

Gröbner Bases Theory

Fabian Immler and Alexander Maletzky*

June 10, 2026

Abstract

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

Contents

1	Introduction	6
1.1	Related Work	6
1.2	Future Work	7
2	General Utilities	7
2.1	Lists	7
2.1.1	<i>max-list</i>	8
2.1.2	<i>insort-wrt</i>	9
2.1.3	<i>diff-list</i> and <i>insert-list</i>	9
2.1.4	<i>remdups-wrt</i>	9
2.1.5	<i>map-idx</i>	10
2.1.6	<i>map-dup</i>	11
2.1.7	Filtering Minimal Elements	11
3	Properties of Binary Relations	13
3.1	<i>Restricted-Predicates.wfp-on</i>	14
3.2	Relations	14
3.3	Setup for Connection to Theory <i>Abstract–Rewriting.Abstract-Rewriting</i>	15

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

3.4	Simple Lemmas	16
3.5	Advanced Results and the Generalized Newman Lemma	18
4	Polynomial Reduction	20
4.1	Basic Properties of Reduction	21
4.2	Reducibility and Addition & Multiplication	25
4.3	Confluence of Reducibility	26
4.4	Reducibility and Module Membership	27
4.5	More Properties of <i>red</i> , <i>red-single</i> and <i>is-red</i>	27
4.6	Well-foundedness and Termination	31
4.7	Algorithms	33
4.7.1	Function <i>find-adds</i>	33
4.7.2	Function <i>trd</i>	34
5	Gröbner Bases and Buchberger's Theorem	35
5.1	Critical Pairs and S-Polynomials	36
5.2	Buchberger's Theorem	38
5.3	Buchberger's Criteria for Avoiding Useless Pairs	39
5.4	Weak and Strong Gröbner Bases	40
5.5	Alternative Characterization of Gröbner Bases via Representations of S-Polynomials	43
5.6	Replacing Elements in Gröbner Bases	43
5.7	An Inconstructive Proof of the Existence of Finite Gröbner Bases	45
5.8	Relation <i>red-supset</i>	46
5.9	Context <i>od-term</i>	47
6	A General Algorithm Schema for Computing Gröbner Bases	47
6.1	<i>processed</i>	48
6.2	Algorithm Schema	49
6.2.1	<i>const-lt-component</i>	49
6.2.2	Type synonyms	50
6.2.3	Specification of the <i>selector</i> parameter	51
6.2.4	Specification of the <i>add-basis</i> parameter	51
6.2.5	Specification of the <i>add-pairs</i> parameter	52
6.2.6	Function <i>args-to-set</i>	55
6.2.7	Functions <i>count-const-lt-components</i> , <i>count-rem-comps</i> and <i>full-gb</i>	56
6.2.8	Specification of the <i>completion</i> parameter	57
6.2.9	Function <i>gb-schema-dummy</i>	60
6.2.10	Function <i>gb-schema-aux</i>	68
6.2.11	Functions <i>gb-schema-direct</i> and <i>term gb-schema-incr</i>	71
6.3	Suitable Instances of the <i>add-pairs</i> Parameter	73
6.3.1	Specification of the <i>crit</i> parameters	73

6.3.2	Suitable instances of the <i>crit</i> parameters	76
6.3.3	Creating Initial List of New Pairs	78
6.3.4	Applying Criteria to New Pairs	81
6.3.5	Applying Criteria to Old Pairs	83
6.3.6	Creating Final List of Pairs	84
6.4	Suitable Instances of the <i>completion</i> Parameter	85
6.5	Suitable Instances of the <i>add-basis</i> Parameter	87
6.6	Special Case: Scalar Polynomials	88
7	Buchberger's Algorithm	88
7.1	Reduction	89
7.2	Pair Selection	90
7.3	Buchberger's Algorithm	90
7.3.1	Special Case: <i>punit</i>	91
8	Benchmark Problems for Computing Gröbner Bases	92
8.1	Cyclic	92
8.2	Katsura	92
8.3	Eco	93
8.4	Noon	93
9	Code Equations Related to the Computation of Gröbner Bases	93
10	Sample Computations with Buchberger's Algorithm	95
10.1	Scalar Polynomials	95
10.2	Vector Polynomials	98
11	Further Properties of Multivariate Polynomials	101
11.1	Modules and Linear Hulls	102
11.2	Ordered Polynomials	102
11.2.1	Sets of Leading Terms and -Coefficients	102
11.2.2	Monicity	103
12	Auto-reducing Lists of Polynomials	104
12.1	Reduction and Monic Sets	105
12.2	Minimal Bases and Auto-reduced Bases	105
12.3	Computing Minimal Bases	107
12.4	Auto-Reduction	107
12.5	Auto-Reduction and Monicity	109
13	Reduced Gröbner Bases	110
13.1	Definition and Uniqueness of Reduced Gröbner Bases	110
13.2	Computing Reduced Gröbner Bases by Auto-Reduction	111
13.2.1	Minimal Bases	111

13.2.2	Computing Minimal Bases	111
13.2.3	Computing Reduced Bases	112
13.2.4	Computing Reduced Gröbner Bases	112
13.2.5	Properties of the Reduced Gröbner Basis of an Ideal	115
13.2.6	Context <i>od-term</i>	115
14	Sample Computations of Reduced Gröbner Bases	116
15	Macaulay Matrices	118
15.1	More about Vectors	119
15.2	More about Matrices	120
15.2.1	<i>nzrows</i>	120
15.2.2	<i>row-space</i>	120
15.2.3	<i>row-echelon</i>	121
15.3	Converting Between Polynomials and Macaulay Matrices	122
15.4	Properties of Macaulay Matrices	125
15.5	Functions <i>Macaulay-mat</i> and <i>Macaulay-list</i>	127
16	Faugère’s F4 Algorithm	128
16.1	Symbolic Preprocessing	128
16.2	<i>lin-red</i>	133
16.3	Reduction	134
16.4	Pair Selection	137
16.5	The F4 Algorithm	137
16.5.1	Special Case: <i>punit</i>	138
17	Sample Computations with the F4 Algorithm	138
17.1	Preparations	139
17.2	Computations	140
18	Syzygies of Multivariate Polynomials	142
18.1	Syzygy Modules Generated by Sets	142
18.2	Polynomial Mappings on List-Indices	145
18.3	POT Orders	147
18.4	Gröbner Bases of Syzygy Modules	148
18.4.1	<i>lift-poly-syz</i>	149
18.4.2	<i>proj-poly-syz</i>	150
18.4.3	<i>cofactor-list-syz</i>	151
18.4.4	<i>init-syzygy-list</i>	152
18.4.5	<i>proj-orig-basis</i>	152
18.4.6	<i>filter-syzygy-basis</i>	153
18.4.7	<i>syzygy-module-list</i>	153
18.4.8	Cofactors	154
18.4.9	Modules	155

18.4.10 Gröbner Bases	155
19 Sample Computations of Syzygies	156
19.1 Preparations	156
19.2 Computations	159
19.3 Univariate Polynomials	161
19.4 Homogeneity	162

1 Introduction

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

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

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

1.1 Related Work

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

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

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

1.2 Future Work

This formalization can be extended in several ways:

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

2 General Utilities

```
theory General
  imports Polynomials.Utills
begin
```

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

2.1 Lists

```
lemma distinct-reorder: distinct (xs @ (y # ys)) = distinct (y # (xs @ ys)) <proof>
```

```
lemma set-reorder: set (xs @ (y # ys)) = set (y # (xs @ ys)) <proof>
```

```
lemma distinctI:
```

```
  assumes  $\bigwedge i j. i < j \implies i < \text{length } xs \implies j < \text{length } xs \implies xs ! i \neq xs ! j$ 
  shows distinct xs
  <proof>
```

```
lemma filter-nth-pairE:
```

```
  assumes  $i < j$  and  $i < \text{length } (\text{filter } P \text{ } xs)$  and  $j < \text{length } (\text{filter } P \text{ } xs)$ 
  obtains  $i' j'$  where  $i' < j'$  and  $i' < \text{length } xs$  and  $j' < \text{length } xs$ 
    and  $(\text{filter } P \text{ } xs) ! i = xs ! i'$  and  $(\text{filter } P \text{ } xs) ! j = xs ! j'$ 
  <proof>
```

```
lemma distinct-filterI:
```

```
  assumes  $\bigwedge i j. i < j \implies i < \text{length } xs \implies j < \text{length } xs \implies P (xs ! i) \implies P (xs ! j) \implies xs ! i \neq xs ! j$ 
  shows distinct (filter P xs)
  <proof>
```

lemma *set- zip -map*: $\text{set} (\text{zip} (\text{map } f \text{ } xs) (\text{map } g \text{ } xs)) = (\lambda x. (f \text{ } x, g \text{ } x)) \text{ ` } (\text{set } xs)$
 <proof>

lemma *set- zip -map1*: $\text{set} (\text{zip} (\text{map } f \text{ } xs) xs) = (\lambda x. (f \text{ } x, x)) \text{ ` } (\text{set } xs)$
 <proof>

lemma *set- zip -map2*: $\text{set} (\text{zip } xs (\text{map } f \text{ } xs)) = (\lambda x. (x, f \text{ } x)) \text{ ` } (\text{set } xs)$
 <proof>

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

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

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

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

2.1.1 *max-list*

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

context *linorder*
begin

lemma *max-list-Max*: $xs \neq [] \implies \text{max-list } xs = \text{Max } (\text{set } xs)$
 <proof>

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

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

end

2.1.2 *insort-wrt*

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

lemma *insort-wrt-not-Nil* [simp]: *insort-wrt* r x xs ≠ []
 ⟨proof⟩

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

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

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

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

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

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

notation *minus-list-set* (**infixl** $\langle - - \rangle$ 65)

declare *set-minus-list-set*[simp]

definition *insert-list* :: 'a ⇒ 'a list ⇒ 'a list

where *insert-list* x xs = (if x ∈ set xs then xs else x # xs)

lemma *set-insert-list*: set (*insert-list* x xs) = insert x (set xs)
 ⟨proof⟩

2.1.4 *remdups-wrt*

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

lemma *set-remdups-wrt*: $f \text{ ' set (remdups-wrt f xs) = f \text{ ' set xs}$
<proof>

lemma *subset-remdups-wrt*: $\text{set (remdups-wrt f xs) } \subseteq \text{set xs}$
<proof>

lemma *remdups-wrt-distinct-wrt*:
assumes $x \in \text{set (remdups-wrt f xs)}$ **and** $y \in \text{set (remdups-wrt f xs)}$ **and** $x \neq y$
shows $f x \neq f y$
<proof>

lemma *distinct-remdups-wrt*: $\text{distinct (remdups-wrt f xs)}$
<proof>

lemma *map-remdups-wrt*: $\text{map f (remdups-wrt f xs) = remdups (map f xs)}$
<proof>

lemma *remdups-wrt-append*:
 $\text{remdups-wrt f (xs @ ys) = (filter (\lambda a. f a \notin f \text{ ' set ys) (remdups-wrt f xs)) @}$
 $\text{(remdups-wrt f ys)}$
<proof>

2.1.5 map-idx

primrec *map-idx* :: $('a \Rightarrow \text{nat} \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'b \text{ list}$ **where**
 $\text{map-idx f [] n = []}$
 $\text{map-idx f (x \# xs) n = (f x n) \# (map-idx f xs (Suc n))}$

lemma *map-idx-eq-map2*: $\text{map-idx f xs n = map2 f xs [n..<n + length xs]}$
<proof>

lemma *length-map-idx [simp]*: $\text{length (map-idx f xs n) = length xs}$
<proof>

lemma *map-idx-append*: $\text{map-idx f (xs @ ys) n = (map-idx f xs n) @ (map-idx f}$
 $\text{ys (n + length xs))}$
<proof>

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

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

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

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

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

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

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

2.1.6 *map-dup*

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

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

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

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

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

2.1.7 Filtering Minimal Elements

context
fixes $rel :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
begin

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

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

definition *filter-min-append* :: 'a list \Rightarrow 'a list \Rightarrow 'a list
where *filter-min-append* xs ys =
 (let P = (λ zs. λ x. \neg (\exists z \in set zs. rel z x)); ys1 = *filter* (P xs) ys in
 (*filter* (P ys1) xs) @ ys1)

lemma *filter-min-aux-supset*: set ys \subseteq set (*filter-min-aux* xs ys)
 <proof>

lemma *filter-min-aux-subset*: set (*filter-min-aux* xs ys) \subseteq set xs \cup set ys
 <proof>

lemma *filter-min-aux-relE*:
assumes *transp* rel **and** x \in set xs **and** x \notin set (*filter-min-aux* xs ys)
obtains y **where** y \in set (*filter-min-aux* xs ys) **and** rel y x
 <proof>

lemma *filter-min-aux-minimal*:
assumes *transp* rel **and** x \in set (*filter-min-aux* xs ys) **and** y \in set (*filter-min-aux* xs ys)
and rel x y
assumes \bigwedge a b. a \in set xs \cup set ys \implies b \in set ys \implies rel a b \implies a = b
shows x = y
 <proof>

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

lemma *filter-min-subset*: set (*filter-min* xs) \subseteq set xs
 <proof>

lemma *filter-min-cases*:
assumes *transp* rel **and** x \in set xs
assumes x \in set (*filter-min* xs) \implies *thesis*
assumes \bigwedge y. y \in set (*filter-min* xs) \implies x \notin set (*filter-min* xs) \implies rel y x \implies
thesis
shows *thesis*
 <proof>

corollary *filter-min-relE*:
assumes *transp* rel **and** *reflp* rel **and** x \in set xs
obtains y **where** y \in set (*filter-min* xs) **and** rel y x
 <proof>

lemma *filter-min-minimal*:

assumes *transp rel* **and** $x \in \text{set } (\text{filter-min } xs)$ **and** $y \in \text{set } (\text{filter-min } xs)$ **and**
rel x y
shows $x = y$
<proof>

lemma *filter-min-distinct*:
assumes *reflp rel*
shows *distinct (filter-min xs)*
<proof>

lemma *filter-min-append-subset*: $\text{set } (\text{filter-min-append } xs \ ys) \subseteq \text{set } xs \cup \text{set } ys$
<proof>

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

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

lemma *filter-min-append-minimal*:
assumes $\bigwedge x' \ y'. x' \in \text{set } xs \implies y' \in \text{set } xs \implies \textit{rel } x' \ y' \implies x' = y'$
and $\bigwedge x' \ y'. x' \in \text{set } ys \implies y' \in \text{set } ys \implies \textit{rel } x' \ y' \implies x' = y'$
and $x \in \text{set } (\text{filter-min-append } xs \ ys)$ **and** $y \in \text{set } (\text{filter-min-append } xs \ ys)$
and *rel x y*
shows $x = y$
<proof>

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

end

end

3 Properties of Binary Relations

theory *Confluence*
imports *Abstract-Rewriting.Abstract-Rewriting Open-Induction.Restricted-Predicates*
begin

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

3.1 *Restricted-Predicates.wfp-on*

lemma *wfp-on-imp-wfP*:

assumes *wfp-on r A*

shows *wfP* ($\lambda x y. r x y \wedge x \in A \wedge y \in A$) (**is** *wfP ?r*)

<proof>

lemma *wfp-onI-min*:

assumes $\bigwedge x Q. x \in Q \implies Q \subseteq A \implies \exists z \in Q. \forall y \in A. r y z \longrightarrow y \notin Q$

shows *wfp-on r A*

<proof>

lemma *wfp-onE-min*:

assumes *wfp-on r A* **and** $x \in Q$ **and** $Q \subseteq A$

obtains *z* **where** $z \in Q$ **and** $\bigwedge y. r y z \implies y \notin Q$

<proof>

lemma *wfp-onI-chain*: $\neg (\exists f. \forall i. f i \in A \wedge r (f (Suc i)) (f i)) \implies wfp-on r A$

<proof>

lemma *finite-minimalE*:

assumes *finite A* **and** $A \neq \{\}$ **and** *irreflp rel* **and** *transp rel*

obtains *a* **where** $a \in A$ **and** $\bigwedge b. rel b a \implies b \notin A$

<proof>

lemma *wfp-on-finite*:

assumes *irreflp rel* **and** *transp rel* **and** *finite A*

shows *wfp-on rel A*

<proof>

3.2 Relations

locale *relation* = **fixes** $r::'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\langle \rightarrow \rangle$ 50)

begin

abbreviation $rtc::'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\langle \rightarrow^* \rangle$ 50)

where $rtc a b \equiv r^{**} a b$

abbreviation $sc::'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\langle \leftrightarrow \rangle$ 50)

where $sc a b \equiv a \rightarrow b \vee b \rightarrow a$

definition $is-final::'a \Rightarrow bool$ **where**

$is-final a \equiv \neg (\exists b. r a b)$

definition $srtc::'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\langle \leftrightarrow^* \rangle$ 50) **where**

$srtc a b \equiv sc^{**} a b$

definition $cs::'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\langle \downarrow^* \rangle$ 50) **where**
 $cs\ a\ b \equiv (\exists s. (a \rightarrow^* s) \wedge (b \rightarrow^* s))$

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

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

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

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

definition $dw-closed :: 'a\ set \Rightarrow bool$
where $dw-closed\ A \longleftrightarrow (\forall a \in A. \forall b. a \rightarrow b \longrightarrow b \in A)$

lemma $dw-closedI$ [*intro*]:
assumes $\bigwedge a\ b. a \in A \Longrightarrow a \rightarrow b \Longrightarrow b \in A$
shows $dw-closed\ A$
 $\langle proof \rangle$

lemma $dw-closedD$:
assumes $dw-closed\ A$ **and** $a \in A$ **and** $a \rightarrow b$
shows $b \in A$
 $\langle proof \rangle$

lemma $dw-closed-rtrancl$:
assumes $dw-closed\ A$ **and** $a \in A$ **and** $a \rightarrow^* b$
shows $b \in A$
 $\langle proof \rangle$

lemma $dw-closed-empty$: $dw-closed\ \{\}$
 $\langle proof \rangle$

lemma $dw-closed-UNIV$: $dw-closed\ UNIV$
 $\langle proof \rangle$

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

abbreviation (*input*) $relset::('a * 'a)\ set$ **where**
 $relset \equiv \{(x, y). x \rightarrow y\}$

lemma $rtc-rtranclI$:
assumes $a \rightarrow^* b$
shows $(a, b) \in relset^*$
 $\langle proof \rangle$

lemma *final-NF*: $(is\text{-}final\ a) = (a \in NF\ relset)$
<proof>

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

lemma *srtc-conversion*: $(a \leftrightarrow^* b) = ((a, b) \in relset^{\leftrightarrow*})$
<proof>

lemma *cs-join*: $(a \downarrow^* b) = ((a, b) \in relset^{\downarrow})$
<proof>

lemma *confluent-CR*: $is\text{-}confluent = CR\ relset$
<proof>

lemma *ChurchRosser-conversion*: $is\text{-}ChurchRosser = (relset^{\leftrightarrow*} \subseteq relset^{\downarrow})$
<proof>

lemma *loc-confluent-WCR*:
shows $is\text{-}loc\text{-}confluent = WCR\ relset$
<proof>

lemma *wf-converse*:
shows $(wfP\ r^{\hat{}}\text{-}1) = (wf\ (relset^{-1}))$
<proof>

lemma *wf-SN*:
shows $(wfP\ r^{\hat{}}\text{-}1) = (SN\ relset)$
<proof>

3.4 Simple Lemmas

lemma *rtrancl-is-final*:
assumes $a \rightarrow^* b$ **and** $is\text{-}final\ a$
shows $a = b$
<proof>

lemma *cs-refl*:
shows $x \downarrow^* x$
<proof>

lemma *cs-sym*:
assumes $x \downarrow^* y$
shows $y \downarrow^* x$
<proof>

lemma *rtc-implies-cs*:
assumes $x \rightarrow^* y$

shows $x \downarrow^* y$
<proof>

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

lemma *srtc-symmetric*:
assumes $a \leftrightarrow^* b$
shows $b \leftrightarrow^* a$
<proof>

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

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

lemma *confluence-equiv-ChurchRosser*: $is-confluent = is-ChurchRosser$
<proof>

corollary *confluence-implies-ChurchRosser*:
assumes $is-confluent$
shows $is-ChurchRosser$
<proof>

lemma *ChurchRosser-unique-final*:
assumes $is-ChurchRosser$ **and** $a \rightarrow^* b1$ **and** $a \rightarrow^* b2$ **and** $is-final\ b1$ **and**
 $is-final\ b2$
shows $b1 = b2$
<proof>

lemma *wf-on-imp-nf-ex*:
assumes $wfp-on\ ((\rightarrow)^{-1-1})\ A$ **and** $dw-closed\ A$ **and** $a \in A$
obtains b **where** $a \rightarrow^* b$ **and** $is-final\ b$
<proof>

