algorithms defined for sig-poly-pairs (i. e. those whose names end with *-spp*) behave exactly as those defined for module elements. More precisely, if A is some algorithm defined for module elements, we prove something like *spp-of* (A x) = A*-spp* (*spp-of* x).

fun *spp-inv-pair* :: ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) ⇒ *bool* **where**
spp-inv-pair (*Inl* (*p*, *q*)) = (*spp-inv p* ∧ *spp-inv q*) |
spp-inv-pair (*Inr j*) = *True*

fun *app-pair* :: ('*x* ⇒ '*y*) ⇒ (('*x* × '*x*) + *nat*) ⇒ (('*y* × '*y*) + *nat*) **where**
app-pair f (*Inl* (*p*, *q*)) = *Inl* (*f p*, *f q*) |
app-pair f (*Inr j*) = *Inr j*

fun *app-args* :: ('*x* ⇒ '*y*) ⇒ (('*x list* × '*z* × ((('*x* × '*x*) + *nat*) *list*)) × *nat*) ⇒
(('*y list* × '*z* × ((('*y* × '*y*) + *nat*) *list*)) × *nat*) **where**
app-args f ((*as*, *bs*, *cs*), *n*) = ((*map f as*, *bs*, *map* (*app-pair f*) *cs*), *n*)

lemma *app-pair-spp-of-vec-of*:
assumes *spp-inv-pair p*
shows *app-pair spp-of* (*app-pair vec-of p*) = *p*
⟨*proof*⟩

lemma *map-app-pair-spp-of-vec-of*:

assumes *list-all spp-inv-pair ps*
shows *map* (*app-pair spp-of* ∘ *app-pair vec-of*) *ps* = *ps*
⟨*proof*⟩

lemma *snd-app-args*: *snd* (*app-args f args*) = *snd args*
⟨*proof*⟩

lemma *new-syz-sigs-alt-spp*:
new-syz-sigs ss bs p = *new-syz-sigs-spp ss* (*map spp-of bs*) (*app-pair spp-of p*)
⟨*proof*⟩

lemma *is-rewritable-alt-spp*:
assumes $0 \notin$ *set bs*
shows *is-rewritable bs p u* = *is-rewritable-spp* (*map spp-of bs*) (*spp-of p*) *u*
⟨*proof*⟩

lemma *spair-sigs-alt-spp*: *spair-sigs p q* = *spair-sigs-spp* (*spp-of p*) (*spp-of q*)
⟨*proof*⟩

lemma *sig-crit-alt-spp*:
assumes $0 \notin$ *set bs*
shows *sig-crit bs ss p* = *sig-crit-spp* (*map spp-of bs*) *ss* (*app-pair spp-of p*)
⟨*proof*⟩

lemma *spair-alt-spp*:
assumes *is-regular-spair p q*
shows *spp-of* (*spair p q*) = *spair-spp* (*spp-of p*) (*spp-of q*)
⟨*proof*⟩

lemma *sig-trd-spp-body-alt-Some*:
assumes *find-sig-reducer* (*map spp-of bs*) *v* (*punit.lt p*) *0* = *Some i*
shows *sig-trd-spp-body* (*map spp-of bs*) *v* (*p*, *r*) =
(*punit.lower* (*p* − *local.punit.monom-mult* (*punit.lc p* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*punit.lt p* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*))) (*punit.lt p*), *r*)
(**is** *?thesis1*)
and *sig-trd-spp-body* (*map spp-of bs*) *v* (*p*, *r*) =
(*p* − *local.punit.monom-mult* (*punit.lc p* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*punit.lt p* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*)), *r*)
(**is** *?thesis2*)
⟨*proof*⟩

lemma *sig-trd-aux-alt-spp*:
assumes *fst args* ∈ *keys* (*rep-list* (*snd args*))
shows *rep-list* (*sig-trd-aux bs args*) =
sig-trd-spp-aux (*map spp-of bs*) (*lt* (*snd args*))
(*rep-list* (*snd args*) − *punit.higher* (*rep-list* (*snd args*)) (*fst args*),
punit.higher (*rep-list* (*snd args*)) (*fst args*))

⟨*proof*⟩

lemma *sig-trd-alt-spp*: *spp-of* (*sig-trd bs p*) = *sig-trd-spp* (*map spp-of bs*) (*spp-of p*)
⟨*proof*⟩

lemma *is-regular-spair-alt-spp*: *is-regular-spair p q* ⟷ *is-regular-spair-spp* (*spp-of p*) (*spp-of q*)
⟨*proof*⟩

lemma *sig-of-spair-alt-spp*: *sig-of-pair p* = *sig-of-pair-spp* (*app-pair spp-of p*)
⟨*proof*⟩

lemma *pair-ord-alt-spp*: *pair-ord x y* ⟷ *pair-ord-spp* (*app-pair spp-of x*) (*app-pair spp-of y*)
⟨*proof*⟩

lemma *new-spairs-alt-spp*:
map (*app-pair spp-of*) (*new-spairs bs p*) = *new-spairs-spp* (*map spp-of bs*) (*spp-of p*)
⟨*proof*⟩

lemma *add-spairs-alt-spp*:
assumes $\bigwedge x$. *x* ∈ *set bs* ⟹ *Inl* (*spp-of p*, *spp-of x*) ∉ *app-pair spp-of* ‘ *set ps*
shows *map* (*app-pair spp-of*) (*add-spairs ps bs p*) =
add-spairs-spp (*map* (*app-pair spp-of*) *ps*) (*map spp-of bs*) (*spp-of p*)
⟨*proof*⟩

lemma *rb-aux-invD-app-args*:
assumes *rb-aux-inv* (*fst* (*app-args vec-of* ((*bs*, *ss*, *ps*), *z*)))
shows *list-all spp-inv bs* **and** *list-all spp-inv-pair ps*
⟨*proof*⟩

lemma *app-args-spp-of-vec-of*:
assumes *rb-aux-inv* (*fst* (*app-args vec-of args*))
shows *app-args spp-of* (*app-args vec-of args*) = *args*
⟨*proof*⟩

lemma *poly-of-pair-alt-spp*:
assumes *distinct fs* **and** *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
shows *spp-of* (*poly-of-pair p*) = *spp-of-pair* (*app-pair spp-of p*)
⟨*proof*⟩

lemma *rb-aux-alt-spp*:
assumes *rb-aux-inv* (*fst args*)
shows *app-args spp-of* (*rb-aux args*) = *rb-spp-aux* (*app-args spp-of args*)
⟨*proof*⟩

corollary *rb-spp-aux-alt*:

rb-aux-inv (*fst* (*app-args vec-of args*)) $\Longrightarrow$
rb-spp-aux args = *app-args spp-of* (*rb-aux* (*app-args vec-of args*))
⟨*proof*⟩

corollary *rb-spp-aux*:
hom-grading dgrad $\Longrightarrow$
punit.is-Groebner-basis (*set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))))
(**is** - $\Longrightarrow$ *?thesis1*)
ideal (*set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))) = *ideal* (*set fs*)
(**is** *?thesis2*)
set (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) $\subseteq$ *punit-dgrad-max-set dgrad*
(**is** *?thesis3*)
0 $\notin$ *set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))
(**is** *?thesis4*)
hom-grading dgrad $\Longrightarrow$ *is-pot-ord* $\Longrightarrow$ *is-regular-sequence fs* $\Longrightarrow$
snd (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)) = *z*
(**is** - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ *?thesis5*)
rword-strict = *rw-rat-strict* $\Longrightarrow$ *p* $\in$ *set* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))) $\Longrightarrow$
q $\in$ *set* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))) $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$
punit.lt (*snd p*) *adds punit.lt* (*snd q*) $\Longrightarrow$ *punit.lt* (*snd p*) $\oplus$ *fst q* $\prec_t$ *punit.lt* (*snd q*) $\oplus$ *fst p*
⟨*proof*⟩

end

end

end

end

end

definition *gb-sig-z* ::
(('*t* × ('*a* $\Rightarrow_0$ '*b*)) ⇒ ('*t* × ('*a* $\Rightarrow_0$ '*b*)) ⇒ *bool*) ⇒ ('*a* $\Rightarrow_0$ '*b*) *list* ⇒ (('*t* × ('*a* $\Rightarrow_0$ '*b*::*field*)) *list* × *nat*)
where *gb-sig-z rword-strict fs0* =
(*let fs* = *rev* (*remdups* (*rev* (*removeAll 0 fs0*)));
res = *rb-spp-aux fs rword-strict* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *0*) *in*
(*fst* (*fst res*), *snd res*))