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

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

corollary *unique-nf-imp-confluence*:
assumes $\bigwedge a b1 b2. (a \rightarrow^* b1) \implies (a \rightarrow^* b2) \implies is-final\ b1 \implies is-final\ b2$
 $\implies b1 = b2$
and $wfP ((\rightarrow)^{-1-1})$
shows *is-confluent*
 $\langle proof \rangle$

end

3.5 Advanced Results and the Generalized Newman Lemma

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

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

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

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

definition *is-loc-connective-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *is-loc-connective-on* $A\ ord\ r \iff (\forall a \in A. \forall b1\ b2. r\ a\ b1 \wedge r\ a\ b2 \longrightarrow cbelow-on\ A\ ord\ a\ (relation.sc\ r)\ b1\ b2)$

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

lemma *cbelow-on-first-below*:
assumes *cbelow-on* $A\ ord\ z\ rel\ a\ b$
shows *ord* $a\ z$
 $\langle proof \rangle$

lemma *cbelow-on-second-below*:
assumes *cbelow-on* $A\ ord\ z\ rel\ a\ b$
shows *ord* $b\ z$
 $\langle proof \rangle$

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

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

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

lemma *cbelow-on-induct* [*consumes 1, case-names base step*]:
assumes *a: cbelow-on A ord z rel a b*
and *base: $a \in A \implies ord a z \implies P a$*
and *ind: $\bigwedge b c. [\![\ cbelow-on A ord z rel a b; rel b c; c \in A; ord c z; P b \!\!] \implies$*
 $P c$
shows $P b$
 $\langle proof \rangle$

lemma *cbelow-on-symmetric*:
assumes *main: cbelow-on A ord z rel a b and symp rel*
shows *cbelow-on A ord z rel b a*
 $\langle proof \rangle$

lemma *cbelow-on-transitive*:
assumes *cbelow-on A ord z rel a b and cbelow-on A ord z rel b c*
shows *cbelow-on A ord z rel a c*
 $\langle proof \rangle$

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

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

lemma *relation-refines*:
assumes $a \rightarrow b$

shows $ord\ b\ a$
 $\langle proof \rangle$

lemma *relation-wf*: $wfp\text{-on}\ (\rightarrow)^{-1-1}\ A$
 $\langle proof \rangle$

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

lemma *cs-implies-cbelow-on*:
assumes $dw\text{-closed}\ A$ **and** $a \downarrow^* b$ **and** $a \in A$ **and** $b \in A$ **and** $ord\ a\ c$ **and** $ord\ b\ c$
shows $cbelow\text{-on}\ A\ ord\ c\ (\leftrightarrow)\ a\ b$
 $\langle proof \rangle$

The generalized Newman lemma, taken from [17]:

lemma *loc-connectivity-implies-confluence*:
assumes $is\text{-loc}\text{-connective}\text{-on}\ A\ ord\ (\rightarrow)$ **and** $dw\text{-closed}\ A$
shows $is\text{-confluent}\text{-on}\ A$
 $\langle proof \rangle$

end

theorem *loc-connectivity-equiv-ChurchRosser*:
assumes $relation\text{-order}\ r\ ord\ UNIV$
shows $relation.is\text{-ChurchRosser}\ r = is\text{-loc}\text{-connective}\text{-on}\ UNIV\ ord\ r$
 $\langle proof \rangle$

end

4 Polynomial Reduction

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

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

context *ordered-term*
begin

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

definition *red* :: $(t \Rightarrow_0 'b::field)\ set \Rightarrow (t \Rightarrow_0 'b) \Rightarrow (t \Rightarrow_0 'b) \Rightarrow bool$

where $\text{red } F \ p \ q \longleftrightarrow (\exists f \in F. \exists t. \text{red-single } p \ q \ f \ t)$

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

4.1 Basic Properties of Reduction

lemma red-setI :

assumes $f \in F$ **and** a : $\text{red-single } p \ q \ f \ t$

shows $\text{red } F \ p \ q$

$\langle \text{proof} \rangle$

lemma red-setE :

assumes $\text{red } F \ p \ q$

obtains f **and** t **where** $f \in F$ **and** $\text{red-single } p \ q \ f \ t$

$\langle \text{proof} \rangle$

lemma red-empty : $\neg \text{red } \{\} \ p \ q$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma red-union : $\text{red } (F \cup G) \ p \ q = (\text{red } F \ p \ q \vee \text{red } G \ p \ q)$

$\langle \text{proof} \rangle$

lemma red-unionI1 :

assumes $\text{red } F \ p \ q$

shows $\text{red } (F \cup G) \ p \ q$

$\langle \text{proof} \rangle$

lemma red-unionI2 :

assumes $\text{red } G \ p \ q$

shows $\text{red } (F \cup G) \ p \ q$

$\langle \text{proof} \rangle$

lemma red-subset :

assumes $\text{red } G \ p \ q$ **and** $G \subseteq F$

shows $\text{red } F \ p \ q$

$\langle \text{proof} \rangle$

lemma $\text{red-union-singleton-zero}$: $\text{red } (F \cup \{0\}) = \text{red } F$

$\langle \text{proof} \rangle$

lemma $\text{red-minus-singleton-zero}$: $\text{red } (F - \{0\}) = \text{red } F$

$\langle \text{proof} \rangle$

lemma $\text{red-rtrancl-subset}$:

assumes $\text{major}: (\text{red } G)^{**} \ p \ q$ **and** $G \subseteq F$

shows $(red\ F)^{**}\ p\ q$
 $\langle proof \rangle$

lemma *red-singleton*: $red\ \{f\}\ p\ q \longleftrightarrow (\exists\ t.\ red\ single\ p\ q\ f\ t)$
 $\langle proof \rangle$

lemma *red-single-lookup*:
assumes $red\ single\ p\ q\ f\ t$
shows $lookup\ q\ (t\ \oplus\ lt\ f) = 0$
 $\langle proof \rangle$

lemma *red-single-higher*:
assumes $red\ single\ p\ q\ f\ t$
shows $higher\ q\ (t\ \oplus\ lt\ f) = higher\ p\ (t\ \oplus\ lt\ f)$
 $\langle proof \rangle$

lemma *red-single-ord*:
assumes $red\ single\ p\ q\ f\ t$
shows $q\ \prec_p\ p$
 $\langle proof \rangle$

lemma *red-single-nonzero1*:
assumes $red\ single\ p\ q\ f\ t$
shows $p \neq 0$
 $\langle proof \rangle$

lemma *red-single-nonzero2*:
assumes $red\ single\ p\ q\ f\ t$
shows $f \neq 0$
 $\langle proof \rangle$

lemma *red-single-self*:
assumes $p \neq 0$
shows $red\ single\ p\ 0\ p\ 0$
 $\langle proof \rangle$

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

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

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

<proof>

lemma *red-ord*:

assumes $red\ F\ p\ q$

shows $q \prec_p p$

<proof>

lemma *red-indI1*:

assumes $f \in F$ **and** $f \neq 0$ **and** $p \neq 0$ **and** $adds: lt\ f\ adds_t\ lt\ p$

shows $red\ F\ p\ (p - monom-mult\ (lc\ p\ /\ lc\ f)\ (lp\ p - lp\ f)\ f)$

<proof>

lemma *red-indI2*:

assumes $p \neq 0$ **and** $r: red\ F\ (tail\ p)\ q$

shows $red\ F\ p\ (q + monomial\ (lc\ p)\ (lt\ p))$

<proof>

lemma *red-indE*:

assumes $red\ F\ p\ q$

shows $(\exists f \in F. f \neq 0 \wedge lt\ f\ adds_t\ lt\ p \wedge$

$(q = p - monom-mult\ (lc\ p\ /\ lc\ f)\ (lp\ p - lp\ f)\ f)) \vee$

$red\ F\ (tail\ p)\ (q - monomial\ (lc\ p)\ (lt\ p))$

<proof>

lemma *is-redI*:

assumes $red\ F\ a\ b$

shows $is-red\ F\ a$

<proof>

lemma *is-redE*:

assumes $is-red\ F\ a$

obtains b **where** $red\ F\ a\ b$

<proof>

lemma *is-red-alt*:

shows $is-red\ F\ a \iff (\exists b. red\ F\ a\ b)$

<proof>

lemma *is-red-singletonI*:

assumes $is-red\ F\ q$

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

<proof>

lemma *is-red-singletonD*:

assumes $is-red\ \{p\}\ q$ **and** $p \in F$

shows $is-red\ F\ q$

<proof>

lemma *is-red-singleton-trans*:

assumes *is-red* {*f*} *p* **and** *lt g adds_t lt f* **and** *g ≠ 0*
shows *is-red* {*g*} *p*
⟨*proof*⟩

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

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

lemma *is-red-indI1*:
assumes *f ∈ F* **and** *f ≠ 0* **and** *p ≠ 0* **and** *lt f adds_t lt p*
shows *is-red* *F* *p*
⟨*proof*⟩

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

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

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

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

lemma *components-red-subset*:
assumes *red* *F* *p* *q*
shows *component-of-term* ‘*keys* *q* \subseteq *component-of-term* ‘*keys* *p* \cup *component-of-term* ‘*Keys* *F*
⟨*proof*⟩

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

4.2 Reducibility and Addition & Multiplication

lemma *red-single-monom-mult*:

assumes *red-single* $p\ q\ f\ t$ **and** $c \neq 0$

shows *red-single* (*monom-mult* $c\ s\ p$) (*monom-mult* $c\ s\ q$) $f\ (s + t)$

<proof>

lemma *red-single-plus-1*:

assumes *red-single* $p\ q\ f\ t$ **and** $t \oplus lt\ f \notin keys\ (p + r)$

shows *red-single* $(q + r)\ (p + r)\ f\ t$

<proof>

lemma *red-single-plus-2*:

assumes *red-single* $p\ q\ f\ t$ **and** $t \oplus lt\ f \notin keys\ (q + r)$

shows *red-single* $(p + r)\ (q + r)\ f\ t$

<proof>

lemma *red-single-plus-3*:

assumes *red-single* $p\ q\ f\ t$ **and** $t \oplus lt\ f \in keys\ (p + r)$ **and** $t \oplus lt\ f \in keys\ (q + r)$

shows $\exists s. red_single\ (p + r)\ s\ f\ t \wedge red_single\ (q + r)\ s\ f\ t$

<proof>

lemma *red-single-plus*:

assumes *red-single* $p\ q\ f\ t$

shows *red-single* $(p + r)\ (q + r)\ f\ t \vee$

red-single $(q + r)\ (p + r)\ f\ t \vee$

$(\exists s. red_single\ (p + r)\ s\ f\ t \wedge red_single\ (q + r)\ s\ f\ t)$ **(is** $?A \vee ?B \vee ?C)$

<proof>

lemma *red-single-diff*:

assumes *red-single* $(p - q)\ r\ f\ t$

shows *red-single* $p\ (r + q)\ f\ t \vee red_single\ q\ (p - r)\ f\ t \vee$

$(\exists p'\ q'. red_single\ p\ p'\ f\ t \wedge red_single\ q\ q'\ f\ t \wedge r = p' - q')$ **(is** $?A \vee ?B$

$\vee ?C)$

<proof>

lemma *red-monom-mult*:

assumes $a: red\ F\ p\ q$ **and** $c \neq 0$

shows *red* $F\ (monom_mult\ c\ s\ p)\ (monom_mult\ c\ s\ q)$

<proof>

lemma *red-plus-keys-disjoint*:

assumes *red* $F\ p\ q$ **and** $keys\ p \cap keys\ r = \{\}$

shows *red* $F\ (p + r)\ (q + r)$

<proof>

lemma *red-plus*:

assumes *red* $F\ p\ q$

obtains s **where** $(red\ F)^{**}\ (p + r)\ s$ **and** $(red\ F)^{**}\ (q + r)\ s$

$\langle proof \rangle$

corollary *red-plus-cs*:

assumes $red\ F\ p\ q$

shows $relation.cs\ (red\ F)\ (p + r)\ (q + r)$

$\langle proof \rangle$

lemma *red-uminus*:

assumes $red\ F\ p\ q$

shows $red\ F\ (-p)\ (-q)$

$\langle proof \rangle$

lemma *red-diff*:

assumes $red\ F\ (p - q)\ r$

obtains $p'\ q'$ **where** $(red\ F)^{**}\ p\ p'$ **and** $(red\ F)^{**}\ q\ q'$ **and** $r = p' - q'$

$\langle proof \rangle$

lemma *red-diff-rtrancl'*:

assumes $(red\ F)^{**}\ (p - q)\ r$

obtains $p'\ q'$ **where** $(red\ F)^{**}\ p\ p'$ **and** $(red\ F)^{**}\ q\ q'$ **and** $r = p' - q'$

$\langle proof \rangle$

lemma *red-diff-rtrancl*:

assumes $(red\ F)^{**}\ (p - q)\ 0$

obtains s **where** $(red\ F)^{**}\ p\ s$ **and** $(red\ F)^{**}\ q\ s$

$\langle proof \rangle$

corollary *red-diff-rtrancl-cs*:

assumes $(red\ F)^{**}\ (p - q)\ 0$

shows $relation.cs\ (red\ F)\ p\ q$

$\langle proof \rangle$

4.3 Confluence of Reducibility

lemma *confluent-distinct-aux*:

assumes $r1: red-single\ p\ q1\ f1\ t1$ **and** $r2: red-single\ p\ q2\ f2\ t2$

and $t1 \oplus lt\ f1 \prec_t t2 \oplus lt\ f2$ **and** $f1 \in F$ **and** $f2 \in F$

obtains s **where** $(red\ F)^{**}\ q1\ s$ **and** $(red\ F)^{**}\ q2\ s$

$\langle proof \rangle$

lemma *confluent-distinct*:

assumes $r1: red-single\ p\ q1\ f1\ t1$ **and** $r2: red-single\ p\ q2\ f2\ t2$

and $ne: t1 \oplus lt\ f1 \neq t2 \oplus lt\ f2$ **and** $f1 \in F$ **and** $f2 \in F$

obtains s **where** $(red\ F)^{**}\ q1\ s$ **and** $(red\ F)^{**}\ q2\ s$

$\langle proof \rangle$

corollary *confluent-same*:

assumes $r1: red-single\ p\ q1\ f\ t1$ **and** $r2: red-single\ p\ q2\ f\ t2$ **and** $f \in F$

obtains s **where** $(red\ F)^{**}\ q1\ s$ **and** $(red\ F)^{**}\ q2\ s$

<proof>

4.4 Reducibility and Module Membership

lemma *srtc-in-pmdl*:

assumes *relation.srtc* (*red F*) *p q*

shows $p - q \in \text{pmdl } F$

<proof>

lemma *in-pmdl-srtc*:

assumes $p \in \text{pmdl } F$

shows *relation.srtc* (*red F*) *p 0*

<proof>

lemma *red-rtranclp-diff-in-pmdl*:

assumes $(\text{red } F)^{**} p q$

shows $p - q \in \text{pmdl } F$

<proof>

corollary *red-diff-in-pmdl*:

assumes *red F p q*

shows $p - q \in \text{pmdl } F$

<proof>

corollary *red-rtranclp-0-in-pmdl*:

assumes $(\text{red } F)^{**} p 0$

shows $p \in \text{pmdl } F$

<proof>

lemma *pmdl-closed-red*:

assumes $\text{pmdl } B \subseteq \text{pmdl } A$ **and** $p \in \text{pmdl } A$ **and** *red B p q*

shows $q \in \text{pmdl } A$

<proof>

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

lemma *red-rtrancl-mult*:

assumes $(\text{red } F)^{**} p q$

shows $(\text{red } F)^{**} (\text{monom-mult } c t p) (\text{monom-mult } c t q)$

<proof>

corollary *red-rtrancl-uminus*:

assumes $(\text{red } F)^{**} p q$

shows $(\text{red } F)^{**} (-p) (-q)$

<proof>

lemma *red-rtrancl-diff-induct* [*consumes 1, case-names base step*]:

assumes *a*: $(\text{red } F)^{**} (p - q) r$

and cases: $P p p !!y z. [| (\text{red } F)^{**} (p - q) z; \text{red } F z y; P p (q + z)|] ==> P p (q + y)$

shows $P p (q + r)$
<proof>

lemma *red-rtrancl-diff-0-induct* [*consumes 1, case-names base step*]:
assumes $a: (red\ F)^{**} (p - q)\ 0$
and base: $P\ p\ p$ **and** $ind: \bigwedge y\ z. [\![(red\ F)^{**} (p - q)\ y; red\ F\ y\ z; P\ p\ (y + q)\]\!]$
 $\implies P\ p\ (z + q)$
shows $P\ p\ q$
<proof>

lemma *is-red-union:* $is-red\ (A \cup B)\ p \longleftrightarrow (is-red\ A\ p \vee is-red\ B\ p)$
<proof>

lemma *red-single-0-lt:*
assumes *red-single* $f\ 0\ h\ t$
shows $lt\ f = t \oplus lt\ h$
<proof>

lemma *red-single-lt-distinct-lt:*
assumes $rs: red-single\ f\ g\ h\ t$ **and** $g \neq 0$ **and** $lt\ g \neq lt\ f$
shows $lt\ f = t \oplus lt\ h$
<proof>

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

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

lemma *is-red-addsI:*
assumes $f \in F$ **and** $f \neq 0$ **and** $v \in keys\ p$ **and** $lt\ f\ adds_t\ v$
shows $is-red\ F\ p$
<proof>

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

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

lemma *is-red-adds-iff:*

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

lemma *is-red-subset*:

assumes *red*: *is-red* $A \ p$ **and** *sub*: $A \subseteq B$

shows *is-red* $B \ p$

⟨proof⟩

lemma *not-is-red-empty*: $\neg \text{is-red } \{ \} \ f$

⟨proof⟩

lemma *red-single-mult-const*:

assumes *red-single* $p \ q \ f \ t$ **and** $c \neq 0$

shows *red-single* $p \ q \ (\text{monom-mult } c \ 0 \ f) \ t$

⟨proof⟩

lemma *red-rtrancl-plus-higher*:

assumes $(\text{red } F)^{**} \ p \ q$ **and** $\bigwedge u \ v. u \in \text{keys } p \implies v \in \text{keys } r \implies u \prec_t v$

shows $(\text{red } F)^{**} \ (p + r) \ (q + r)$

⟨proof⟩

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

$(- \ p \odot \text{tail } f)$

⟨proof⟩

corollary *red-mult-scalar-lt*:

assumes $f \neq 0$

shows $(\text{red } \{f\})^{**} \ (p \odot \text{monomial } c \ (lt \ f)) \ (\text{monom-mult } (- \ c \ / \ lc \ f) \ 0 \ (p \odot \text{tail } f))$

⟨proof⟩

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

⟨proof⟩

lemma *is-red-monomialI*:

assumes $c \neq 0$ **and** $f \in F$ **and** $f \neq 0$ **and** $\text{lt } f \text{ adds}_t v$

shows *is-red* $F \ (\text{monomial } c \ v)$

⟨proof⟩

lemma *is-red-monomialD*:

assumes *is-red* $F \ (\text{monomial } c \ v)$

shows $c \neq 0$

⟨proof⟩

lemma *is-red-monomialE*:

assumes *is-red* $F \ (\text{monomial } c \ v)$

obtains f **where** $f \in F$ **and** $f \neq 0$ **and** $\text{lt } f \text{ adds}_t v$

⟨proof⟩

lemma *replace-lt-adds-stable-is-red*:

assumes *red*: *is-red* F f **and** $q \neq 0$ **and** $lt\ q\ adds_t\ lt\ p$

shows *is-red* ($insert\ q\ (F - \{p\})$) f

<proof>

lemma *conversion-property*:

assumes *is-red* $\{p\}$ f **and** *red* $\{r\}$ $p\ q$

shows *is-red* $\{q\}$ $f \vee is-red\ \{r\}\ f$

<proof>

lemma *replace-red-stable-is-red*:

assumes $a1$: *is-red* F f **and** $a2$: *red* $(F - \{p\})$ $p\ q$

shows *is-red* ($insert\ q\ (F - \{p\})$) f (**is** *is-red* $?F'$ f)

<proof>

lemma *is-red-map-scale*:

assumes *is-red* F $(c \cdot p)$

shows *is-red* F p

<proof>

corollary *is-irred-map-scale*: $\neg is-red\ F\ p \implies \neg is-red\ F\ (c \cdot p)$

<proof>

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

<proof>

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

<proof>

lemma *is-red-plus*:

assumes *is-red* F $(p + q)$

shows *is-red* F $p \vee is-red\ F\ q$

<proof>

lemma *is-irred-plus*: $\neg is-red\ F\ p \implies \neg is-red\ F\ q \implies \neg is-red\ F\ (p + q)$

<proof>

lemma *is-red-minus*:

assumes *is-red* F $(p - q)$

shows *is-red* F $p \vee is-red\ F\ q$

<proof>

lemma *is-irred-minus*: $\neg is-red\ F\ p \implies \neg is-red\ F\ q \implies \neg is-red\ F\ (p - q)$