The second return value of *gb-sig-z* is the total number of zero-reductions.

definition *gb-sig* :: $(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('a \Rightarrow_0 'b)$ *list* $\Rightarrow$ $('a \Rightarrow_0 'b{::}field)$ *list*
where *gb-sig rword-strict fs0* = *map snd* (*fst* (*gb-sig-z rword-strict fs0*))

theorem
assumes $\bigwedge$*fs. is-strict-rewrite-ord fs rword-strict*
shows *gb-sig-isGB*: *punit.is-Groebner-basis* (*set* (*gb-sig rword-strict fs*)) (**is** *?thesis1*)
and *gb-sig-ideal*: *ideal* (*set* (*gb-sig rword-strict fs*)) = *ideal* (*set fs*) (**is** *?thesis2*)
and *dgrad-p-set-closed-gb-sig*:
dickson-grading d $\Longrightarrow$ *set fs* $\subseteq$ *punit.dgrad-p-set d m* $\Longrightarrow$ *set* (*gb-sig rword-strict fs*) $\subseteq$ *punit.dgrad-p-set d m*
(**is** - $\Longrightarrow$ - $\Longrightarrow$ *?thesis3*)
and *gb-sig-nonzero*: *0* $\notin$ *set* (*gb-sig rword-strict fs*) (**is** *?thesis4*)
and *gb-sig-no-zero-red*: *is-pot-ord* $\Longrightarrow$ *is-regular-sequence fs* $\Longrightarrow$ *snd* (*gb-sig-z rword-strict fs*) = *0*
⟨*proof*⟩

theorem *gb-sig-z-is-min-sig-GB*:
assumes *p* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict fs*)) **and** *q* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict fs*))
and $p \neq q$ **and** *punit.lt* (*snd p*) *adds punit.lt* (*snd q*)
shows *punit.lt* (*snd p*) $\oplus$ *fst q* $\prec_t$ *punit.lt* (*snd q*) $\oplus$ *fst p*
⟨*proof*⟩

Summarizing, these are the four main results proved in this theory:

- ($\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*) $\Longrightarrow$ *punit.is-Groebner-basis* (*set* (*gb-sig ?rword-strict ?fs*)),
- ($\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*) $\Longrightarrow$ *ideal* (*set* (*gb-sig ?rword-strict ?fs*)) = *ideal* (*set ?fs*),
- ⟦$\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*; *is-pot-ord*; *is-regular-sequence ?fs*⟧ $\Longrightarrow$ *snd* (*gb-sig-z ?rword-strict ?fs*) = *0*, and
- ⟦*?p* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict ?fs*)); *?q* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict ?fs*)); *?p* $\neq$ *?q*; *punit.lt* (*snd ?p*) *adds punit.lt* (*snd ?q*)⟧ $\Longrightarrow$ *punit.lt* (*snd ?p*) $\oplus$ *fst ?q* $\prec_t$ *punit.lt* (*snd ?q*) $\oplus$ *fst ?p*.

end

end

5 Sample Computations with Signature-Based Algorithms

theory *Signature-Examples*
imports *Signature-Groebner Groebner-Bases.Benchmarks Groebner-Bases.Code-Target-Rat*
begin