<proof>

end

4.6 Well-foundedness and Termination

context *gd-term*
begin

lemma *dgrad-set-le-red-single*:
assumes *dickson-grading d* **and** *red-single p q f t*
shows *dgrad-set-le d {t}* (*pp-of-term* ‘*keys p*)
<proof>

lemma *dgrad-p-set-le-red-single*:
assumes *dickson-grading d* **and** *red-single p q f t*
shows *dgrad-p-set-le d {q} {f, p}*
<proof>

lemma *dgrad-p-set-le-red*:
assumes *dickson-grading d* **and** *red F p q*
shows *dgrad-p-set-le d {q}* (*insert p F*)
<proof>

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

lemma *dgrad-p-set-red-single-pp*:
assumes *dickson-grading d* **and** $p \in dgrad-p-set\ d\ m$ **and** *red-single p q f t*
shows $d\ t \leq m$
<proof>

lemma *dgrad-p-set-closed-red-single*:
assumes *dickson-grading d* **and** $p \in dgrad-p-set\ d\ m$ **and** $f \in dgrad-p-set\ d\ m$
and *red-single p q f t*
shows $q \in dgrad-p-set\ d\ m$
<proof>

lemma *dgrad-p-set-closed-red*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$ **and** $p \in dgrad-p-set\ d\ m$
and *red F p q*
shows $q \in dgrad-p-set\ d\ m$
<proof>

lemma *dgrad-p-set-closed-red-rtrancl*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$ **and** $p \in dgrad-p-set\ d\ m$
and $(red\ F)^{**}\ p\ q$
shows $q \in dgrad-p-set\ d\ m$
<proof>

lemma *red-rtrancl-repE*:
assumes *dickson-grading d* **and** $G \subseteq dgrad-p-set\ d\ m$ **and** *finite G* **and** $p \in$

dgrad-p-set d m
and $(red\ G)^{**}\ p\ r$
obtains q **where** $p = r + (\sum_{g \in G}. q\ g \odot g)$ **and** $\bigwedge g. q\ g \in punit.dgrad-p-set\ d\ m$
and $\bigwedge g. lt\ (q\ g \odot g) \preceq_t\ lt\ p$
 $\langle proof \rangle$

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

lemma *red-wf-dgrad-p-set-aux*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$
shows *wfp-on* $(red\ F)^{-1-1}$ $(dgrad-p-set\ d\ m)$
 $\langle proof \rangle$

lemma *red-wf-dgrad-p-set*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$
shows *wfP* $(red\ F)^{-1-1}$
 $\langle proof \rangle$

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

lemma *cbelow-on-monom-mult*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$ **and** $d\ t \leq m$ **and** $c \neq 0$
and *cbelow-on* $(dgrad-p-set\ d\ m)$ (\prec_p) z $(\lambda a\ b. red\ F\ a\ b \vee red\ F\ b\ a)$ $p\ q$
shows *cbelow-on* $(dgrad-p-set\ d\ m)$ (\prec_p) $(monom-mult\ c\ t\ z)$ $(\lambda a\ b. red\ F\ a\ b \vee red\ F\ b\ a)$
 $(monom-mult\ c\ t\ p)$ $(monom-mult\ c\ t\ q)$
 $\langle proof \rangle$

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

lemma *cbelow-on-plus*:
assumes *dickson-grading d* **and** $F \subseteq dgrad-p-set\ d\ m$ **and** $r \in dgrad-p-set\ d\ m$
and *keys* $r \cap keys\ z = \{\}$
and *cbelow-on* $(dgrad-p-set\ d\ m)$ (\prec_p) z $(\lambda a\ b. red\ F\ a\ b \vee red\ F\ b\ a)$ $p\ q$
shows *cbelow-on* $(dgrad-p-set\ d\ m)$ (\prec_p) $(z + r)$ $(\lambda a\ b. red\ F\ a\ b \vee red\ F\ b\ a)$
 $(p + r)$ $(q + r)$
 $\langle proof \rangle$

lemma *is-full-pmdlI-lt-dgrad-p-set*:

assumes *dickson-grading* d **and** $B \subseteq \text{dgrad-p-set } d \ m$
assumes $\bigwedge k. k \in \text{component-of-term 'Keys (B::('t \Rightarrow_0 'b)::field) set} \implies$
 $(\exists b \in B. b \neq 0 \wedge \text{component-of-term (lt } b) = k \wedge \text{lp } b = 0)$
shows *is-full-pmdl* B
 $\langle \text{proof} \rangle$

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

end

4.7 Algorithms

4.7.1 Function *find-adds*

context *ordered-term*
begin

primrec *find-adds* :: $('t \Rightarrow_0 'b)$ *list* $\Rightarrow 't \Rightarrow ('t \Rightarrow_0 'b::\text{zero})$ **option** **where**
 $\text{find-adds } [] \text{ -} = \text{None}$
 $\text{find-adds } (f \# fs) \ u = (\text{if } f \neq 0 \wedge \text{lt } f \ \text{adds}_t \ u \ \text{then } \text{Some } f \ \text{else } \text{find-adds } fs \ u)$

lemma *find-adds-SomeD1*:
assumes $\text{find-adds } fs \ u = \text{Some } f$
shows $f \in \text{set } fs$
 $\langle \text{proof} \rangle$

lemma *find-adds-SomeD2*:
assumes $\text{find-adds } fs \ u = \text{Some } f$
shows $f \neq 0$
 $\langle \text{proof} \rangle$

lemma *find-adds-SomeD3*:
assumes $\text{find-adds } fs \ u = \text{Some } f$
shows $\text{lt } f \ \text{adds}_t \ u$
 $\langle \text{proof} \rangle$

lemma *find-adds-NoneE*:
assumes $\text{find-adds } fs \ u = \text{None}$ **and** $f \in \text{set } fs$
assumes $f = 0 \implies \text{thesis}$ **and** $f \neq 0 \implies \neg \text{lt } f \ \text{adds}_t \ u \implies \text{thesis}$
shows *thesis*
 $\langle \text{proof} \rangle$

lemma *find-adds-SomeD-red-single*:
assumes $p \neq 0$ **and** $\text{find-adds } fs \ (\text{lt } p) = \text{Some } f$
shows $\text{red-single } p \ (\text{tail } p - \text{monom-mult } (\text{lc } p / \text{lc } f) \ (\text{lp } p - \text{lp } f) \ (\text{tail } f)) \ f \ (\text{lp } p - \text{lp } f)$
 $\langle \text{proof} \rangle$

lemma *find-adds-SomeD-red*:

assumes $p \neq 0$ **and** $\text{find-adds fs (lt p) = Some f}$
shows $\text{red (set fs) p (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail f))}$
 $\langle \text{proof} \rangle$

end

4.7.2 Function *trd*

context *gd-term*

begin

definition $\text{trd-term} :: ('a \Rightarrow \text{nat}) \Rightarrow (((t \Rightarrow_0 'b::\text{field}) \text{list} \times (t \Rightarrow_0 'b) \times (t \Rightarrow_0 'b)) \times$

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

lemma *trd-term-wf*:

assumes *dickson-grading d*

shows $\text{wf (trd-term } d)$

$\langle \text{proof} \rangle$

function $\text{trd-aux} :: (t \Rightarrow_0 'b) \text{list} \Rightarrow (t \Rightarrow_0 'b) \Rightarrow (t \Rightarrow_0 'b) \Rightarrow (t \Rightarrow_0 'b::\text{field})$

where

trd-aux fs p r =

$(\text{if } p = 0 \text{ then}$

r

else

$\text{case find-adds fs (lt p) of}$

$\text{None} \Rightarrow \text{trd-aux fs (tail p) (r + monomial (lc p) (lt p))}$

$| \text{Some } f \Rightarrow \text{trd-aux fs (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail$

$f)) r$

$)$

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

definition $\text{trd} :: (t \Rightarrow_0 'b::\text{field}) \text{list} \Rightarrow (t \Rightarrow_0 'b) \Rightarrow (t \Rightarrow_0 'b)$

where $\text{trd fs p} = \text{trd-aux fs p } 0$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *trd-aux-irred*:

assumes $\neg \text{is-red (set fs) } r$

shows $\neg \text{is-red (set fs) (trd-aux fs p } r)$

$\langle \text{proof} \rangle$

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

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

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

end

end

5 Gröbner Bases and Buchberger's Theorem

theory *Groebner-Bases*
imports *Reduction*
begin

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

context *gd-term*
begin

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

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

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

else 0)

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

5.1 Critical Pairs and S-Polynomials

lemma *crit-pair-same*: fst (crit-pair p p) = snd (crit-pair p p)
(proof)

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

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

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

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

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

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

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

lemma *fst-crit-pair-below-lcs*:
fst (crit-pair p q) \prec_p monomial 1 (term-of-pair (lcs (lp p) (lp q), component-of-term (lt p)))
(proof)

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

lemma *crit-pair-cbelow-same*:

assumes *dickson-grading* d **and** $p \in \text{dgrad-p-set } d \ m$

shows *crit-pair-cbelow-on* $d \ m \ F \ p \ p$

$\langle \text{proof} \rangle$

lemma *crit-pair-cbelow-distinct-component*:

assumes *component-of-term* $(\text{lt } p) \neq \text{component-of-term } (\text{lt } q)$

shows *crit-pair-cbelow-on* $d \ m \ F \ p \ q$

$\langle \text{proof} \rangle$

lemma *crit-pair-cbelow-sym*:

assumes *crit-pair-cbelow-on* $d \ m \ F \ p \ q$

shows *crit-pair-cbelow-on* $d \ m \ F \ q \ p$

$\langle \text{proof} \rangle$

lemma *crit-pair-cs-imp-crit-pair-cbelow-on*:

assumes *dickson-grading* d **and** $F \subseteq \text{dgrad-p-set } d \ m$ **and** $p \in \text{dgrad-p-set } d \ m$

and $q \in \text{dgrad-p-set } d \ m$

and *relation.cs* $(\text{red } F) (\text{fst } (\text{crit-pair } p \ q)) (\text{snd } (\text{crit-pair } p \ q))$

shows *crit-pair-cbelow-on* $d \ m \ F \ p \ q$

$\langle \text{proof} \rangle$

lemma *crit-pair-cbelow-mono*:

assumes *crit-pair-cbelow-on* $d \ m \ F \ p \ q$ **and** $F \subseteq G$

shows *crit-pair-cbelow-on* $d \ m \ G \ p \ q$

$\langle \text{proof} \rangle$

lemma *lcs-red-single-fst-crit-pair*:

assumes $p \neq 0$ **and** *component-of-term* $(\text{lt } p) = \text{component-of-term } (\text{lt } q)$

defines $t1 \equiv \text{lp } p$

defines $t2 \equiv \text{lp } q$

shows *red-single* $(\text{monomial } (- \ 1) (\text{term-of-pair } (\text{lcs } t1 \ t2), \text{component-of-term } (\text{lt } p))))$

$(\text{fst } (\text{crit-pair } p \ q)) \ p \ (\text{lcs } t1 \ t2 - t1)$

$\langle \text{proof} \rangle$

corollary *lcs-red-single-snd-crit-pair*:

assumes $q \neq 0$ **and** *component-of-term* $(\text{lt } p) = \text{component-of-term } (\text{lt } q)$

defines $t1 \equiv \text{lp } p$

defines $t2 \equiv \text{lp } q$

shows *red-single* $(\text{monomial } (- \ 1) (\text{term-of-pair } (\text{lcs } t1 \ t2), \text{component-of-term } (\text{lt } p))))$

$(\text{snd } (\text{crit-pair } p \ q)) \ q \ (\text{lcs } t1 \ t2 - t2)$

$\langle \text{proof} \rangle$

lemma *GB-imp-crit-pair-cbelow-dgrad-p-set*:

assumes *dickson-grading* d **and** $F \subseteq \text{dgrad-p-set } d \ m$ **and** *is-Groebner-basis* F

assumes $p \in F$ **and** $q \in F$ **and** $p \neq 0$ **and** $q \neq 0$

shows *crit-pair-cbelow-on* $d \ m \ F \ p \ q$

<proof>

lemma *spoly-alt*:

assumes $p \neq 0$ **and** $q \neq 0$

shows $\text{spoly } p \ q = \text{fst } (\text{crit-pair } p \ q) - \text{snd } (\text{crit-pair } p \ q)$

<proof>

lemma *spoly-same*: $\text{spoly } p \ p = 0$

<proof>

lemma *spoly-swap*: $\text{spoly } p \ q = - \text{spoly } q \ p$

<proof>

lemma *spoly-red-zero-imp-crit-pair-cbelow-on*:

assumes *dickson-grading* d **and** $F \subseteq \text{dgrad-p-set } d \ m$ **and** $p \in \text{dgrad-p-set } d \ m$

and $q \in \text{dgrad-p-set } d \ m$ **and** $p \neq 0$ **and** $q \neq 0$ **and** $(\text{red } F)^{**} (\text{spoly } p \ q) = 0$

shows *crit-pair-cbelow-on* $d \ m \ F \ p \ q$

<proof>

lemma *dgrad-p-set-le-spoly-zero*: $\text{dgrad-p-set-le } d \ \{\text{spoly } p \ 0\} \ \{p\}$

<proof>

lemma *dgrad-p-set-le-spoly*:

assumes *dickson-grading* d

shows $\text{dgrad-p-set-le } d \ \{\text{spoly } p \ q\} \ \{p, q\}$

<proof>

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

assumes *dickson-grading* d **and** $p \in \text{dgrad-p-set } d \ m$ **and** $q \in \text{dgrad-p-set } d \ m$

shows $\text{spoly } p \ q \in \text{dgrad-p-set } d \ m$

<proof>

lemma *components-spoly-subset*: $\text{component-of-term } \text{'keys } (\text{spoly } p \ q) \subseteq \text{component-of-term } \text{'Keys } \{p, q\}$

<proof>

lemma *pmdl-closed-spoly*:

assumes $p \in \text{pmdl } F$ **and** $q \in \text{pmdl } F$

shows $\text{spoly } p \ q \in \text{pmdl } F$

<proof>

5.2 Buchberger's Theorem

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

easier to prove the correctness of Buchberger's second criterion for avoiding useless pairs.

lemma *crit-pair-cbelow-imp-confluent-dgrad-p-set:*

assumes *dg: dickson-grading d and $F \subseteq \text{dgrad-p-set } d \ m$*

assumes main: $\bigwedge p \ q. p \in F \implies q \in F \implies p \neq 0 \implies q \neq 0 \implies \text{crit-pair-cbelow-on } d \ m \ F \ p \ q$

shows *relation.is-confluent-on (red F) (dgrad-p-set d m)*

<proof>

corollary *crit-pair-cbelow-imp-GB-dgrad-p-set:*

assumes *dickson-grading d and $F \subseteq \text{dgrad-p-set } d \ m$*

assumes $\bigwedge p \ q. p \in F \implies q \in F \implies p \neq 0 \implies q \neq 0 \implies \text{crit-pair-cbelow-on } d \ m \ F \ p \ q$

shows *is-Groebner-basis F*

<proof>

corollary *Buchberger-criterion-dgrad-p-set:*

assumes *dickson-grading d and $F \subseteq \text{dgrad-p-set } d \ m$*

assumes $\bigwedge p \ q. p \in F \implies q \in F \implies p \neq 0 \implies q \neq 0 \implies p \neq q \implies \text{component-of-term (lt p) = component-of-term (lt q)} \implies (\text{red } F)^{**} (\text{spoly } p \ q) \ 0$

shows *is-Groebner-basis F*

<proof>

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

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

assumes *is-Groebner-basis G and $f \in \text{pmdl } G$*

shows $(\text{red } G)^{**} f \ 0$

<proof>

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

assumes *is-Groebner-basis G and $f \neq 0$ and $f \in \text{pmdl } G$*

shows *is-red G f*

<proof>

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

<proof>

lemma *is-Groebner-basis-singleton: is-Groebner-basis {f}*

<proof>

5.3 Buchberger's Criteria for Avoiding Useless Pairs

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

lemma *(in gd-powerprod) product-criterion:*

assumes *dickson-grading d and $F \subseteq \text{punit.dgrad-p-set } d \ m$ and $p \in F$ and $q \in$*

F
and $p \neq 0$ **and** $q \neq 0$ **and** $gcs (punit.lt p) (punit.lt q) = 0$
shows $punit.crit-pair-cbelow-on d m F p q$
 $\langle proof \rangle$

lemma *chain-criterion*:

assumes *dickson-grading* d **and** $F \subseteq dgrad-p-set d m$ **and** $p \in F$ **and** $q \in F$
and $p \neq 0$ **and** $q \neq 0$ **and** $lp r adds lcs (lp p) (lp q)$
and $component-of-term (lt r) = component-of-term (lt p)$
and $pr: crit-pair-cbelow-on d m F p r$ **and** $rq: crit-pair-cbelow-on d m F r q$
shows $crit-pair-cbelow-on d m F p q$
 $\langle proof \rangle$

5.4 Weak and Strong Gröbner Bases

lemma *ord-p-wf-on*:

assumes *dickson-grading* d
shows $wfp-on (\prec_p) (dgrad-p-set d m)$
 $\langle proof \rangle$

lemma *is-red-implies-0-red-dgrad-p-set*:

assumes *dickson-grading* d **and** $B \subseteq dgrad-p-set d m$
assumes $pmdl B \subseteq pmdl A$ **and** $\bigwedge q. q \in pmdl A \implies q \in dgrad-p-set d m \implies q \neq 0 \implies is-red B q$
and $p \in pmdl A$ **and** $p \in dgrad-p-set d m$
shows $(red B)** p 0$
 $\langle proof \rangle$

lemma *is-red-implies-0-red-dgrad-p-set'*:

assumes *dickson-grading* d **and** $B \subseteq dgrad-p-set d m$
assumes $pmdl B \subseteq pmdl A$ **and** $\bigwedge q. q \in pmdl A \implies q \neq 0 \implies is-red B q$
and $p \in pmdl A$
shows $(red B)** p 0$
 $\langle proof \rangle$

lemma *pmdl-eqI-adds-lt-dgrad-p-set*:

fixes $G::('t \Rightarrow_0 'b::field) set$
assumes *dickson-grading* d **and** $G \subseteq dgrad-p-set d m$ **and** $B \subseteq dgrad-p-set d m$
and $pmdl G \subseteq pmdl B$
assumes $\bigwedge f. f \in pmdl B \implies f \in dgrad-p-set d m \implies f \neq 0 \implies (\exists g \in G. g \neq 0 \wedge lt g adds_t lt f)$
shows $pmdl G = pmdl B$
 $\langle proof \rangle$

lemma *pmdl-eqI-adds-lt-dgrad-p-set'*:

fixes $G::('t \Rightarrow_0 'b::field) set$
assumes *dickson-grading* d **and** $G \subseteq dgrad-p-set d m$ **and** $pmdl G \subseteq pmdl B$
assumes $\bigwedge f. f \in pmdl B \implies f \neq 0 \implies (\exists g \in G. g \neq 0 \wedge lt g adds_t lt f)$

shows $\text{pmdl } G = \text{pmdl } B$
 ⟨proof⟩

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

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

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

lemma *weak-GB-is-strong-GB-dgrad-p-set:*
assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$
assumes $\bigwedge f. f \in \text{pmdl } G \implies f \in \text{dgrad-p-set } d \ m \implies (\text{red } G)^{**} f \ 0$
shows *is-Groebner-basis* G
 ⟨proof⟩

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

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

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

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

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

lemma *GB-adds-lt*:

assumes *is-Groebner-basis* G **and** $f \in \text{pmdl } G$ **and** $f \neq 0$
obtains g **where** $g \in G$ **and** $g \neq 0$ **and** $\text{lt } g \text{ adds}_t \text{ lt } f$

<proof>

lemma *isGB-I-adds-lt*:

assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$

assumes $\bigwedge f. f \in \text{pmdl } G \implies f \in \text{dgrad-p-set } d \ m \implies f \neq 0 \implies (\exists g \in G. g \neq 0 \wedge \text{lt } g \text{ adds}_t \text{ lt } f)$

shows *is-Groebner-basis* G

<proof>

lemma *GB-alt-3-dgrad-p-set*:

assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$

shows *is-Groebner-basis* $G \iff (\forall f \in \text{pmdl } G. f \neq 0 \implies (\exists g \in G. g \neq 0 \wedge \text{lt } g \text{ adds}_t \text{ lt } f))$

(**is** $?L \iff ?R$)

<proof>

lemma *GB-insert*:

assumes *is-Groebner-basis* G **and** $f \in \text{pmdl } G$

shows *is-Groebner-basis* (*insert* f G)

<proof>

lemma *GB-subset*:

assumes *is-Groebner-basis* G **and** $G \subseteq G'$ **and** $\text{pmdl } G' = \text{pmdl } G$

shows *is-Groebner-basis* G'

<proof>

lemma (**in** *ordered-term*) *GB-remove-0-stable-GB*:

assumes *is-Groebner-basis* G

shows *is-Groebner-basis* ($G - \{0\}$)