5.1 Setup

lift-definition *except-pp* :: $('a, 'b)$ *pp* $\Rightarrow$ *'a set* $\Rightarrow$ $('a, 'b::zero)$ *pp* **is** *except* ⟨*proof*⟩

lemma *hom-grading-varnum-pp*: *hom-grading* (*varnum-pp*::('*a*::*countable*, '*b*::*add-wellorder*) *pp* $\Rightarrow$ *nat*)
⟨*proof*⟩

instance *pp* :: (*countable*, *add-wellorder*) *quasi-pm-powerprod*
⟨*proof*⟩

5.1.1 Projections of Term Orders to Orders on Power-Products

definition *proj-comp* :: (('*a*::*nat*, '*b*::*nat*) *pp* $\times$ *nat*) *nat-term-order* $\Rightarrow$ ('*a*, '*b*) *pp* $\Rightarrow$ ('*a*, '*b*) *pp* $\Rightarrow$ *order*
where *proj-comp cmp* = ($\lambda x\ y.$ *nat-term-compare cmp* $(x,\ 0)$ $(y,\ 0)$)

definition *proj-ord* :: (('*a*::*nat*, '*b*::*nat*) *pp* $\times$ *nat*) *nat-term-order* $\Rightarrow$ ('*a*, '*b*) *pp* *nat-term-order*
where *proj-ord cmp* = *Abs-nat-term-order* (*proj-comp cmp*)

In principle, *proj-comp* and *proj-ord* could be defined more generally on type '*a* $\times$ *nat*, but then '*a* would have to belong to some new type-class which is the intersection of *nat-pp-term* and *nat-pp-compare* and additionally requires *rep-nat-term* x = (*rep-nat-pp* x, 0).

lemma *comparator-proj-comp*: *comparator* (*proj-comp cmp*)
⟨*proof*⟩

lemma *nat-term-comp-proj-comp*: *nat-term-comp* (*proj-comp cmp*)
⟨*proof*⟩

corollary *nat-term-compare-proj-ord*: *nat-term-compare* (*proj-ord cmp*) = *proj-comp cmp*
⟨*proof*⟩

lemma *proj-ord-LEX* [*code*]: *proj-ord LEX* = *LEX*
⟨*proof*⟩

lemma *proj-ord-DRLEX* [*code*]: *proj-ord DRLEX* = *DRLEX*
⟨*proof*⟩

lemma *proj-ord-DEG* [*code*]: *proj-ord* (*DEG to*) = *DEG* (*proj-ord to*)
⟨*proof*⟩

lemma *proj-ord-POT* [*code*]: *proj-ord* (*POT to*) = *proj-ord to*
⟨*proof*⟩

5.1.2 Locale Interpretation

locale *qpm-nat-inf-term* = *gd-nat-term* $\lambda x.\ x$ $\lambda x.\ x$ *to*

for *to*::((′*a*::*nat*, ′*b*::*nat*) *pp* × *nat*) *nat-term-order*
begin

sublocale *aux*: *qpm-inf-term* $\lambda x.\ x$ $\lambda x.\ x$
le-of-nat-term-order (*proj-ord to*)
lt-of-nat-term-order (*proj-ord to*)
le-of-nat-term-order to
lt-of-nat-term-order to
⟨*proof*⟩

end

We must define the following two constants outside the global interpretation, since otherwise their types are too general.

definition *splus-pprod* :: (′*a*::*nat*, ′*b*::*nat*) *pp* ⇒ -
where *splus-pprod* = *pprod.splus*

definition *adds-term-pprod* :: ((′*a*::*nat*, ′*b*::*nat*) *pp* × -) ⇒ -
where *adds-term-pprod* = *pprod.adds-term*