<proof>

lemmas *is-red-implies-0-red-finite = is-red-implies-0-red-dgrad-p-set'*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemmas *GB-implies-unique-nf-finite = GB-implies-unique-nf-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemmas *GB-alt-2-finite = GB-alt-2-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemmas *GB-alt-3-finite = GB-alt-3-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

lemmas *pmdl-eqI-adds-lt-finite = pmdl-eqI-adds-lt-dgrad-p-set'*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

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

definition *spoly-rep* :: ('a ⇒ nat) ⇒ nat ⇒ ('t ⇒₀ 'b) set ⇒ ('t ⇒₀ 'b) ⇒ ('t ⇒₀ 'b::field) ⇒ bool

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

lemma *spoly-repI*:

spoly g1 g2 = $(\sum_{g \in G}. q \ g \odot g) \implies (\bigwedge g. q \ g \in \text{punit.dgrad-p-set } d \ m) \implies$
 $(\bigwedge g. q \ g \odot g \neq 0 \implies \text{lt } (q \ g \odot g) \prec_t \text{term-of-pair } (\text{lcs } (\text{lp } g1) (\text{lp } g2),$
component-of-term (lt g2)))) \implies

spoly-rep d m G g1 g2
 ⟨proof⟩

lemma *spoly-repI-zero*:

assumes *spoly* g1 g2 = 0
shows *spoly-rep* d m G g1 g2
 ⟨proof⟩

lemma *spoly-repE*:

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

corollary *isGB-D-spoly-rep*:

assumes *dickson-grading* d **and** *is-Groebner-basis* G **and** $G \subseteq \text{dgrad-p-set } d \ m$
and *finite* G
and g1 ∈ G **and** g2 ∈ G **and** g1 ≠ 0 **and** g2 ≠ 0
shows *spoly-rep* d m G g1 g2
 ⟨proof⟩

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

lemma *isGB-I-spoly-rep*:

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

5.6 Replacing Elements in Gröbner Bases

lemma *replace-in-dgrad-p-set*:

assumes $G \subseteq \text{dgrad-p-set } d \ m$
obtains n **where** $q \in \text{dgrad-p-set } d \ n$ **and** $G \subseteq \text{dgrad-p-set } d \ n$
and $\text{insert } q \ (G - \{p\}) \subseteq \text{dgrad-p-set } d \ n$
 <proof>

lemma *GB-replace-lt-adds-stable-GB-dgrad-p-set:*
assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$
assumes *isGB: is-Groebner-basis* G **and** $q \neq 0$ **and** $q: q \in (\text{pmdl } G)$ **and** *lt* q
adds_t *lt* p
shows *is-Groebner-basis* $(\text{insert } q \ (G - \{p\}))$ (**is** *is-Groebner-basis* $?G'$)
 <proof>

lemma *GB-replace-lt-adds-stable-pmdl-dgrad-p-set:*
assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$
assumes *isGB: is-Groebner-basis* G **and** $q \neq 0$ **and** $q \in \text{pmdl } G$ **and** *lt* q *adds_t*
lt p
shows $\text{pmdl } (\text{insert } q \ (G - \{p\})) = \text{pmdl } G$ (**is** $\text{pmdl } ?G' = \text{pmdl } G$)
 <proof>

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

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

lemma *GB-replace-red-rtranclp-stable-GB-dgrad-p-set:*
assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$
assumes *isGB: is-Groebner-basis* G **and** $p \in G$ **and** *ptoq: (red* $(G - \{p\}))^{**} \ p$
 q
shows *is-Groebner-basis* $(\text{insert } q \ (G - \{p\}))$
 <proof>

lemma *GB-replace-red-rtranclp-stable-pmdl-dgrad-p-set:*
assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$
assumes *isGB: is-Groebner-basis* G **and** $p \in G$ **and** *ptoq: (red* $(G - \{p\}))^{**} \ p$
 q
shows $\text{pmdl } (\text{insert } q \ (G - \{p\})) = \text{pmdl } G$
 <proof>

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

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

5.7 An Inconstructive Proof of the Existence of Finite Gröbner Bases

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

The preceding lemma justifies the following definition.

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

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

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

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

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

<proof>

lemma *finite-imp-finite-component-Keys:*

assumes *finite F*

shows *finite (component-of-term ' Keys F)*

<proof>

lemma *finite-some-GB-finite: finite F \implies finite (some-GB F)*

<proof>

lemma *some-GB-isGB-finite: finite F \implies is-Groebner-basis (some-GB F)*

<proof>

lemma *some-GB-pmdl-finite: finite F \implies pmdl (some-GB F) = pmdl F*

<proof>

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

5.8 Relation *red-supset*

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

definition *red-supset::('t \Rightarrow_0 'b::field) set \Rightarrow ('t \Rightarrow_0 'b) set \Rightarrow bool (infixl *\sqsupset* 50)*

where *red-supset A B \equiv ($\exists p. is-red A p \wedge \neg is-red B p$) \wedge ($\forall p. is-red B p \longrightarrow is-red A p$)*

lemma *red-supsetE:*

assumes *A $\sqsupset p$ B*

obtains *p where is-red A p and $\neg is-red B p$*

<proof>

lemma *red-supsetD:*

assumes *a1: A $\sqsupset p$ B and a2: is-red B p*

shows *is-red A p*

<proof>

lemma *red-supsetI [intro]:*

assumes $\bigwedge q. is-red B q \implies is-red A q$ **and** *is-red A p and $\neg is-red B p$*

shows *A $\sqsupset p$ B*

<proof>

lemma *red-supset-insertI:*

assumes *x $\neq 0$ and $\neg is-red A x$*

shows *(insert x A) $\sqsupset p$ A*

<proof>

lemma *red-supset-transitive*:
assumes $A \sqsupset_p B$ **and** $B \sqsupset_p C$
shows $A \sqsupset_p C$
 \langle *proof* \rangle

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

end

lemma *in-lex-prod-alt*:
 $(x, y) \in r \text{ <*\textit{lex}*> } s \longleftrightarrow (((fst \ x), (fst \ y)) \in r \vee (fst \ x = fst \ y \wedge ((snd \ x), (snd \ y)) \in s))$
 \langle *proof* \rangle

5.9 Context *od-term*

context *od-term*
begin

lemmas *red-wf = red-wf-dgrad-p-set* $[OF \ \textit{dickson-grading-zero subset-dgrad-p-set-zero}]$
lemmas *Buchberger-criterion = Buchberger-criterion-dgrad-p-set* $[OF \ \textit{dickson-grading-zero subset-dgrad-p-set-zero}]$

end

end

6 A General Algorithm Schema for Computing Gröbner Bases

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

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

6.1 processed

definition *minus-pairs* (**infixl** $\langle -_p \rangle$ 65) **where** *minus-pairs* $A B = A - (B \cup \text{prod.swap } B)$

definition *Int-pairs* (**infixl** $\langle \cap_p \rangle$ 65) **where** *Int-pairs* $A B = A \cap (B \cup \text{prod.swap } B)$

definition *in-pair* (**infix** $\langle \in_p \rangle$ 50) **where** *in-pair* $p A \longleftrightarrow (p \in A \cup \text{prod.swap } A)$

definition *subset-pairs* (**infix** $\langle \subseteq_p \rangle$ 50) **where** *subset-pairs* $A B \longleftrightarrow (\forall x. x \in_p A \longrightarrow x \in_p B)$

abbreviation *not-in-pair* (**infix** $\langle \notin_p \rangle$ 50) **where** *not-in-pair* $p A \equiv \neg p \in_p A$

lemma *in-pair-alt*: $p \in_p A \longleftrightarrow (p \in A \vee \text{prod.swap } p \in A)$
<proof>

lemma *in-pair-iff*: $(a, b) \in_p A \longleftrightarrow ((a, b) \in A \vee (b, a) \in A)$
<proof>

lemma *in-pair-minus-pairs* [*simp*]: $p \in_p A -_p B \longleftrightarrow (p \in_p A \wedge p \notin_p B)$
<proof>

lemma *in-minus-pairs* [*simp*]: $p \in A -_p B \longleftrightarrow (p \in A \wedge p \notin_p B)$
<proof>

lemma *in-pair-Int-pairs* [*simp*]: $p \in_p A \cap_p B \longleftrightarrow (p \in_p A \wedge p \in_p B)$
<proof>

lemma *in-pair-Un* [*simp*]: $p \in_p A \cup B \longleftrightarrow (p \in_p A \vee p \in_p B)$
<proof>

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

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

lemma *subset-pairsI* [*intro*]:
assumes $\bigwedge x. x \in_p A \implies x \in_p B$
shows $A \subseteq_p B$
<proof>

lemma *subset-pairsD* [*trans*]:
assumes $x \in_p A$ **and** $A \subseteq_p B$
shows $x \in_p B$
<proof>

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

lemma *processed-alt*:
 $processed\ (a, b)\ xs\ ps \longleftrightarrow ((a \in set\ xs) \wedge (b \in set\ xs) \wedge (a, b) \notin_p\ set\ ps)$
 $\langle proof \rangle$

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

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

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

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

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

lemma *processed-Cons*:
assumes $processed\ (a, b)\ xs\ ps$
and $a1: a = p \implies b = q \implies thesis$
and $a2: a = q \implies b = p \implies thesis$
and $a3: processed\ (a, b)\ xs\ ((p, q) \# ps) \implies thesis$
shows $thesis$
 $\langle proof \rangle$

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

6.2 Algorithm Schema

6.2.1 *const-lt-component*

context *ordered-term*
begin

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

lemma *const-lt-component-SomeI*:
assumes lp p = 0 **and** component-of-term (lt p) = cmp
shows *const-lt-component* p = Some cmp
⟨proof⟩

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

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

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

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

end

6.2.2 Type synonyms

type-synonym ('a, 'b, 'c) pdata' = ('a \Rightarrow_0 'b) \times 'c
type-synonym ('a, 'b, 'c) pdata = ('a \Rightarrow_0 'b) \times nat \times 'c
type-synonym ('a, 'b, 'c) pdata-pair = ('a, 'b, 'c) pdata \times ('a, 'b, 'c) pdata
type-synonym ('a, 'b, 'c, 'd) selT = ('a, 'b, 'c) pdata list \Rightarrow ('a, 'b, 'c) pdata list
 \Rightarrow
 ('a, 'b, 'c) pdata-pair list \Rightarrow nat \times 'd \Rightarrow ('a, 'b, 'c)
pdata-pair list
type-synonym ('a, 'b, 'c, 'd) complT = ('a, 'b, 'c) pdata list \Rightarrow ('a, 'b, 'c) pdata
list \Rightarrow
 ('a, 'b, 'c) pdata-pair list \Rightarrow ('a, 'b, 'c) pdata-pair list
 \Rightarrow
 nat \times 'd \Rightarrow (('a, 'b, 'c) pdata' list \times 'd)
type-synonym ('a, 'b, 'c, 'd) apT = ('a, 'b, 'c) pdata list \Rightarrow ('a, 'b, 'c) pdata list
 \Rightarrow

$$\begin{aligned} & ('a, 'b, 'c) \text{ pdata-pair list} \Rightarrow ('a, 'b, 'c) \text{ pdata list} \Rightarrow \\ \text{nat} \times 'd & \Rightarrow \\ & ('a, 'b, 'c) \text{ pdata-pair list} \\ \text{type-synonym } ('a, 'b, 'c, 'd) \text{ abT} & = ('a, 'b, 'c) \text{ pdata list} \Rightarrow ('a, 'b, 'c) \text{ pdata list} \\ & \Rightarrow \\ & ('a, 'b, 'c) \text{ pdata list} \Rightarrow \text{nat} \times 'd \Rightarrow ('a, 'b, 'c) \text{ pdata list} \end{aligned}$$

6.2.3 Specification of the *selector* parameter

definition *sel-spec* :: ('a, 'b, 'c, 'd) selT ⇒ bool
where *sel-spec sel* ↔
 $(\forall gs \ bs \ ps \ data. \ ps \neq [] \longrightarrow (sel \ gs \ bs \ ps \ data \neq [] \wedge set \ (sel \ gs \ bs \ ps \ data) \subseteq set \ ps))$

lemma *sel-specI*:
assumes $\bigwedge gs \ bs \ ps \ data. \ ps \neq [] \implies (sel \ gs \ bs \ ps \ data \neq [] \wedge set \ (sel \ gs \ bs \ ps \ data) \subseteq set \ ps)$
shows *sel-spec sel*
 ⟨proof⟩

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

lemma *sel-specD2*:
assumes *sel-spec sel* **and** $ps \neq []$
shows $set \ (sel \ gs \ bs \ ps \ data) \subseteq set \ ps$
 ⟨proof⟩

6.2.4 Specification of the *add-basis* parameter

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

lemma *ab-specI*:
assumes $\bigwedge gs \ bs \ ns \ data. \ ns \neq [] \implies set \ (ab \ gs \ bs \ ns \ data) = set \ bs \cup set \ ns$
and $\bigwedge gs \ bs \ data. \ ab \ gs \ bs \ [] \ data = bs$
shows *ab-spec ab*
 ⟨proof⟩

lemma *ab-specD1*:
assumes *ab-spec ab*
shows $set \ (ab \ gs \ bs \ ns \ data) = set \ bs \cup set \ ns$
 ⟨proof⟩

lemma *ab-specD2*:
assumes *ab-spec ab*

shows $ab\ gs\ bs\ []\ data = bs$
 $\langle proof \rangle$

6.2.5 Specification of the *add-pairs* parameter

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

lemma $unique-idxI$:
assumes $\bigwedge f\ g. f \in set\ bs \Longrightarrow g \in set\ bs \Longrightarrow fst\ (snd\ f) = fst\ (snd\ g) \Longrightarrow f = g$
and $\bigwedge f. f \in set\ bs \Longrightarrow fst\ (snd\ f) < fst\ data$
shows $unique-idx\ bs\ data$
 $\langle proof \rangle$

lemma $unique-idxD1$:
assumes $unique-idx\ bs\ data$ **and** $f \in set\ bs$ **and** $g \in set\ bs$ **and** $fst\ (snd\ f) = fst\ (snd\ g)$
 $(snd\ f)$
shows $f = g$
 $\langle proof \rangle$

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

lemma $unique-idx-Nil$: $unique-idx\ []\ data$
 $\langle proof \rangle$

lemma $unique-idx-subset$:
assumes $unique-idx\ bs\ data$ **and** $set\ bs' \subseteq set\ bs$
shows $unique-idx\ bs'\ data$
 $\langle proof \rangle$

context $gd-term$
begin

definition $ap-spec :: ('t, 'b::field, 'c, 'd)\ apT \Rightarrow bool$
where $ap-spec\ ap \longleftrightarrow (\forall gs\ bs\ ps\ hs\ data.$
 $set\ (ap\ gs\ bs\ ps\ hs\ data) \subseteq set\ ps \cup (set\ hs \times (set\ gs \cup set\ bs \cup set\ hs)) \wedge$
 $(\forall B\ d\ m. \forall h \in set\ hs. \forall g \in set\ gs \cup set\ bs \cup set\ hs. dickson-grading\ d \longrightarrow$
 $set\ gs \cup set\ bs \cup set\ hs \subseteq B \longrightarrow fst\ 'B \subseteq dgrad-p-set\ d\ m \longrightarrow$
 $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow unique-idx\ (gs\ @\ bs\ @\ hs)\ data \longrightarrow$
 $is-Groebner-basis\ (fst\ 'set\ gs) \longrightarrow h \neq g \longrightarrow fst\ h \neq 0 \longrightarrow fst\ g \neq 0 \longrightarrow$
 $(\forall a\ b. (a, b) \in_p\ set\ (ap\ gs\ bs\ ps\ hs\ data) \longrightarrow fst\ a \neq 0 \longrightarrow fst\ b \neq 0 \longrightarrow$
 $crit-pair-cbelow-on\ d\ m\ (fst\ 'B)\ (fst\ a)\ (fst\ b)) \longrightarrow$
 $(\forall a\ b. a \in set\ gs \cup set\ bs \longrightarrow b \in set\ gs \cup set\ bs \longrightarrow fst\ a \neq 0 \longrightarrow fst\ b \neq$
 $0 \longrightarrow$

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

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

lemma *ap-specI*:

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

$\text{crit-pair-cbelow-on } d \ m \ (\text{fst } ' B) \ (\text{fst } h) \ (\text{fst } g)$
shows $\text{ap-spec } \text{ap}$
 $\langle \text{proof} \rangle$

lemma ap-specD1 :
assumes $\text{ap-spec } \text{ap}$
shows $\text{set } (\text{ap } \text{gs } \text{bs } \text{ps } \text{hs } \text{data}) \subseteq \text{set } \text{ps} \cup (\text{set } \text{hs} \times (\text{set } \text{gs} \cup \text{set } \text{bs} \cup \text{set } \text{hs}))$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{ap-spec-Nil-subset}$:
assumes $\text{ap-spec } \text{ap}$
shows $\text{set } (\text{ap } \text{gs } \text{bs } \text{ps } [] \ \text{data}) \subseteq \text{set } \text{ps}$
 $\langle \text{proof} \rangle$

lemma $\text{ap-spec-fst-subset}$:
assumes $\text{ap-spec } \text{ap}$
shows $\text{fst } ' \text{set } (\text{ap } \text{gs } \text{bs } \text{ps } \text{hs } \text{data}) \subseteq \text{fst } ' \text{set } \text{ps} \cup \text{set } \text{hs}$
 $\langle \text{proof} \rangle$

lemma $\text{ap-spec-snd-subset}$:
assumes $\text{ap-spec } \text{ap}$
shows $\text{snd } ' \text{set } (\text{ap } \text{gs } \text{bs } \text{ps } \text{hs } \text{data}) \subseteq \text{snd } ' \text{set } \text{ps} \cup \text{set } \text{gs} \cup \text{set } \text{bs} \cup \text{set } \text{hs}$
 $\langle \text{proof} \rangle$

lemma *ap-spec-inE*:

assumes *ap-spec ap* **and** $(p, q) \in \text{set } (ap \text{ } gs \text{ } bs \text{ } ps \text{ } hs \text{ } data)$

assumes 1: $(p, q) \in \text{set } ps \implies \text{thesis}$

assumes 2: $p \in \text{set } hs \implies q \in \text{set } gs \cup \text{set } bs \cup \text{set } hs \implies \text{thesis}$

shows *thesis*

<proof>

lemma *subset-Times-ap*:

assumes *ap-spec ap* **and** *ab-spec ab* **and** $\text{set } ps \subseteq \text{set } bs \times (\text{set } gs \cup \text{set } bs)$

shows $\text{set } (ap \text{ } gs \text{ } bs \text{ } (ps \text{ } \text{---} \text{ } sps) \text{ } hs \text{ } data) \subseteq \text{set } (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data) \times (\text{set } gs \cup \text{set } (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data))$

<proof>

6.2.6 Function *args-to-set*

definition *args-to-set* :: $(t, 'b::\text{field}, 'c) \text{ pdata list} \times (t, 'b, 'c) \text{ pdata list} \times (t, 'b, 'c) \text{ pdata-pair list} \Rightarrow (t \Rightarrow_0 'b) \text{ set}$

where $\text{args-to-set } x = \text{fst } ' (\text{set } (\text{fst } x) \cup \text{set } (\text{fst } (\text{snd } x)) \cup \text{fst } ' \text{ set } (\text{snd } (\text{snd } x))) \cup \text{snd } ' \text{ set } (\text{snd } (\text{snd } x))$

lemma *args-to-set-alt*:

$\text{args-to-set } (gs, bs, ps) = \text{fst } ' \text{ set } gs \cup \text{fst } ' \text{ set } bs \cup \text{fst } ' \text{ fst } ' \text{ set } ps \cup \text{fst } ' \text{ snd } ' \text{ set } ps$

<proof>

lemma *args-to-set-subset-Times*:

assumes $\text{set } ps \subseteq \text{set } bs \times (\text{set } gs \cup \text{set } bs)$

shows $\text{args-to-set } (gs, bs, ps) = \text{fst } ' \text{ set } gs \cup \text{fst } ' \text{ set } bs$

<proof>

lemma *args-to-set-subset*:

assumes *ap-spec ap* **and** *ab-spec ab*

shows $\text{args-to-set } (gs, ab \text{ } gs \text{ } bs \text{ } hs \text{ } data, ap \text{ } gs \text{ } bs \text{ } ps \text{ } hs \text{ } data) \subseteq$

$\text{fst } ' (\text{set } gs \cup \text{set } bs \cup \text{fst } ' \text{ set } ps \cup \text{snd } ' \text{ set } ps \cup \text{set } hs) \text{ (is } ?l \subseteq \text{fst } ' ?r)$

<proof>

lemma *args-to-set-alt2*:

assumes *ap-spec ap* **and** *ab-spec ab* **and** $\text{set } ps \subseteq \text{set } bs \times (\text{set } gs \cup \text{set } bs)$

shows $\text{args-to-set } (gs, ab \text{ } gs \text{ } bs \text{ } hs \text{ } data, ap \text{ } gs \text{ } bs \text{ } (ps \text{ } \text{---} \text{ } sps) \text{ } hs \text{ } data) =$