global-interpretation *pprod′*: *qpm-nat-inf-term to*
rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.adds-term* = *adds-term-pprod*
and *punit.monom-mult* = *monom-mult-punit*
and *pprod′.aux.punit.lt* = *lt-punit* (*proj-ord to*)
and *pprod′.aux.punit.lc* = *lc-punit* (*proj-ord to*)
and *pprod′.aux.punit.tail* = *tail-punit* (*proj-ord to*)
for *to* :: ((′*a*::*nat*, ′*b*::*nat*) *pp* × *nat*) *nat-term-order*
defines *max-pprod* = *pprod′.ord-term-lin.max*
and *Koszul-syz-sigs-aux-pprod* = *pprod′.aux.Koszul-syz-sigs-aux*
and *Koszul-syz-sigs-pprod* = *pprod′.aux.Koszul-syz-sigs*
and *find-sig-reducer-pprod* = *pprod′.aux.find-sig-reducer*
and *sig-trd-spp-body-pprod* = *pprod′.aux.sig-trd-spp-body*
and *sig-trd-spp-aux-pprod* = *pprod′.aux.sig-trd-spp-aux*
and *sig-trd-spp-pprod* = *pprod′.aux.sig-trd-spp*
and *spair-sigs-spp-pprod* = *pprod′.aux.spair-sigs-spp*
and *is-pred-syz-pprod* = *pprod′.aux.is-pred-syz*
and *is-rewritable-spp-pprod* = *pprod′.aux.is-rewritable-spp*
and *sig-crit-spp-pprod* = *pprod′.aux.sig-crit-spp*
and *spair-spp-pprod* = *pprod′.aux.spair-spp*
and *spp-of-pair-pprod* = *pprod′.aux.spp-of-pair*
and *pair-ord-spp-pprod* = *pprod′.aux.pair-ord-spp*
and *sig-of-pair-spp-pprod* = *pprod′.aux.sig-of-pair-spp*
and *new-spairs-spp-pprod* = *pprod′.aux.new-spairs-spp*
and *is-regular-spair-spp-pprod* = *pprod′.aux.is-regular-spair-spp*
and *add-spairs-spp-pprod* = *pprod′.aux.add-spairs-spp*
and *is-pot-ord-pprod* = *pprod′.is-pot-ord*

and *new-syz-sigs-spp-pprod* = *pprod'.aux.new-syz-sigs-spp*
and *rb-spp-body-pprod* = *pprod'.aux.rb-spp-body*
and *rb-spp-aux-pprod* = *pprod'.aux.rb-spp-aux*
and *gb-sig-z-pprod'* = *pprod'.aux.gb-sig-z*
and *gb-sig-pprod'* = *pprod'.aux.gb-sig*
and *rw-rat-strict-pprod* = *pprod'.aux.rw-rat-strict*
and *rw-add-strict-pprod* = *pprod'.aux.rw-add-strict*
⟨*proof*⟩

5.1.3 More Lemmas and Definitions

lemma *compute-adds-term-pprod* [*code*]:
adds-term-pprod u v = (*snd u* = *snd v* ∧ *adds-pp-add-linorder* (*fst u*) (*fst v*))
⟨*proof*⟩

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

lemma *compute-sig-trd-spp-body-pprod* [*code*]:
sig-trd-spp-body-pprod to bs v (*p*, *r*) =
(*case find-sig-reducer-pprod to bs v* (*lt-punit* (*proj-ord to*) *p*) *0 of*
None ⇒ (*tail-punit* (*proj-ord to*) *p*, *plus-monomial-less r* (*lc-punit* (*proj-ord to*) *p*) (*lt-punit* (*proj-ord to*) *p*))
| *Some i* ⇒ *let b* = *snd* (*bs* ! *i*) *in*
(*tail-punit* (*proj-ord to*) *p* − *monom-mult-punit* (*lc-punit* (*proj-ord to*) *p* / *lc-punit* (*proj-ord to*) *b*)
(*lt-punit* (*proj-ord to*) *p* − *lt-punit* (*proj-ord to*) *b*) (*tail-punit* (*proj-ord to*) *b*), *r*))
⟨*proof*⟩

lemma *compute-sig-trd-spp-pprod* [*code*]:
sig-trd-spp-pprod to bs (*v*, *p*) ≡ (*v*, *sig-trd-spp-aux-pprod to bs v* (*p*, *change-ord* (*proj-ord to*) *0*))
⟨*proof*⟩

lemmas [*code*] = *conversep-iff*

lemma *compute-is-pot-ord* [*code*]:
is-pot-ord-pprod (*LEX*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*) = *False*
(**is** *is-pot-ord-pprod ?lex* = -)
is-pot-ord-pprod (*DRLEX*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*) = *False*
(**is** *is-pot-ord-pprod ?drlex* = -)
is-pot-ord-pprod (*DEG* (*to*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*)) = *False*
is-pot-ord-pprod (*POT* (*to*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*)) = *True*
⟨*proof*⟩