$\text{fst } ' (\text{set } gs \cup \text{set } bs \cup \text{set } hs) \text{ (is } ?l = \text{fst } ' ?r)$

<proof>

lemma *args-to-set-subset1*:

assumes $\text{set } gs1 \subseteq \text{set } gs2$

shows $\text{args-to-set } (gs1, bs, ps) \subseteq \text{args-to-set } (gs2, bs, ps)$

<proof>

lemma *args-to-set-subset2*:
assumes *set bs1* \subseteq *set bs2*
shows *args-to-set* (*gs*, *bs1*, *ps*) \subseteq *args-to-set* (*gs*, *bs2*, *ps*)
 \langle *proof* \rangle

lemma *args-to-set-subset3*:
assumes *set ps1* \subseteq *set ps2*
shows *args-to-set* (*gs*, *bs*, *ps1*) \subseteq *args-to-set* (*gs*, *bs*, *ps2*)
 \langle *proof* \rangle

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

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

definition *count-const-lt-components* :: ('t, 'b::zero, 'c) *pdata' list* \Rightarrow *nat*
where *count-const-lt-components* *hs* = *length* (*remdups* (*filter* ($\lambda x. x \neq$ None)
(*map* (*const-lt-component* \circ *fst*) *hs*)))

definition *count-rem-components* :: ('t, 'b::zero, 'c) *pdata' list* \Rightarrow *nat*
where *count-rem-components* *bs* = *length* (*remdups* (*map* *component-of-term*
(*Keys-to-list* (*map* *fst* *bs*)))) -
count-const-lt-components [*b* \leftarrow *bs* . *fst* *b* \neq 0]

lemma *count-const-lt-components-alt*:
count-const-lt-components *hs* = *card* (*const-lt-component* ' *fst* ' *set* *hs* - {None})
 \langle *proof* \rangle

lemma *count-rem-components-alt*:
count-rem-components *bs* + *card* (*const-lt-component* ' (*fst* ' *set* *bs* - {0}) -
{None}) =
card (*component-of-term* ' *Keys* (*fst* ' *set* *bs*))
 \langle *proof* \rangle

lemma *rem-comps-spec-count-rem-components*: *rem-comps-spec* *bs* (*count-rem-components*
bs, *data*)
 \langle *proof* \rangle

definition *full-gb* :: ('t, 'b, 'c) *pdata list* \Rightarrow ('t, 'b::zero-neq-one, 'c::default) *pdata*
list
where *full-gb* *bs* = *map* ($\lambda k. (\text{monomial } 1 (\text{term-of-pair } (0, k)), 0, \text{default}))$
(*remdups* (*map* *component-of-term* (*Keys-to-list* (*map* *fst* *bs*))))

lemma *fst-set-full-gb*:
fst ' *set* (*full-gb* *bs*) = ($\lambda v. \text{monomial } 1 (\text{term-of-pair } (0, \text{component-of-term } v))$)
' *Keys* (*fst* ' *set* *bs*)

$\langle \text{proof} \rangle$

lemma *Keys-full-gb*:

$\text{Keys } (fst \text{ ' set (full-gb bs)}) = (\lambda v. \text{term-of-pair } (0, \text{component-of-term } v)) \text{ ' Keys } (fst \text{ ' set bs})$
 $\langle \text{proof} \rangle$

lemma *pps-full-gb*: $\text{pp-of-term ' Keys } (fst \text{ ' set (full-gb bs)}) \subseteq \{0\}$
 $\langle \text{proof} \rangle$

lemma *components-full-gb*:

$\text{component-of-term ' Keys } (fst \text{ ' set (full-gb bs)}) = \text{component-of-term ' Keys } (fst \text{ ' set bs})$
 $\langle \text{proof} \rangle$

lemma *full-gb-is-full-pmdl*: $\text{is-full-pmdl } (fst \text{ ' set (full-gb bs)})$

for $bs::('t, 'b::\text{field}, 'c::\text{default}) \text{ pdata list}$
 $\langle \text{proof} \rangle$

In fact, $\text{is-full-pmdl } (fst \text{ ' set (full-gb ?bs)})$ also holds if $'b$ is no field.

lemma *full-gb-isGB*: $\text{is-Groebner-basis } (fst \text{ ' set (full-gb bs)})$
 $\langle \text{proof} \rangle$

6.2.8 Specification of the *completion* parameter

definition *compl-struct* :: $('t, 'b::\text{field}, 'c, 'd) \text{ compl}T \Rightarrow \text{bool}$

where $\text{compl-struct } \text{compl} \longleftrightarrow$
 $(\forall gs \ bs \ ps \ sps \ data. \ sps \neq [] \longrightarrow \text{set } sps \subseteq \text{set } ps \longrightarrow$
 $(\forall d. \ \text{dickson-grading } d \longrightarrow$
 $\text{dgrad-p-set-le } d \ (fst \text{ ' (set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data))))))$
 $(\text{args-to-set } (gs, \ bs, \ ps))) \wedge$
 $\text{component-of-term ' Keys } (fst \text{ ' (set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data)))) \subseteq$
 $\text{component-of-term ' Keys } (\text{args-to-set } (gs, \ bs, \ ps)) \wedge$
 $0 \notin \text{fst ' set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data)) \wedge$
 $(\forall h \in \text{set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data)). \ \forall b \in \text{set } gs \cup \text{set } bs.$
 $\text{fst } b \neq 0 \longrightarrow \neg \text{lt (fst } b) \ \text{adds}_t \ \text{lt (fst } h))$

lemma *compl-structI*:

assumes $\bigwedge d \ gs \ bs \ ps \ sps \ data. \ \text{dickson-grading } d \Longrightarrow \ sps \neq [] \Longrightarrow \ \text{set } sps \subseteq \text{set } ps \Longrightarrow$
 $\text{dgrad-p-set-le } d \ (fst \text{ ' (set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data))))$
 $(\text{args-to-set } (gs, \ bs, \ ps))$
assumes $\bigwedge gs \ bs \ ps \ sps \ data. \ \ sps \neq [] \Longrightarrow \ \text{set } sps \subseteq \text{set } ps \Longrightarrow$
 $\text{component-of-term ' Keys } (fst \text{ ' (set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data)))) \subseteq$
 $\text{component-of-term ' Keys } (\text{args-to-set } (gs, \ bs, \ ps))$
assumes $\bigwedge gs \ bs \ ps \ sps \ data. \ \ sps \neq [] \Longrightarrow \ \text{set } sps \subseteq \text{set } ps \Longrightarrow \ 0 \notin \text{fst ' set (fst (compl } gs \ bs \ (ps \ \text{-- } \ sps) \ sps \ data))$

assumes $\bigwedge gs\ bs\ ps\ sps\ h\ b\ data.\ sps \neq [] \implies set\ sps \subseteq set\ ps \implies h \in set\ (fst\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)) \implies$
 $b \in set\ gs \cup set\ bs \implies fst\ b \neq 0 \implies \neg lt\ (fst\ b)\ adds_t\ lt\ (fst\ h)$
shows *compl-struct compl*
<proof>

lemma *compl-structD1:*

assumes *compl-struct compl* **and** *dickson-grading d* **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$
shows *dgrad-p-set-le d (fst ' (set (fst (compl gs bs (ps --- sps) sps data))))*
(args-to-set (gs, bs, ps))
<proof>

lemma *compl-structD2:*

assumes *compl-struct compl* **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$
shows *component-of-term ' Keys (fst ' (set (fst (compl gs bs (ps --- sps) sps data))))*
 \subseteq
component-of-term ' Keys (args-to-set (gs, bs, ps))
<proof>

lemma *compl-structD3:*

assumes *compl-struct compl* **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$
shows $0 \notin fst\ ' set\ (fst\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data))$
<proof>

lemma *compl-structD4:*

assumes *compl-struct compl* **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$
and $h \in set\ (fst\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data))$ **and** $b \in set\ gs \cup set\ bs$
and $fst\ b \neq 0$
shows $\neg lt\ (fst\ b)\ adds_t\ lt\ (fst\ h)$
<proof>

definition *struct-spec* :: $(t, 'b::field, 'c, 'd)\ selT \Rightarrow (t, 'b, 'c, 'd)\ apT \Rightarrow (t, 'b, 'c, 'd)\ abT \Rightarrow$

$$(t, 'b, 'c, 'd)\ complT \Rightarrow bool$$

where *struct-spec sel ap ab compl* $\longleftrightarrow (sel\ \text{---}\ sel \wedge ap\ \text{---}\ ap \wedge ab\ \text{---}\ ab \wedge compl\ \text{---}\ compl)$

lemma *struct-specI:*

assumes *sel-spec sel* **and** *ap-spec ap* **and** *ab-spec ab* **and** *compl-struct compl*
shows *struct-spec sel ap ab compl*
<proof>

lemma *struct-specD1:*

assumes *struct-spec sel ap ab compl*
shows *sel-spec sel*
<proof>

lemma *struct-specD2:*

assumes *struct-spec sel ap ab compl*
shows *ap-spec ap*
 \langle *proof* \rangle

lemma *struct-specD3*:
assumes *struct-spec sel ap ab compl*
shows *ab-spec ab*
 \langle *proof* \rangle

lemma *struct-specD4*:
assumes *struct-spec sel ap ab compl*
shows *compl-struct compl*
 \langle *proof* \rangle

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

definition *compl-pmdl* :: (*t, 'b::field, 'c, 'd*) *complT* \Rightarrow *bool*
where *compl-pmdl compl* \longleftrightarrow
 $(\forall$ *gs bs ps sps data. is-Groebner-basis (fst ' set gs) \longrightarrow sps \neq [] \longrightarrow set*
sps \subseteq set ps \longrightarrow
 $\text{unique-idx (gs @ bs) data} \longrightarrow$
 $\text{fst ' (set (fst (compl gs bs (ps -- sps) sps data)))} \subseteq \text{pmdl (args-to-set$
 $(gs, bs, ps))})$

lemma *compl-pmdlI*:
assumes \bigwedge *gs bs ps sps data. is-Groebner-basis (fst ' set gs) \implies sps \neq [] \implies set*
sps \subseteq set ps \implies
 $\text{unique-idx (gs @ bs) data} \implies$
 $\text{fst ' (set (fst (compl gs bs (ps -- sps) sps data)))} \subseteq \text{pmdl (args-to-set}$
 $(gs, bs, ps))$
shows *compl-pmdl compl*
 \langle *proof* \rangle

lemma *compl-pmdlD*:
assumes *compl-pmdl compl and is-Groebner-basis (fst ' set gs)*
and *sps \neq [] and set sps \subseteq set ps and unique-idx (gs @ bs) data*
shows $\text{fst ' (set (fst (compl gs bs (ps -- sps) sps data)))} \subseteq \text{pmdl (args-to-set}$
 $(gs, bs, ps))$
 \langle *proof* \rangle

definition *compl-conn* :: (*t, 'b::field, 'c, 'd*) *complT* \Rightarrow *bool*
where *compl-conn compl* \longleftrightarrow
 $(\forall$ *d m gs bs ps sps p q data. dickson-grading d \longrightarrow fst ' set gs \subseteq dgrad-p-set*
 $d m \longrightarrow$
 $\text{is-Groebner-basis (fst ' set gs) \longrightarrow fst ' set bs \subseteq dgrad-p-set d m $\longrightarrow$$
 $\text{set ps} \subseteq \text{set bs} \times (\text{set gs} \cup \text{set bs}) \longrightarrow \text{sps} \neq [] \longrightarrow \text{set sps} \subseteq \text{set ps} \longrightarrow$
 $\text{unique-idx (gs @ bs) data} \longrightarrow (p, q) \in \text{set sps} \longrightarrow \text{fst p} \neq 0 \longrightarrow \text{fst q}$
 $\neq 0 \longrightarrow$
 $\text{crit-pair-cbelow-on d m (fst ' (set gs} \cup \text{set bs)} \cup \text{fst ' set (fst (compl gs}$

$bs (ps \text{ --- } sps) sps data))) (fst p) (fst q)$

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

lemma *compl-connI*:

assumes $\bigwedge d m gs bs ps sps p q data. dickson\text{-}grading\ d \implies fst \text{ ' } set\ gs \subseteq dgrad\text{-}p\text{-}set\ d\ m \implies$

$is\text{-}Groebner\text{-}basis\ (fst \text{ ' } set\ gs) \implies fst \text{ ' } set\ bs \subseteq dgrad\text{-}p\text{-}set\ d\ m \implies$

$set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \implies sps \neq [] \implies set\ sps \subseteq set\ ps \implies$

$unique\text{-}idx\ (gs @ bs) data \implies (p, q) \in set\ sps \implies fst\ p \neq 0 \implies fst\ q \neq$

$0 \implies$

$crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst \text{ ' } (set\ gs \cup set\ bs) \cup fst \text{ ' } set\ (fst\ (compl\ gs\ bs\ (ps \text{ --- } sps)\ sps\ data)))) (fst\ p)\ (fst\ q)$

shows *compl-conn compl*

<proof>

lemma *compl-connD*:

assumes *compl-conn compl* **and** *dickson-grading d* **and** $fst \text{ ' } set\ gs \subseteq dgrad\text{-}p\text{-}set\ d\ m$

and *is-Groebner-basis (fst ' set gs)* **and** $fst \text{ ' } set\ bs \subseteq dgrad\text{-}p\text{-}set\ d\ m$

and $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs)$ **and** $sps \neq []$ **and** $set\ sps \subseteq set\ ps$

and *unique-idx (gs @ bs) data* **and** $(p, q) \in set\ sps$ **and** $fst\ p \neq 0$ **and** $fst\ q \neq$

0

shows *crit-pair-cbelow-on d m (fst ' (set gs ∪ set bs) ∪ fst ' set (fst (compl gs bs (ps --- sps) sps data))) (fst p) (fst q)*

<proof>

6.2.9 Function *gb-schema-dummy*

definition (**in** $-$) *add-indices* :: $((a, b, c) pdata\ list \times d) \Rightarrow (nat \times d) \Rightarrow ((a, b, c) pdata\ list \times nat \times d)$

where *add-indices ns data* =

$(map\text{-}idx\ (\lambda h\ i. (fst\ h, i, snd\ h))\ (fst\ ns)\ (fst\ data),\ fst\ data + length\ (fst\ ns),\ snd\ ns)$

lemma (**in** $-$) *add-indices-code* [code]:

$add\text{-}indices\ (ns,\ data)\ (n,\ data') = (map\text{-}idx\ (\lambda(h, d)\ i. (h, i, d))\ ns\ n,\ n + length\ ns,\ data)$

<proof>

lemma *fst-add-indices*: $map\ fst\ (fst\ (add\text{-}indices\ ns\ data')) = map\ fst\ (fst\ ns)$

<proof>

corollary *fst-set-add-indices*: $fst \text{ ' } set\ (fst\ (add\text{-}indices\ ns\ data')) = fst \text{ ' } set\ (fst\ ns)$

<proof>

lemma *in-set-add-indicesE*:

assumes $f \in \text{set } (\text{fst } (\text{add-indices } \text{aux } \text{data}))$
obtains i **where** $i < \text{length } (\text{fst } \text{aux})$ **and** $f = (\text{fst } ((\text{fst } \text{aux}) ! i), \text{fst } \text{data} + i,$
 $\text{snd } ((\text{fst } \text{aux}) ! i))$
 $\langle \text{proof} \rangle$

definition *gb-schema-aux-term1* :: $((('t, 'b::\text{field}, 'c) \text{pdata list} \times ('t, 'b, 'c) \text{pdata-pair list}) \times$

$((('t, 'b, 'c) \text{pdata list} \times ('t, 'b, 'c) \text{pdata-pair list})) \text{set}$
where $\text{gb-schema-aux-term1} = \{(a, b::('t, 'b, 'c) \text{pdata list}). (\text{fst } \text{'set } a) \sqsupseteq p (\text{fst}$
 $\text{'set } b)\}$ $\langle *lex* \rangle$
 $(\text{measure } (\text{card } \circ \text{set}))$

definition *gb-schema-aux-term2* ::

$(\text{'a} \Rightarrow \text{nat}) \Rightarrow ('t, 'b::\text{field}, 'c) \text{pdata list} \Rightarrow (((('t, 'b, 'c) \text{pdata list} \times ('t, 'b, 'c)$
 $\text{pdata-pair list}) \times$
 $((('t, 'b, 'c) \text{pdata list} \times ('t, 'b, 'c) \text{pdata-pair list})) \text{set}$
where $\text{gb-schema-aux-term2 } d \text{ gs} = \{(a, b). \text{dgrad-p-set-le } d (\text{args-to-set } (\text{gs}, a))$
 $(\text{args-to-set } (\text{gs}, b)) \wedge$
 $\text{component-of-term } \text{'Keys } (\text{args-to-set } (\text{gs}, a)) \subseteq \text{component-of-term}$
 $\text{'Keys } (\text{args-to-set } (\text{gs}, b))\}$

definition *gb-schema-aux-term* **where** $\text{gb-schema-aux-term } d \text{ gs} = \text{gb-schema-aux-term1}$
 $\cap \text{gb-schema-aux-term2 } d \text{ gs}$

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

lemma *gb-schema-aux-term1-wf-on*:

assumes *dickson-grading* d **and** *finite* K
shows *wfp-on* $(\lambda x y. (x, y) \in \text{gb-schema-aux-term1})$
 $\{x::((('t, 'b, 'c) \text{pdata list}) \times (((('t, 'b::\text{field}, 'c) \text{pdata-pair list})).$
 $\text{args-to-set } (\text{gs}, x) \subseteq \text{dgrad-p-set } d \text{ m} \wedge \text{component-of-term } \text{'Keys}$
 $(\text{args-to-set } (\text{gs}, x)) \subseteq K\}$
 $\langle \text{proof} \rangle$

lemma *gb-schema-aux-term-wf*:

assumes *dickson-grading* d
shows *wf* $(\text{gb-schema-aux-term } d \text{ gs})$
 $\langle \text{proof} \rangle$

lemma *dgrad-p-set-le-args-to-set-ab*:

assumes *dickson-grading* d **and** *ap-spec* ap **and** *ab-spec* ab **and** *compl-struct*
 compl
assumes $\text{sps} \neq []$ **and** $\text{set } \text{sps} \subseteq \text{set } \text{ps}$ **and** $\text{hs} = \text{fst } (\text{add-indices } (\text{compl } \text{gs } \text{bs}$
 $(\text{ps} \text{ --- } \text{sps}) \text{ sps } \text{data}) \text{ data})$
shows $\text{dgrad-p-set-le } d (\text{args-to-set } (\text{gs}, \text{ab } \text{gs } \text{bs } \text{hs } \text{data}', \text{ap } \text{gs } \text{bs } (\text{ps} \text{ --- } \text{sps})$
 $\text{hs } \text{data}')) (\text{args-to-set } (\text{gs}, \text{bs}, \text{ps}))$
 $(\text{is } \text{dgrad-p-set-le } - \text{?l } \text{?r})$
 $\langle \text{proof} \rangle$

corollary *dgrad-p-set-le-args-to-set-struct*:

assumes *dickson-grading* d **and** *struct-spec* $sel\ ap\ ab\ compl$ **and** $ps \neq []$
assumes $sps = sel\ gs\ bs\ ps\ data$ **and** $hs = fst\ (add_indices\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)\ data)$
shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ (args\text{-}to\text{-}set\ (gs,\ ab\ gs\ bs\ hs\ data',\ ap\ gs\ bs\ (ps\ \text{---}\ sps)\ hs\ data'))\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$
 $\langle proof \rangle$

lemma *components-subset-ab*:

assumes *ap-spec* ap **and** *ab-spec* ab **and** *compl-struct* $compl$
assumes $sps \neq []$ **and** $set\ sps \subseteq set\ ps$ **and** $hs = fst\ (add_indices\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)\ data)$
shows $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ ab\ gs\ bs\ hs\ data',\ ap\ gs\ bs\ (ps\ \text{---}\ sps)\ hs\ data')) \subseteq$
 $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$ **(is** $?l \subseteq ?r$ **)**
 $\langle proof \rangle$

corollary *components-subset-struct*:

assumes *struct-spec* $sel\ ap\ ab\ compl$ **and** $ps \neq []$
assumes $sps = sel\ gs\ bs\ ps\ data$ **and** $hs = fst\ (add_indices\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)\ data)$
shows $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ ab\ gs\ bs\ hs\ data',\ ap\ gs\ bs\ (ps\ \text{---}\ sps)\ hs\ data')) \subseteq$
 $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$
 $\langle proof \rangle$

corollary *components-struct*:

assumes *struct-spec* $sel\ ap\ ab\ compl$ **and** $ps \neq []$ **and** $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs)$
assumes $sps = sel\ gs\ bs\ ps\ data$ **and** $hs = fst\ (add_indices\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)\ data)$
shows $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ ab\ gs\ bs\ hs\ data',\ ap\ gs\ bs\ (ps\ \text{---}\ sps)\ hs\ data')) =$
 $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$ **(is** $?l = ?r$ **)**
 $\langle proof \rangle$

lemma *struct-spec-red-supset*:

assumes *struct-spec* $sel\ ap\ ab\ compl$ **and** $ps \neq []$ **and** $sps = sel\ gs\ bs\ ps\ data$
and $hs = fst\ (add_indices\ (compl\ gs\ bs\ (ps\ \text{---}\ sps)\ sps\ data)\ data)$ **and** $hs \neq []$
shows $(fst\ 'set\ (ab\ gs\ bs\ hs\ data')) \supseteq (fst\ 'set\ bs)$
 $\langle proof \rangle$

lemma *unique-idx-append*:

assumes *unique-idx* $gs\ data$ **and** $(hs,\ data') = add_indices\ aux\ data$
shows $unique\text{-}idx\ (gs\ @\ hs)\ data'$
 $\langle proof \rangle$

corollary *unique-idx-ab*:

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

lemma *rem-comps-spec-struct*:

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

lemma *pmdl-struct*:

assumes $struct\text{-spec } sel \text{ } ap \text{ } ab \text{ } compl$ **and** $compl\text{-pmdl } compl$ **and** $is\text{-Groebner-basis } (fst \text{ } ' \text{ } set \text{ } gs)$
and $ps \neq \square$ **and** $set \text{ } ps \subseteq (set \text{ } bs) \times (set \text{ } gs \cup set \text{ } bs)$ **and** $unique\text{-idx } (gs @ bs) \text{ } (snd \text{ } data)$
and $sps = sel \text{ } gs \text{ } bs \text{ } ps \text{ } (snd \text{ } data)$ **and** $aux = compl \text{ } gs \text{ } bs \text{ } (ps \text{ } \text{---} \text{ } sps) \text{ } sps \text{ } (snd \text{ } data)$
and $(hs, data') = add\text{-indices } aux \text{ } (snd \text{ } data)$
shows $pmdl \text{ } (fst \text{ } ' \text{ } set \text{ } (gs @ ab \text{ } gs \text{ } bs \text{ } hs \text{ } data')) = pmdl \text{ } (fst \text{ } ' \text{ } set \text{ } (gs @ bs))$
 $\langle proof \rangle$

lemma *discarded-subset*:

assumes $ab\text{-spec } ab$
and $D' = D \cup (set \text{ } hs \times (set \text{ } gs \cup set \text{ } bs \cup set \text{ } hs) \cup (set \text{ } ps \text{ } - \text{ } set \text{ } sps) \text{ } \text{--}_p \text{ } set \text{ } (ap \text{ } gs \text{ } bs \text{ } (ps \text{ } \text{---} \text{ } sps) \text{ } hs \text{ } data'))$
and $set \text{ } ps \subseteq set \text{ } bs \times (set \text{ } gs \cup set \text{ } bs)$ **and** $D \subseteq (set \text{ } gs \cup set \text{ } bs) \times (set \text{ } gs \cup set \text{ } bs)$
shows $D' \subseteq (set \text{ } gs \cup set \text{ } (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data')) \times (set \text{ } gs \cup set \text{ } (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data'))$
 $\langle proof \rangle$

lemma *compl-struct-disjoint*:

assumes $compl\text{-struct } compl$ **and** $sps \neq \square$ **and** $set \text{ } sps \subseteq set \text{ } ps$
shows $fst \text{ } ' \text{ } set \text{ } (fst \text{ } (compl \text{ } gs \text{ } bs \text{ } (ps \text{ } \text{---} \text{ } sps) \text{ } sps \text{ } data)) \cap fst \text{ } ' \text{ } (set \text{ } gs \cup set \text{ } bs) = \{\}$
 $\langle proof \rangle$

context

fixes $sel::('t, 'b::field, 'c::default, 'd) \text{ } selT$ **and** $ap::('t, 'b, 'c, 'd) \text{ } apT$
and $ab::('t, 'b, 'c, 'd) \text{ } abT$ **and** $compl::('t, 'b, 'c, 'd) \text{ } complT$
and $gs::('t, 'b, 'c) \text{ } pdata \text{ } list$

begin

function $(domintros) \text{ } gb\text{-schema-dummy} :: nat \times nat \times 'd \Rightarrow ('t, 'b, 'c) \text{ } pdata\text{-pair}$

$(D \cup ((\text{set } hs \times (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) \cup (\text{set } ps - \text{set } sps))) -_p$
 $\text{set } (ap \text{ } gs \text{ } bs \text{ } ps0 \text{ } hs \text{ } data')$
 $(ab \text{ } gs \text{ } bs \text{ } hs \text{ } data') (ap \text{ } gs \text{ } bs \text{ } ps0 \text{ } hs \text{ } data')$
 $)$
 $)$
 $\langle \text{proof} \rangle$

lemma *gb-schema-dummy-induct* [consumes 1, case-names base rec1 rec2]:

assumes *struct-spec sel ap ab compl*
assumes *base*: $\bigwedge bs \text{ } data \text{ } D. P \text{ } data \text{ } D \text{ } bs \text{ } [] (gs \text{ } @ \text{ } bs, D)$
and *rec1*: $\bigwedge bs \text{ } ps \text{ } sps \text{ } data \text{ } D. ps \neq [] \implies sps = \text{sel } gs \text{ } bs \text{ } ps (snd \text{ } data) \implies$
 $\text{fst } (data) \leq \text{count-const-lt-components } (\text{fst } (compl \text{ } gs \text{ } bs (ps \text{ } -- \text{ } sps))) \implies$
 $P \text{ } data \text{ } D \text{ } bs \text{ } ps (full\text{-}gb (gs \text{ } @ \text{ } bs), D)$
and *rec2*: $\bigwedge bs \text{ } ps \text{ } sps \text{ } aux \text{ } hs \text{ } rc \text{ } data \text{ } data' \text{ } D \text{ } D'. ps \neq [] \implies sps = \text{sel } gs \text{ } bs \text{ } ps$
 $(snd \text{ } data) \implies$
 $aux = \text{compl } gs \text{ } bs (ps \text{ } -- \text{ } sps) sps (snd \text{ } data) \implies (hs, data') =$
 $\text{add-indices } aux (snd \text{ } data) \implies$
 $rc = \text{fst } data - \text{count-const-lt-components } (\text{fst } aux) \implies 0 < rc \implies$
 $D' = (D \cup ((\text{set } hs \times (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) \cup (\text{set } ps - \text{set } sps)))$
 $-_p \text{ set } (ap \text{ } gs \text{ } bs (ps \text{ } -- \text{ } sps) \text{ } hs \text{ } data')) \implies$
 $P (rc, data') D' (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data') (ap \text{ } gs \text{ } bs (ps \text{ } -- \text{ } sps) \text{ } hs \text{ } data')$
 $(gb\text{-}schema\text{-}dummy (rc, data') D' (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data') (ap \text{ } gs \text{ } bs (ps$
 $-- \text{ } sps) \text{ } hs \text{ } data')) \implies$
 $P \text{ } data \text{ } D \text{ } bs \text{ } ps (gb\text{-}schema\text{-}dummy (rc, data') D' (ab \text{ } gs \text{ } bs \text{ } hs \text{ } data')$
 $(ap \text{ } gs \text{ } bs (ps \text{ } -- \text{ } sps) \text{ } hs \text{ } data'))$
shows $P \text{ } data \text{ } D \text{ } bs \text{ } ps (gb\text{-}schema\text{-}dummy \text{ } data \text{ } D \text{ } bs \text{ } ps)$
 $\langle \text{proof} \rangle$

lemma *fst-gb-schema-dummy-dgrad-p-set-le*:

assumes *dickson-grading d and struct-spec sel ap ab compl*
shows *dgrad-p-set-le d* ($\text{fst } ' \text{set } (\text{fst } (gb\text{-}schema\text{-}dummy \text{ } data \text{ } D \text{ } bs \text{ } ps)) (args\text{-}to\text{-}set$
 $(gs, bs, ps))$)
 $\langle \text{proof} \rangle$

lemma *fst-gb-schema-dummy-components*:

assumes *struct-spec sel ap ab compl and set ps* $\subseteq (\text{set } bs) \times (\text{set } gs \cup \text{set } bs)$
shows *component-of-term 'Keys* ($\text{fst } ' \text{set } (\text{fst } (gb\text{-}schema\text{-}dummy \text{ } data \text{ } D \text{ } bs \text{ } ps))$)
 $=$
 $\text{component-of-term } ' \text{Keys } (args\text{-}to\text{-}set (gs, bs, ps))$
 $\langle \text{proof} \rangle$

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

assumes *struct-spec sel ap ab compl and compl-pmdl compl and is-Groebner-basis*
 $(\text{fst } ' \text{set } gs)$
and $\text{set } ps \subseteq \text{set } bs \times (\text{set } gs \cup \text{set } bs)$ **and** *unique-idx* ($gs \text{ } @ \text{ } bs$) ($snd \text{ } data$)
and *rem-comps-spec* ($gs \text{ } @ \text{ } bs$) $data$
shows $pmdl (\text{fst } ' \text{set } (\text{fst } (gb\text{-}schema\text{-}dummy \text{ } data \text{ } D \text{ } bs \text{ } ps))) = pmdl (\text{fst } ' \text{set}$
 $(gs \text{ } @ \text{ } bs))$

<proof>

lemma *snd-gb-schema-dummy-subset:*

assumes *struct-spec sel ap ab compl* **and** *set ps* \subseteq *set bs* \times (*set gs* \cup *set bs*)
and *D* \subseteq (*set gs* \cup *set bs*) \times (*set gs* \cup *set bs*) **and** *res* = *gb-schema-dummy*
data D bs ps
shows *snd res* \subseteq *set (fst res)* \times *set (fst res)* \vee (\exists *xs*. *fst (res)* = *full-gb xs*)
<proof>

lemma *gb-schema-dummy-connectible1:*

assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *dickson-grading*
d
and *fst ' set gs* \subseteq *dgrad-p-set d m* **and** *is-Groebner-basis (fst ' set gs)*
and *fst ' set bs* \subseteq *dgrad-p-set d m*
and *set ps* \subseteq *set bs* \times (*set gs* \cup *set bs*)
and *unique-idx (gs @ bs) (snd data)*
and $\bigwedge p q$. *processed (p, q) (gs @ bs) ps* \implies (*p, q*) $\notin_p D \implies$ *fst p* $\neq 0 \implies$ *fst*
q $\neq 0 \implies$
crit-pair-cbelow-on d m (fst ' (set gs \cup set bs)) (fst p) (fst q)
and $\neg(\exists$ *xs*. *fst (gb-schema-dummy data D bs ps)* = *full-gb xs*)
assumes *f* \in *set (fst (gb-schema-dummy data D bs ps))*
and *g* \in *set (fst (gb-schema-dummy data D bs ps))*
and (*f, g*) \notin_p *snd (gb-schema-dummy data D bs ps)*
and *fst f* $\neq 0$ **and** *fst g* $\neq 0$
shows *crit-pair-cbelow-on d m (fst ' set (fst (gb-schema-dummy data D bs ps)))*
(fst f) (fst g)
<proof>

lemma *gb-schema-dummy-connectible2:*

assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *dickson-grading*
d
and *fst ' set gs* \subseteq *dgrad-p-set d m* **and** *is-Groebner-basis (fst ' set gs)*
and *fst ' set bs* \subseteq *dgrad-p-set d m*
and *set ps* \subseteq *set bs* \times (*set gs* \cup *set bs*) **and** *D* \subseteq (*set gs* \cup *set bs*) \times (*set gs* \cup
set bs)
and *set ps* $\cap_p D = \{\}$ **and** *unique-idx (gs @ bs) (snd data)*
and $\bigwedge B a b$. *set gs* \cup *set bs* $\subseteq B \implies$ *fst ' B* \subseteq *dgrad-p-set d m* \implies (*a, b*) \in_p
D \implies
fst a $\neq 0 \implies$ *fst b* $\neq 0 \implies$
 $(\bigwedge x y$. *x* \in *set gs* \cup *set bs* \implies *y* \in *set gs* \cup *set bs* \implies \neg (*x, y*) $\in_p D \implies$
fst x $\neq 0 \implies$ *fst y* $\neq 0 \implies$ *crit-pair-cbelow-on d m (fst ' B) (fst x)*
(fst y)) \implies
crit-pair-cbelow-on d m (fst ' B) (fst a) (fst b)
and $\bigwedge x y$. *x* \in *set (fst (gb-schema-dummy data D bs ps))* \implies *y* \in *set (fst*
(gb-schema-dummy data D bs ps)) \implies
(x, y) \notin_p *snd (gb-schema-dummy data D bs ps)* \implies *fst x* $\neq 0 \implies$ *fst y*
 $\neq 0 \implies$
crit-pair-cbelow-on d m (fst ' set (fst (gb-schema-dummy data D bs ps)))
(fst x) (fst y)

and $\neg(\exists xs. \text{fst } (gb\text{-schema-dummy data } D \text{ bs ps}) = \text{full-gb } xs)$
assumes $(f, g) \in_p \text{snd } (gb\text{-schema-dummy data } D \text{ bs ps})$
and $\text{fst } f \neq 0$ **and** $\text{fst } g \neq 0$
shows $\text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' \text{ set } (\text{fst } (gb\text{-schema-dummy data } D \text{ bs ps})))$
 $(\text{fst } f) (\text{fst } g)$
 $\langle \text{proof} \rangle$

corollary *gb-schema-dummy-connectible:*

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

lemma *fst-gb-schema-dummy-dgrad-p-set-le-init:*

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

corollary *fst-gb-schema-dummy-dgrad-p-set-init:*

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

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

fixes $bs \text{ data}$
defines $bs0 \equiv ab \text{ gs } [] \text{ bs } \text{ data}$

```

defines  $ps0 \equiv ap\ gs \ [] \ []\ bs\ data$ 
assumes  $struct-spec\ sel\ ap\ ab\ compl$ 
shows  $component-of-term\ 'Keys\ (fst\ 'set\ (fst\ (gb-schema-dummy\ (rc,\ data)\ D\ bs0\ ps0))) =$ 
 $component-of-term\ 'Keys\ (fst\ 'set\ (gs\ @\ bs))\ (is\ ?l = ?r)$ 
<proof>

```

```

lemma  $fst-gb-schema-dummy-pmdl-init:$ 
fixes  $bs\ data$ 
defines  $bs0 \equiv ab\ gs \ [] \ []\ bs\ data$ 
defines  $ps0 \equiv ap\ gs \ [] \ []\ bs\ data$ 
assumes  $struct-spec\ sel\ ap\ ab\ compl$  and  $compl-pmdl\ compl$  and  $is-Groebner-basis$ 
 $(fst\ 'set\ gs)$ 
and  $unique-idx\ (gs\ @\ bs0)\ data$  and  $rem-comps-spec\ (gs\ @\ bs0)\ (rc,\ data)$ 
shows  $pmdl\ (fst\ 'set\ (fst\ (gb-schema-dummy\ (rc,\ data)\ D\ bs0\ ps0))) =$ 
 $pmdl\ (fst\ '(set\ (gs\ @\ bs)))\ (is\ ?l = ?r)$ 
<proof>

```

```

lemma  $fst-gb-schema-dummy-isGB-init:$ 
fixes  $bs\ data$ 
defines  $bs0 \equiv ab\ gs \ [] \ []\ bs\ data$ 
defines  $ps0 \equiv ap\ gs \ [] \ []\ bs\ data$ 
defines  $D0 \equiv set\ bs \times (set\ gs \cup set\ bs) -_p\ set\ ps0$ 
assumes  $struct-spec\ sel\ ap\ ab\ compl$  and  $compl-conn\ compl$  and  $is-Groebner-basis$ 
 $(fst\ 'set\ gs)$ 
and  $unique-idx\ (gs\ @\ bs0)\ data$  and  $rem-comps-spec\ (gs\ @\ bs0)\ (rc,\ data)$ 
shows  $is-Groebner-basis\ (fst\ 'set\ (fst\ (gb-schema-dummy\ (rc,\ data)\ D0\ bs0\ ps0)))$ 
<proof>

```

6.2.10 Function $gb-schema-aux$

```

function  $(domintros)\ gb-schema-aux :: nat \times nat \times 'd \Rightarrow ('t,\ 'b,\ 'c)\ pdata\ list \Rightarrow$ 
 $('t,\ 'b,\ 'c)\ pdata-pair\ list \Rightarrow ('t,\ 'b,\ 'c)\ pdata\ list$ 
where
 $gb-schema-aux\ data\ bs\ ps =$ 
 $(if\ ps = []\ then$ 
 $gs\ @\ bs$ 
 $else$ 
 $(let\ sps = sel\ gs\ bs\ ps\ (snd\ data); ps0 = ps -_ sps; aux = compl\ gs\ bs$ 
 $ps0\ sps\ (snd\ data);$ 
 $remcomps = fst\ (data) - count-const-lt-components\ (fst\ aux)\ in$ 
 $(if\ remcomps = 0\ then$ 
 $full-gb\ (gs\ @\ bs)$ 
 $else$ 
 $let\ (hs,\ data') = add-indices\ aux\ (snd\ data)\ in$ 
 $gb-schema-aux\ (remcomps,\ data')\ (ab\ gs\ bs\ hs\ data')\ (ap\ gs\ bs\ ps0\ hs$ 
 $data')$ 
 $)$ 
 $)$ 

```

)
 ⟨proof⟩

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

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

lemma *gb-schema-aux-domI2*:

assumes *struct-spec sel ap ab compl*

shows *gb-schema-aux-dom* (*data*, *args*)

⟨proof⟩

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

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

lemma *gb-schema-aux-induct* [*consumes 1*, *case-names base rec1 rec2*]:

assumes *struct-spec sel ap ab compl*

assumes *base*: $\bigwedge bs \ data. P \ data \ bs \ [] \ (gs @ bs)$

and *rec1*: $\bigwedge bs \ ps \ sps \ data. ps \neq [] \implies sps = sel \ gs \ bs \ ps \ (snd \ data) \implies$
 $fst \ (data) \leq count-const-lt-components \ (fst \ (compl \ gs \ bs \ (ps \ -- \ sps) \ sps \ (snd \ data))) \implies$

$P \ data \ bs \ ps \ (full-gb \ (gs @ bs))$

and *rec2*: $\bigwedge bs \ ps \ sps \ aux \ hs \ rc \ data \ data'. ps \neq [] \implies sps = sel \ gs \ bs \ ps \ (snd \ data) \implies$

$aux = compl \ gs \ bs \ (ps \ -- \ sps) \ sps \ (snd \ data) \implies (hs, data') = add-indices \ aux \ (snd \ data) \implies$

$rc = fst \ data - count-const-lt-components \ (fst \ aux) \implies 0 < rc \implies$

$P \ (rc, data') \ (ab \ gs \ bs \ hs \ data') \ (ap \ gs \ bs \ (ps \ -- \ sps) \ hs \ data')$

$(gb-schema-aux \ (rc, data') \ (ab \ gs \ bs \ hs \ data') \ (ap \ gs \ bs \ (ps \ -- \ sps) \ hs \ data')) \implies$

$P \ data \ bs \ ps \ (gb-schema-aux \ (rc, data') \ (ab \ gs \ bs \ hs \ data') \ (ap \ gs \ bs \ (ps \ -- \ sps) \ hs \ data'))$

shows $P \ data \ bs \ ps \ (gb-schema-aux \ data \ bs \ ps)$

⟨proof⟩

lemma *gb-schema-dummy-eq-gb-schema-aux*:

assumes *struct-spec sel ap ab compl*

shows $fst (gb\text{-}schema\text{-}dummy\ data\ D\ bs\ ps) = gb\text{-}schema\text{-}aux\ data\ bs\ ps$

<proof>

corollary *gb-schema-aux-dgrad-p-set-le*:

assumes *dickson-grading d and struct-spec sel ap ab compl*

shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ (fst\ 'set\ (gb\text{-}schema\text{-}aux\ data\ bs\ ps))\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$

<proof>

corollary *gb-schema-aux-components*:

assumes *struct-spec sel ap ab compl and set ps \subseteq set bs \times (set gs \cup set bs)*

shows $component\text{-}of\text{-}term\ 'Keys\ (fst\ 'set\ (gb\text{-}schema\text{-}aux\ data\ bs\ ps)) = component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ ps))$

<proof>

lemma *gb-schema-aux-pmdl*:

assumes *struct-spec sel ap ab compl and compl-pmdl compl and is-Groebner-basis (fst 'set gs)*

and *set ps \subseteq set bs \times (set gs \cup set bs) and unique-idx (gs @ bs) (snd data)*

and *rem-comps-spec (gs @ bs) data*

shows $pmdl\ (fst\ 'set\ (gb\text{-}schema\text{-}aux\ data\ bs\ ps)) = pmdl\ (fst\ 'set\ (gs\ @\ bs))$