corollary *is-pot-ord-POT*: *is-pot-ord-pprod* (*POT to*)
⟨*proof*⟩

definition *gb-sig-z-pprod to rword-strict fs* $\equiv$
(*let res* = *gb-sig-z-pprod′ to* (*rword-strict to*) (*map* (*change-ord* (*proj-ord to*)) *fs*) *in*
(*length* (*fst res*), *snd res*))

definition *gb-sig-pprod to rword-strict fs* $\equiv$ *gb-sig-pprod′ to* (*rword-strict to*) (*map* (*change-ord* (*proj-ord to*)) *fs*)

lemma *snd-gb-sig-z-pprod′-eq-gb-sig-z-pprod*:
snd (*gb-sig-z-pprod′ to* (*rword-strict to*) *fs*) = *snd* (*gb-sig-z-pprod to rword-strict fs*)
⟨*proof*⟩

lemma *gb-sig-pprod′-eq-gb-sig-pprod*:
gb-sig-pprod′ to (*rword-strict to*) *fs* = *gb-sig-pprod to rword-strict fs*
⟨*proof*⟩

thm *pprod′.aux.gb-sig-isGB*[*OF pprod′.aux.rw-rat-strict-is-strict-rewrite-ord*, *simplified gb-sig-pprod′-eq-gb-sig-pprod*]
thm *pprod′.aux.gb-sig-no-zero-red*[*OF pprod′.aux.rw-rat-strict-is-strict-rewrite-ord is-pot-ord-POT*, *simplified snd-gb-sig-z-pprod′-eq-gb-sig-z-pprod*]

5.2 Computations

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

lemma
gb-sig-pprod DRLEX rw-rat-strict-pprod [$X^2 * Z$ ^ *3* + *3* $* X^2 * Y$, $X * Y * Z$ + *2* $* Y^2$] =
[C_0 (*3* / *4*) $* X$ ^ *3* $* Y^2$ − *2* $* Y$ ^ *4*, − *4* $* Y$ ^ *3* $* Z$ − *3* $* X^2 * Y^2$, $X * Y * Z$ + *2* $* Y^2$, $X^2 * Z$ ^ *3* + *3* $* X^2 * Y$]
⟨*proof*⟩

end

Recall that the first return value of *gb-sig-z-pprod* is the size of the computed Gröbner basis, and the second return value is the total number of useless zero-reductions:

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*cyclic DRLEX 6*)::(- $\Rightarrow_0$ *rat*) *list*) = (*155*, *8*)
⟨*proof*⟩

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*katsura DRLEX 5*)::(- $\Rightarrow_0$ *rat*) *list*) = (*29*, *0*)
⟨*proof*⟩

lemma

gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*eco DRLEX 8*)::(- $\Rightarrow_0$ *rat*) *list*) = (*76*, *0*)
⟨*proof*⟩

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*noon DRLEX 5*)::(- $\Rightarrow_0$ *rat*) *list*) = (*83*, *0*)
⟨*proof*⟩

end

References

[1] C. Eder and J.-C. Faugère. A Survey on Signature-Based Algorithms for Computing Gröbner Bases. *J. Symb. Comput.*, 80(3):719–784, 2017.

[2] C. Eder and B. H. Roune. Signature Rewriting in Gröbner Basis Computation. In *Proceedings of ISSAC'13*, pages 331–338. ACM, 2013.

[3] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases without Reduction to Zero (F_5). In T. Mora, editor, *Proceedings of ISSAC'02*, pages 61–88. ACM, 2002.

[4] F. Immler and A. Maletzky. Gröbner Bases Theory. *Archive of Formal Proofs*, 2016. http://isa-afp.org/entries/Groebner_Bases.html, Formal proof development.

[5] B. H. Roune and M. Stillman. Practical Gröbner Basis Computation. In *Proceedings of ISSAC'12*, pages 203–210. ACM, 2012.