<proof>

corollary *gb-schema-aux-dgrad-p-set-le-init*:

assumes *dickson-grading d and struct-spec sel ap ab compl*

shows $dgrad\text{-}p\text{-}set\text{-}le\ d\ (fst\ 'set\ (gb\text{-}schema\text{-}aux\ data\ (ab\ gs\ []\ bs\ (snd\ data)))\ (ap\ gs\ []\ []\ bs\ (snd\ data)))$

$(fst\ '(set\ gs\ \cup\ set\ bs))$

<proof>

corollary *gb-schema-aux-dgrad-p-set-init*:

assumes *dickson-grading d and struct-spec sel ap ab compl*

and *fst '(set gs \cup set bs) \subseteq dgrad-p-set d m*

shows $fst\ 'set\ (gb\text{-}schema\text{-}aux\ (rc,\ data)\ (ab\ gs\ []\ bs\ data)\ (ap\ gs\ []\ []\ bs\ data)) \subseteq dgrad\text{-}p\text{-}set\ d\ m$

<proof>

corollary *gb-schema-aux-components-init*:

assumes *struct-spec sel ap ab compl*

shows $component\text{-}of\text{-}term\ 'Keys\ (fst\ 'set\ (gb\text{-}schema\text{-}aux\ (rc,\ data)\ (ab\ gs\ []\ bs\ data)\ (ap\ gs\ []\ []\ bs\ data))) =$

$component\text{-}of\text{-}term\ 'Keys\ (fst\ 'set\ (gs\ @\ bs))$

<proof>

corollary *gb-schema-aux-pmdl-init*:

assumes *struct-spec sel ap ab compl and compl-pmdl compl and is-Groebner-basis (fst 'set gs)*

and *unique-idx* (*gs* @ *ab gs* [] *bs data*) *data* **and** *rem-comps-spec* (*gs* @ *ab gs* [] *bs data*) (*rc, data*)
shows *pmdl* (*fst* ' *set* (*gb-schema-aux* (*rc, data*) (*ab gs* [] *bs data*) (*ap gs* [] [] *bs data*))) =
pmdl (*fst* ' (*set* (*gs* @ *bs*)))
 ⟨*proof*⟩

lemma *gb-schema-aux-isGB-init*:

assumes *struct-spec sel ap ab compl* **and** *compl-conn compl* **and** *is-Groebner-basis* (*fst* ' *set gs*)
and *unique-idx* (*gs* @ *ab gs* [] *bs data*) *data* **and** *rem-comps-spec* (*gs* @ *ab gs* [] *bs data*) (*rc, data*)
shows *is-Groebner-basis* (*fst* ' *set* (*gb-schema-aux* (*rc, data*) (*ab gs* [] *bs data*) (*ap gs* [] [] *bs data*)))
 ⟨*proof*⟩

end

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

definition *gb-schema-direct* :: (*t, 'b, 'c, 'd*) *selT* ⇒ (*t, 'b, 'c, 'd*) *apT* ⇒ (*t, 'b, 'c, 'd*) *abT* ⇒

(*t, 'b, 'c, 'd*) *complT* ⇒ (*t, 'b, 'c*) *pdata' list* ⇒ '*d* ⇒
 (*t, 'b::field, 'c::default*) *pdata' list*

where *gb-schema-direct sel ap ab compl bs0 data0* =
 (*let data* = (*length bs0, data0*); *bs1* = *fst* (*add-indices* (*bs0, data0*) (*0, data0*)));
bs = *ab* [] [] *bs1 data in*
map ($\lambda(f, -, d). (f, d)$)
 (*gb-schema-aux sel ap ab compl* [] (*count-rem-components bs, data*)
bs (*ap* [] [] [] *bs1 data*))
)

primrec *gb-schema-incr* :: (*t, 'b, 'c, 'd*) *selT* ⇒ (*t, 'b, 'c, 'd*) *apT* ⇒ (*t, 'b, 'c, 'd*) *abT* ⇒

(*t, 'b, 'c, 'd*) *complT* ⇒
 ((*t, 'b, 'c*) *pdata list* ⇒ (*t, 'b, 'c*) *pdata* ⇒ '*d* ⇒ '*d*) ⇒
 (*t, 'b, 'c*) *pdata' list* ⇒ '*d* ⇒ (*t, 'b::field, 'c::default*)

pdata' list

where

gb-schema-incr - - - - [] - = []
gb-schema-incr sel ap ab compl upd (*b0 # bs*) *data* =
 (*let* (*gs, n, data'*) = *add-indices* (*gb-schema-incr sel ap ab compl upd bs data, data*) (*0, data*);
b = (*fst b0, n, snd b0*); *data''* = *upd gs b data'* *in*
map ($\lambda(f, -, d). (f, d)$)
 (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b # gs*), *Suc n, data''*)
 (*ab gs* [] [*b*] (*Suc n, data''*)) (*ap gs* [] [] [*b*] (*Suc n, data''*)))

)

lemma (in -) *fst-set-drop-indices*:

fst ' $(\lambda(f, -, d). (f, d))$ ' *A* = *fst* ' *A* **for** *A*::('x × 'y × 'z) *set*
 ⟨*proof*⟩

lemma *fst-gb-schema-direct*:

fst ' *set* (*gb-schema-direct sel ap ab compl bs0 data0*) =
 (let *data* = (*length bs0*, *data0*); *bs1* = *fst* (*add-indices* (*bs0*, *data0*) (*0*, *data0*));
bs = *ab* [] [] *bs1 data* in
fst ' *set* (*gb-schema-aux sel ap ab compl* [] (*count-rem-components bs*, *data*)
bs (*ap* [] [] [] *bs1 data*))
)
 ⟨*proof*⟩

lemma *gb-schema-direct-dgrad-p-set*:

assumes *dickson-grading d* **and** *struct-spec sel ap ab compl* **and** *fst* ' *set bs* ⊆
dgrad-p-set d m
shows *fst* ' *set* (*gb-schema-direct sel ap ab compl bs data*) ⊆ *dgrad-p-set d m*
 ⟨*proof*⟩

theorem *gb-schema-direct-isGB*:

assumes *struct-spec sel ap ab compl* **and** *compl-conn compl*
shows *is-Groebner-basis* (*fst* ' *set* (*gb-schema-direct sel ap ab compl bs data*))
 ⟨*proof*⟩

theorem *gb-schema-direct-pmdl*:

assumes *struct-spec sel ap ab compl* **and** *compl-pmdl compl*
shows *pmdl* (*fst* ' *set* (*gb-schema-direct sel ap ab compl bs data*)) = *pmdl* (*fst* '
set bs)
 ⟨*proof*⟩

lemma *fst-gb-schema-incr*:

fst ' *set* (*gb-schema-incr sel ap ab compl upd* (*b0 # bs*) *data*) =
 (let (*gs*, *n*, *data'*) = *add-indices* (*gb-schema-incr sel ap ab compl upd bs data*,
data) (*0*, *data*);
b = (*fst b0*, *n*, *snd b0*); *data''* = *upd gs b data'* in
fst ' *set* (*gb-schema-aux sel ap ab compl gs* (*count-rem-components* (*b # gs*),
Suc n, *data''*)
 (*ab gs* [] [*b*] (*Suc n*, *data''*)) (*ap gs* [] [] [*b*] (*Suc n*, *data''*)))
)
 ⟨*proof*⟩

lemma *gb-schema-incr-dgrad-p-set*:

assumes *dickson-grading d* **and** *struct-spec sel ap ab compl*
and *fst* ' *set bs* ⊆ *dgrad-p-set d m*
shows *fst* ' *set* (*gb-schema-incr sel ap ab compl upd bs data*) ⊆ *dgrad-p-set d m*
 ⟨*proof*⟩

theorem *gb-schema-incr-dgrad-p-set-isGB*:
assumes *struct-spec sel ap ab compl and compl-conn compl*
shows *is-Groebner-basis (fst ' set (gb-schema-incr sel ap ab compl upd bs data))*
 \langle *proof* \rangle

theorem *gb-schema-incr-pmdl*:
assumes *struct-spec sel ap ab compl and compl-conn compl compl-pmdl compl*
shows *pmdl (fst ' set (gb-schema-incr sel ap ab compl upd bs data)) = pmdl (fst ' set bs)*
 \langle *proof* \rangle

6.3 Suitable Instances of the *add-pairs* Parameter

6.3.1 Specification of the *crit* parameters

type-synonym (in $-$) (*t, 'b, 'c, 'd*) *icritT* = $\text{nat} \times 'd \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow ('t, 'b, 'c) \text{pdata} \Rightarrow \text{bool}$

type-synonym (in $-$) (*t, 'b, 'c, 'd*) *ncritT* = $\text{nat} \times 'd \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow \text{bool} \Rightarrow (\text{bool} \times ('t, 'b, 'c) \text{pdata-pair}) \text{list} \Rightarrow ('t, 'b, 'c) \text{pdata} \Rightarrow \text{bool}$

type-synonym (in $-$) (*t, 'b, 'c, 'd*) *ocritT* = $\text{nat} \times 'd \Rightarrow ('t, 'b, 'c) \text{pdata list} \Rightarrow (\text{bool} \times ('t, 'b, 'c) \text{pdata-pair}) \text{list} \Rightarrow ('t, 'b, 'c) \text{pdata} \Rightarrow \text{bool}$

definition *icrit-spec* :: (*t, 'b::field, 'c, 'd*) *icritT* \Rightarrow *bool*
where *icrit-spec crit* \longleftrightarrow
 $(\forall d m \text{ data } gs \ bs \ hs \ p \ q. \text{dickson-grading } d \longrightarrow$
 $\text{fst ' (set } gs \cup \text{ set } bs \cup \text{ set } hs) \subseteq \text{dgrad-p-set } d \ m \longrightarrow \text{unique-idx (gs @$
 $bs \ @ \ hs) \ data} \longrightarrow$
 $\text{is-Groebner-basis (fst ' set } gs) \longrightarrow p \in \text{set } hs \longrightarrow q \in \text{set } gs \cup \text{set } bs \cup$
 $\text{set } hs \longrightarrow$
 $\text{fst } p \neq 0 \longrightarrow \text{fst } q \neq 0 \longrightarrow \text{crit data } gs \ bs \ hs \ p \ q \longrightarrow$
 $\text{crit-pair-cbelow-on } d \ m \ (\text{fst ' (set } gs \cup \text{ set } bs \cup \text{ set } hs)) \ (\text{fst } p) \ (\text{fst } q))$

Criteria satisfying *icrit-spec* can be used for discarding pairs *instantly*, without reference to any other pairs. The product criterion for scalar polynomials satisfies *icrit-spec*, and so does the component criterion (which checks whether the component-indices of the leading terms of two polynomials are identical).

definition *ncrit-spec* :: (*t, 'b::field, 'c, 'd*) *ncritT* \Rightarrow *bool*
where *ncrit-spec crit* \longleftrightarrow

$$\begin{aligned}
& (\forall d m \text{ data } gs \text{ bs } hs \text{ ps } B \text{ q-in-bs } p \text{ q. dickson-grading } d \longrightarrow \text{set } gs \cup \text{set} \\
& \text{bs} \cup \text{set } hs \subseteq B \longrightarrow \\
& \text{fst } ' B \subseteq \text{dgrad-p-set } d \text{ m} \longrightarrow \text{snd } ' \text{set } ps \subseteq \text{set } hs \times (\text{set } gs \cup \text{set } bs \\
& \cup \text{set } hs) \longrightarrow \\
& \text{unique-idx } (gs \text{ @ } bs \text{ @ } hs) \text{ data} \longrightarrow \text{is-Groebner-basis } (\text{fst } ' \text{set } gs) \longrightarrow \\
& (q\text{-in-bs} \longrightarrow (q \in \text{set } gs \cup \text{set } bs)) \longrightarrow \\
& (\forall p' q'. (p', q') \in_p \text{snd } ' \text{set } ps \longrightarrow \text{fst } p' \neq 0 \longrightarrow \text{fst } q' \neq 0 \longrightarrow \\
& \text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' B) (\text{fst } p') (\text{fst } q')) \longrightarrow \\
& (\forall p' q'. p' \in \text{set } gs \cup \text{set } bs \longrightarrow q' \in \text{set } gs \cup \text{set } bs \longrightarrow \text{fst } p' \neq 0 \longrightarrow \\
& \text{fst } q' \neq 0 \longrightarrow \\
& \text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' B) (\text{fst } p') (\text{fst } q')) \longrightarrow \\
& p \in \text{set } hs \longrightarrow q \in \text{set } gs \cup \text{set } bs \cup \text{set } hs \longrightarrow \text{fst } p \neq 0 \longrightarrow \text{fst } q \neq 0 \\
& \longrightarrow \\
& \text{crit data } gs \text{ bs } hs \text{ q-in-bs } ps \text{ p } q \longrightarrow \\
& \text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' B) (\text{fst } p) (\text{fst } q))
\end{aligned}$$

definition *ocrit-spec* :: ('t, 'b::field, 'c, 'd) *ocritT* \Rightarrow *bool*

where *ocrit-spec crit* \longleftrightarrow

$$\begin{aligned}
& (\forall d m \text{ data } hs \text{ ps } B \text{ p } q. \text{dickson-grading } d \longrightarrow \text{set } hs \subseteq B \longrightarrow \text{fst } ' B \subseteq \\
& \text{dgrad-p-set } d \text{ m} \longrightarrow \\
& \text{unique-idx } (p \# q \# hs \text{ @ } (\text{map } (\text{fst} \circ \text{snd}) \text{ ps}) \text{ @ } (\text{map } (\text{snd} \circ \text{snd}) \\
& \text{ps})) \text{ data} \longrightarrow \\
& (\forall p' q'. (p', q') \in_p \text{snd } ' \text{set } ps \longrightarrow \text{fst } p' \neq 0 \longrightarrow \text{fst } q' \neq 0 \longrightarrow \\
& \text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' B) (\text{fst } p') (\text{fst } q')) \longrightarrow \\
& p \in B \longrightarrow q \in B \longrightarrow \text{fst } p \neq 0 \longrightarrow \text{fst } q \neq 0 \longrightarrow \\
& \text{crit data } hs \text{ ps } p \text{ q} \longrightarrow \text{crit-pair-cbelow-on } d \text{ m } (\text{fst } ' B) (\text{fst } p) (\text{fst } q))
\end{aligned}$$

Criteria satisfying *ncrit-spec* can be used for discarding new pairs by reference to new and old elements, whereas criteria satisfying *ocrit-spec* can be used for discarding old pairs by reference to new elements *only* (no existing ones!). The chain criterion satisfies both *ncrit-spec* and *ocrit-spec*.

lemma *icrit-specI*:

assumes $\bigwedge d m \text{ data } gs \text{ bs } hs \text{ p } q.$

dickson-grading $d \Longrightarrow \text{fst } ' (\text{set } gs \cup \text{set } bs \cup \text{set } hs) \subseteq \text{dgrad-p-set } d \text{ m}$

\Longrightarrow

unique-idx $(gs \text{ @ } bs \text{ @ } hs) \text{ data} \Longrightarrow \text{is-Groebner-basis } (\text{fst } ' \text{set } gs) \Longrightarrow$

$p \in \text{set } hs \Longrightarrow q \in \text{set } gs \cup \text{set } bs \cup \text{set } hs \Longrightarrow \text{fst } p \neq 0 \Longrightarrow \text{fst } q \neq 0$

\Longrightarrow

crit data $gs \text{ bs } hs \text{ p } q \Longrightarrow$

crit-pair-cbelow-on $d \text{ m } (\text{fst } ' (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) (\text{fst } p) (\text{fst } q)$

shows *icrit-spec crit*

<proof>

lemma *icrit-specD*:

assumes *icrit-spec crit* **and** *dickson-grading* d

and $\text{fst } ' (\text{set } gs \cup \text{set } bs \cup \text{set } hs) \subseteq \text{dgrad-p-set } d \text{ m}$ **and** *unique-idx* $(gs \text{ @ } bs \text{ @ } hs) \text{ data}$

and *is-Groebner-basis* $(\text{fst } ' \text{set } gs)$ **and** $p \in \text{set } hs$ **and** $q \in \text{set } gs \cup \text{set } bs \cup \text{set } hs$

and $\text{fst } p \neq 0$ **and** $\text{fst } q \neq 0$ **and** $\text{crit data } gs \text{ } bs \text{ } hs \text{ } p \text{ } q$
shows $\text{crit-pair-cbelow-on } d \text{ } m \text{ } (\text{fst } ' \text{ } (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) \text{ } (\text{fst } p) \text{ } (\text{fst } q)$
<proof>

lemma *ncrit-specI*:

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

lemma *ncrit-specD*:

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

lemma *ocrit-specI*:

assumes $\bigwedge d \text{ } m \text{ } \text{data } hs \text{ } ps \text{ } B \text{ } p \text{ } q.$
 $\text{dickson-grading } d \implies \text{set } hs \subseteq B \implies \text{fst } ' \text{ } B \subseteq \text{dgrad-p-set } d \text{ } m \implies$
 $\text{unique-idx } (p \text{ } \# \text{ } q \text{ } \# \text{ } hs \text{ } @ \text{ } (\text{map } (\text{fst } \circ \text{snd}) \text{ } ps) \text{ } @ \text{ } (\text{map } (\text{snd } \circ \text{snd})$
 $ps)) \text{ } \text{data} \implies$
 $(\bigwedge p' \text{ } q'. (p', q') \in_p \text{snd } ' \text{ } \text{set } ps \implies \text{fst } p' \neq 0 \implies \text{fst } q' \neq 0 \implies$
 $\text{crit-pair-cbelow-on } d \text{ } m \text{ } (\text{fst } ' \text{ } B) \text{ } (\text{fst } p') \text{ } (\text{fst } q')) \implies$
 $p \in B \implies q \in B \implies \text{fst } p \neq 0 \implies \text{fst } q \neq 0 \implies$
 $\text{crit data } hs \text{ } ps \text{ } p \text{ } q \implies \text{crit-pair-cbelow-on } d \text{ } m \text{ } (\text{fst } ' \text{ } B) \text{ } (\text{fst } p) \text{ } (\text{fst } q)$
shows *ocrit-spec crit*

<proof>

lemma *ocrit-specD*:

assumes *ocrit-spec crit and dickson-grading d and set hs* $\subseteq B$ **and** *fst ' B* \subseteq
dgrad-p-set d m

and *unique-idx (p # q # hs @ (map (fst o snd) ps) @ (map (snd o snd) ps))*
data

and $\bigwedge p' q'. (p', q') \in_p \text{snd } ' \text{ set } ps \implies \text{fst } p' \neq 0 \implies \text{fst } q' \neq 0 \implies$
crit-pair-cbelow-on d m (fst ' B) (fst p') (fst q')

and $p \in B$ **and** $q \in B$ **and** $\text{fst } p \neq 0$ **and** $\text{fst } q \neq 0$

and *crit data hs ps p q*

shows *crit-pair-cbelow-on d m (fst ' B) (fst p) (fst q)*

<proof>

6.3.2 Suitable instances of the *crit* parameters

definition *component-crit* :: (*t, 'b::zero, 'c, 'd*) *icritT*

where *component-crit data gs bs hs p q* \longleftrightarrow (*component-of-term (lt (fst p))* \neq
component-of-term (lt (fst q)))

lemma *icrit-spec-component-crit*: *icrit-spec (component-crit::(t, 'b::field, 'c, 'd)*
icritT)

<proof>

The product criterion is only applicable to scalar polynomials.

definition *product-crit* :: (*'a, 'b::zero, 'c, 'd*) *icritT*

where *product-crit data gs bs hs p q* \longleftrightarrow (*gcs (punit.lt (fst p)) (punit.lt (fst q))*
 $= 0$)

lemma (**in** *gd-term*) *icrit-spec-product-crit*: *punit.icrit-spec (product-crit::('a, 'b::field,*
'c, 'd) icritT)

<proof>

component-crit and *product-crit* ignore the *data* parameter.

fun (**in** $-$) *pair-in-list* :: ($\text{bool} \times ('a, 'b, 'c)$ *pdata-pair*) *list* \Rightarrow $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$
where

pair-in-list [] - - = False
pair-in-list ((-, (-, i', -), (-, j', -)) # ps) i j =
 $((i = i' \wedge j = j') \vee (i = j' \wedge j = i') \vee \text{pair-in-list ps } i j)$

lemma (**in** $-$) *pair-in-listE*:

assumes *pair-in-list ps i j*

obtains $p q a b$ **where** $((p, i, a), (q, j, b)) \in_p \text{snd } ' \text{ set } ps$

<proof>

definition *chain-ncrit* :: (*t, 'b::zero, 'c, 'd*) *ncritT*

where *chain-ncrit data gs bs hs q-in-bs ps p q* \longleftrightarrow
 $(\text{let } v = \text{lt } (fst p); l = \text{term-of-pair } (lcs (pp-of-term v)) (lp (fst q)),$
component-of-term v);

$$\begin{aligned}
& i = \text{fst} (\text{snd } p); j = \text{fst} (\text{snd } q) \text{ in} \\
& (\exists r \in \text{set } gs. \text{ let } k = \text{fst} (\text{snd } r) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt} (\text{fst } r) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge (q\text{-in-bs} \vee \\
& \text{pair-in-list } ps \ j \ k) \wedge \text{fst } r \neq 0) \vee \\
& (\exists r \in \text{set } bs. \text{ let } k = \text{fst} (\text{snd } r) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt} (\text{fst } r) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge (q\text{-in-bs} \vee \\
& \text{pair-in-list } ps \ j \ k) \wedge \text{fst } r \neq 0) \vee \\
& (\exists h \in \text{set } hs. \text{ let } k = \text{fst} (\text{snd } h) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt} (\text{fst } h) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge \text{pair-in-list} \\
& \text{ps } j \ k \wedge \text{fst } h \neq 0)
\end{aligned}$$

definition *chain-ocrit* :: ('t, 'b::zero, 'c, 'd) ocritT

where *chain-ocrit data hs ps p q* \longleftrightarrow
 $(\text{let } v = \text{lt} (\text{fst } p); l = \text{term-of-pair} (\text{lcs} (\text{pp-of-term } v) (\text{lp} (\text{fst } q))),$
component-of-term v);
 $i = \text{fst} (\text{snd } p); j = \text{fst} (\text{snd } q) \text{ in}$
 $(\exists h \in \text{set } hs. \text{ let } k = \text{fst} (\text{snd } h) \text{ in}$
 $k \neq i \wedge k \neq j \wedge \text{lt} (\text{fst } h) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge \text{pair-in-list}$
 $ps \ j \ k \wedge \text{fst } h \neq 0))$

chain-ncrit and *chain-ocrit* ignore the *data* parameter.

lemma *chain-ncritE*:

assumes *chain-ncrit data gs bs hs q-in-bs ps p q* **and** *snd ' set ps* \subseteq *set hs* \times
(set gs \cup *set bs* \cup *set hs)*
and *unique-idx (gs @ bs @ hs) data* **and** $p \in \text{set } hs$ **and** $q \in \text{set } gs \cup \text{set } bs \cup$
 $\text{set } hs$
obtains *r* **where** $r \in \text{set } gs \cup \text{set } bs \cup \text{set } hs$ **and** $\text{fst } r \neq 0$ **and** $r \neq p$ **and** r
 $\neq q$
and $\text{lt} (\text{fst } r) \text{ adds}_t \text{term-of-pair} (\text{lcs} (\text{lp} (\text{fst } p)) (\text{lp} (\text{fst } q))), \text{component-of-term}$
 $(\text{lt} (\text{fst } p))$
and $(p, r) \in_p \text{snd ' set } ps$ **and** $(r \in \text{set } gs \cup \text{set } bs \wedge q\text{-in-bs}) \vee (q, r) \in_p \text{snd}$
 $\text{' set } ps$
 $\langle \text{proof} \rangle$

lemma *chain-ocritE*:

assumes *chain-ocrit data hs ps p q*
and *unique-idx (p # q # hs @ (map (fst o snd) ps) @ (map (snd o snd) ps))*
data (is unique-idx ?xs -)
obtains *h* **where** $h \in \text{set } hs$ **and** $\text{fst } h \neq 0$ **and** $h \neq p$ **and** $h \neq q$
and $\text{lt} (\text{fst } h) \text{ adds}_t \text{term-of-pair} (\text{lcs} (\text{lp} (\text{fst } p)) (\text{lp} (\text{fst } q))), \text{component-of-term}$
 $(\text{lt} (\text{fst } p))$
and $(p, h) \in_p \text{snd ' set } ps$ **and** $(q, h) \in_p \text{snd ' set } ps$
 $\langle \text{proof} \rangle$

lemma *ncrit-spec-chain-ncrit*: *ncrit-spec (chain-ncrit::('t, 'b::field, 'c, 'd) ncritT)*
 $\langle \text{proof} \rangle$

lemma *ocrit-spec-chain-ocrit*: *ocrit-spec (chain-ocrit::('t, 'b::field, 'c, 'd) ocritT)*
 $\langle \text{proof} \rangle$

lemma *icrit-spec-no-crit*: *icrit-spec* ((λ- - - - - . *False*):(‘*t*, ‘*b*::*field*, ‘*c*, ‘*d*) *icritT*)
 ⟨*proof*⟩

lemma *ncrit-spec-no-crit*: *ncrit-spec* ((λ- - - - - . *False*):(‘*t*, ‘*b*::*field*, ‘*c*, ‘*d*)
ncritT)
 ⟨*proof*⟩

lemma *ocrit-spec-no-crit*: *ocrit-spec* ((λ- - - - - . *False*):(‘*t*, ‘*b*::*field*, ‘*c*, ‘*d*) *ocritT*)
 ⟨*proof*⟩

6.3.3 Creating Initial List of New Pairs

type-synonym (in $-$) (‘*t*, ‘*b*, ‘*c*) *apsT* = *bool* \Rightarrow (‘*t*, ‘*b*, ‘*c*) *pdata list* \Rightarrow (‘*t*, ‘*b*, ‘*c*) *pdata list* \Rightarrow
 (‘*t*, ‘*b*, ‘*c*) *pdata* \Rightarrow (*bool* \times (‘*t*, ‘*b*, ‘*c*) *pdata-pair*) *list*
 \Rightarrow
 (*bool* \times (‘*t*, ‘*b*, ‘*c*) *pdata-pair*) *list*

type-synonym (in $-$) (‘*t*, ‘*b*, ‘*c*, ‘*d*) *npT* = (‘*t*, ‘*b*, ‘*c*) *pdata list* \Rightarrow (‘*t*, ‘*b*, ‘*c*)
pdata list \Rightarrow
 (‘*t*, ‘*b*, ‘*c*) *pdata list* \Rightarrow *nat* \times ‘*d* \Rightarrow
 (*bool* \times (‘*t*, ‘*b*, ‘*c*) *pdata-pair*) *list*

definition *np-spec* :: (‘*t*, ‘*b*, ‘*c*, ‘*d*) *npT* \Rightarrow *bool*
 where *np-spec np* \longleftrightarrow (\forall *gs bs hs data*.
 $\text{snd } \text{' set } (np \text{ gs bs hs data}) \subseteq \text{set hs} \times (\text{set gs} \cup \text{set bs} \cup \text{set hs}) \wedge$
 $\text{set hs} \times (\text{set gs} \cup \text{set bs}) \subseteq \text{snd } \text{' set } (np \text{ gs bs hs data}) \wedge$
 $(\forall a b. a \in \text{set hs} \longrightarrow b \in \text{set hs} \longrightarrow a \neq b \longrightarrow (a, b) \in_p \text{snd } \text{' set } (np \text{ gs bs hs data})) \wedge$
 $(\forall p q. (True, p, q) \in \text{set } (np \text{ gs bs hs data}) \longrightarrow q \in \text{set gs} \cup \text{set bs}))$

lemma *np-specI*:

assumes \bigwedge *gs bs hs data*.
 $\text{snd } \text{' set } (np \text{ gs bs hs data}) \subseteq \text{set hs} \times (\text{set gs} \cup \text{set bs} \cup \text{set hs}) \wedge$
 $\text{set hs} \times (\text{set gs} \cup \text{set bs}) \subseteq \text{snd } \text{' set } (np \text{ gs bs hs data}) \wedge$
 $(\forall a b. a \in \text{set hs} \longrightarrow b \in \text{set hs} \longrightarrow a \neq b \longrightarrow (a, b) \in_p \text{snd } \text{' set } (np \text{ gs bs hs data})) \wedge$
 $(\forall p q. (True, p, q) \in \text{set } (np \text{ gs bs hs data}) \longrightarrow q \in \text{set gs} \cup \text{set bs})$
shows *np-spec np*
 ⟨*proof*⟩

lemma *np-specD1*:

assumes *np-spec np*
shows $\text{snd } \text{' set } (np \text{ gs bs hs data}) \subseteq \text{set hs} \times (\text{set gs} \cup \text{set bs} \cup \text{set hs})$
 ⟨*proof*⟩

lemma *np-specD2*:

assumes *np-spec np*

shows $set\ hs \times (set\ gs \cup set\ bs) \subseteq snd \text{ ' } set\ (np\ gs\ bs\ hs\ data)$

<proof>

lemma *np-specD3*:

assumes *np-spec np* **and** $a \in set\ hs$ **and** $b \in set\ hs$ **and** $a \neq b$

shows $(a, b) \in_p snd \text{ ' } set\ (np\ gs\ bs\ hs\ data)$

<proof>

lemma *np-specD4*:

assumes *np-spec np* **and** $(True, p, q) \in set\ (np\ gs\ bs\ hs\ data)$

shows $q \in set\ gs \cup set\ bs$

<proof>

lemma *np-specE*:

assumes *np-spec np* **and** $p \in set\ hs$ **and** $q \in set\ gs \cup set\ bs \cup set\ hs$ **and** $p \neq q$

assumes 1: $\bigwedge q\text{-in-}bs. (q\text{-in-}bs, p, q) \in set\ (np\ gs\ bs\ hs\ data) \implies thesis$

assumes 2: $\bigwedge p\text{-in-}bs. (p\text{-in-}bs, q, p) \in set\ (np\ gs\ bs\ hs\ data) \implies thesis$

shows *thesis*

<proof>

definition *add-pairs-single-naive* :: $'d \Rightarrow ('t, 'b::zero, 'c) apsT$

where *add-pairs-single-naive data flag gs bs h ps = ps @ (map (λg. (flag, h, g)) gs) @ (map (λb. (flag, h, b)) bs)*

lemma *set-add-pairs-single-naive*:

$set\ (add\ pairs\ single\ naive\ data\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag \text{ ' } (\{h\} \times (set\ gs \cup set\ bs))$

<proof>

fun *add-pairs-single-sorted* :: $((bool \times ('t, 'b, 'c) pdata\ pair) \Rightarrow (bool \times ('t, 'b, 'c) pdata\ pair)) \Rightarrow bool \Rightarrow$

$(('t, 'b::zero, 'c) apsT \text{ where}$

add-pairs-single-sorted - - $\square \square - ps = ps|$

add-pairs-single-sorted rel flag $\square \square (b \# bs) h ps =$

add-pairs-single-sorted rel flag $\square \square bs h (insort\ wrt\ rel\ (flag, h, b) ps)|$

add-pairs-single-sorted rel flag $(g \# gs) bs h ps =$

add-pairs-single-sorted rel flag gs bs h (insort\ wrt\ rel\ (flag, h, g) ps)

lemma *set-add-pairs-single-sorted*:

$set\ (add\ pairs\ single\ sorted\ rel\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag \text{ ' } (\{h\} \times (set\ gs \cup set\ bs))$

<proof>

primrec (**in** -) *pairs* :: $(('t, 'b, 'c) apsT \Rightarrow bool \Rightarrow ('t, 'b, 'c) pdata\ list \Rightarrow (bool \times ('t, 'b, 'c) pdata\ pair) list$

where

pairs - - $\square = \square|$

$\text{pairs } \text{aps } \text{flag } (x \# xs) = \text{aps } \text{flag } [] \text{ } xs \text{ } x (\text{pairs } \text{aps } \text{flag } xs)$

lemma *pairs-subset*:

assumes $\bigwedge gs \ bs \ h \ ps. \text{set } (\text{aps } \text{flag } gs \ bs \ h \ ps) = \text{set } ps \cup \text{Pair } \text{flag } ' (\{h\} \times (\text{set } gs \cup \text{set } bs))$

shows $\text{set } (\text{pairs } \text{aps } \text{flag } xs) \subseteq \text{Pair } \text{flag } ' (\text{set } xs \times \text{set } xs)$
 $\langle \text{proof} \rangle$

lemma *in-pairsI*:

assumes $\bigwedge gs \ bs \ h \ ps. \text{set } (\text{aps } \text{flag } gs \ bs \ h \ ps) = \text{set } ps \cup \text{Pair } \text{flag } ' (\{h\} \times (\text{set } gs \cup \text{set } bs))$

and $a \neq b$ **and** $a \in \text{set } xs$ **and** $b \in \text{set } xs$

shows $(\text{flag}, a, b) \in \text{set } (\text{pairs } \text{aps } \text{flag } xs) \vee (\text{flag}, b, a) \in \text{set } (\text{pairs } \text{aps } \text{flag } xs)$
 $\langle \text{proof} \rangle$

corollary *in-pairsI'*:

assumes $\bigwedge gs \ bs \ h \ ps. \text{set } (\text{aps } \text{flag } gs \ bs \ h \ ps) = \text{set } ps \cup \text{Pair } \text{flag } ' (\{h\} \times (\text{set } gs \cup \text{set } bs))$

and $a \in \text{set } xs$ **and** $b \in \text{set } xs$ **and** $a \neq b$

shows $(a, b) \in_p \text{snd } ' \text{set } (\text{pairs } \text{aps } \text{flag } xs)$
 $\langle \text{proof} \rangle$

definition *new-pairs-naive* $:: ('t, 'b::\text{zero}, 'c, 'd) \text{ npT}$

where *new-pairs-naive* $gs \ bs \ hs \ data =$

$\text{fold } (\text{add-pairs-single-naive } \text{data } \text{True } gs \ bs) \ hs (\text{pairs } (\text{add-pairs-single-naive } \text{data}) \ \text{False } hs)$

definition *new-pairs-sorted* $:: (\text{nat} \times 'd \Rightarrow (\text{bool} \times ('t, 'b, 'c) \text{pdata-pair}) \Rightarrow (\text{bool} \times ('t, 'b, 'c) \text{pdata-pair}) \Rightarrow \text{bool}) \Rightarrow$

$('t, 'b::\text{zero}, 'c, 'd) \text{ npT}$

where *new-pairs-sorted* $\text{rel } gs \ bs \ hs \ data =$

$\text{fold } (\text{add-pairs-single-sorted } (\text{rel } \text{data}) \ \text{True } gs \ bs) \ hs (\text{pairs } (\text{add-pairs-single-sorted } (\text{rel } \text{data})) \ \text{False } hs)$

lemma *set-fold-aps*:

assumes $\bigwedge gs \ bs \ h \ ps. \text{set } (\text{aps } \text{flag } gs \ bs \ h \ ps) = \text{set } ps \cup \text{Pair } \text{flag } ' (\{h\} \times (\text{set } gs \cup \text{set } bs))$

shows $\text{set } (\text{fold } (\text{aps } \text{flag } gs \ bs) \ hs \ ps) = \text{Pair } \text{flag } ' (\text{set } hs \times (\text{set } gs \cup \text{set } bs)) \cup \text{set } ps$

$\langle \text{proof} \rangle$

lemma *set-new-pairs-naive*:

$\text{set } (\text{new-pairs-naive } gs \ bs \ hs \ data) =$

$\text{Pair } \text{True } ' (\text{set } hs \times (\text{set } gs \cup \text{set } bs)) \cup \text{set } (\text{pairs } (\text{add-pairs-single-naive } \text{data}) \ \text{False } hs)$

$\langle \text{proof} \rangle$

lemma *set-new-pairs-sorted*:

$\text{set } (\text{new-pairs-sorted } \text{rel } gs \ bs \ hs \ data) =$

$\text{Pair True } \langle (\text{set } hs \times (\text{set } gs \cup \text{set } bs)) \cup \text{set } (\text{pairs } (\text{add-pairs-single-sorted } (\text{rel } \text{data}))) \text{ False } hs \rangle$
 $\langle \text{proof} \rangle$

lemma (*in* $-$) *fst-snd-Pair* [*simp*]:
shows $\text{fst} \circ \text{Pair } x = (\lambda-. x)$ **and** $\text{snd} \circ \text{Pair } x = \text{id}$
 $\langle \text{proof} \rangle$

lemma *np-spec-new-pairs-naive*: *np-spec new-pairs-naive*
 $\langle \text{proof} \rangle$

lemma *np-spec-new-pairs-sorted*: *np-spec (new-pairs-sorted rel)*
 $\langle \text{proof} \rangle$

new-pairs-naive gs bs hs *data* and *new-pairs-sorted* rel gs bs hs *data* return lists of triples $(q\text{-in-}bs, p, q)$, where *q-in-bs* indicates whether q is contained in the list gs @ bs or in the list hs . p is always contained in hs .

definition *canon-pair-order-aux* :: $(t, 'b::\text{zero}, 'c)$ *pdata-pair* \Rightarrow $(t, 'b, 'c)$ *pdata-pair* \Rightarrow *bool*
where *canon-pair-order-aux* p $q \iff$
 $(\text{lcs } (\text{lp } (\text{fst } (\text{fst } p))) (\text{lp } (\text{fst } (\text{snd } p)))) \preceq \text{lcs } (\text{lp } (\text{fst } (\text{fst } q))) (\text{lp } (\text{fst } (\text{snd } q))))$

abbreviation *canon-pair-order* $\text{data } p$ $q \equiv \text{canon-pair-order-aux } (\text{snd } p) (\text{snd } q)$

abbreviation *canon-pair-comb* $\equiv \text{merge-wrt } \text{canon-pair-order-aux}$

6.3.4 Applying Criteria to New Pairs

definition *apply-icrit* :: $(t, 'b, 'c, 'd)$ *icritT* \Rightarrow $(\text{nat} \times 'd)$ \Rightarrow $(t, 'b, 'c)$ *pdata list* \Rightarrow
 $(t, 'b, 'c)$ *pdata list* \Rightarrow $(t, 'b, 'c)$ *pdata list* \Rightarrow
 $(\text{bool} \times (t, 'b, 'c)$ *pdata-pair*) *list* \Rightarrow
 $(\text{bool} \times \text{bool} \times (t, 'b, 'c)$ *pdata-pair*) *list*

where *apply-icrit* *crit data gs bs hs ps* = $(\text{let } c = \text{crit data } gs \text{ bs } hs \text{ in map } (\lambda(q\text{-in-}bs, p, q). (c \text{ p } q, q\text{-in-}bs, p, q))) \text{ ps}$

lemma *fst-apply-icrit*:

assumes *icrit-spec crit* **and** *dickson-grading d*
and $\text{fst } \langle (\text{set } gs \cup \text{set } bs \cup \text{set } hs) \subseteq \text{dgrad-p-set } d \text{ m}$ **and** *unique-idx* $(gs$ @ bs @ $hs)$ *data*
and *is-Groebner-basis* $(\text{fst } \langle \text{set } gs)$ **and** $p \in \text{set } hs$ **and** $q \in \text{set } gs \cup \text{set } bs \cup \text{set } hs$
and $\text{fst } p \neq 0$ **and** $\text{fst } q \neq 0$ **and** $(\text{True}, q\text{-in-}bs, p, q) \in \text{set } (\text{apply-icrit } \text{crit } \text{data } gs \text{ bs } hs \text{ ps})$
shows *crit-pair-cbelow-on* $d \text{ m } (\text{fst } \langle (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) (\text{fst } p) (\text{fst } q)$
 $\langle \text{proof} \rangle$

lemma *snd-apply-icrit* [*simp*]: $\text{map } \text{snd } (\text{apply-icrit } \text{crit } \text{data } gs \text{ bs } hs \text{ ps}) = \text{ps}$

<proof>

lemma *set-snd-apply-icrit* [*simp*]: $\text{snd } \text{' set (apply-icrit crit data gs bs hs ps) = set ps}$
<proof>

definition *apply-ncrit* :: $(\text{'t}, \text{'b}, \text{'c}, \text{'d}) \text{ncritT} \Rightarrow (\text{nat} \times \text{'d}) \Rightarrow (\text{'t}, \text{'b}, \text{'c}) \text{pdata list} \Rightarrow$

$(\text{'t}, \text{'b}, \text{'c}) \text{pdata list} \Rightarrow (\text{'t}, \text{'b}, \text{'c}) \text{pdata list} \Rightarrow$
 $(\text{bool} \times \text{bool} \times (\text{'t}, \text{'b}, \text{'c}) \text{pdata-pair}) \text{list} \Rightarrow$
 $(\text{bool} \times (\text{'t}, \text{'b}, \text{'c}) \text{pdata-pair}) \text{list}$

where *apply-ncrit crit data gs bs hs ps* =
 $(\text{let } c = \text{crit data gs bs hs in}$
 $\text{rev (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs } ps' p q \text{ then } ps'$
 $\text{else } (ic, p, q) \# ps') ps \text{ []}))$

lemma *apply-ncrit-append*:

apply-ncrit crit data gs bs hs (xs @ ys) =
 $\text{rev (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge \text{crit data gs bs hs q-in-bs } ps' p q$
 $\text{then } ps' \text{ else } (ic, p, q) \# ps') ys$
 $(\text{rev (apply-ncrit crit data gs bs hs xs}))$
<proof>

lemma *fold-superset*:

set acc \subseteq
 $\text{set (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs } ps' p q \text{ then } ps' \text{ else } (ic,$
 $p, q) \# ps') ps \text{ acc})$
<proof>

lemma *apply-ncrit-superset*:

set (apply-ncrit crit data gs bs hs ps) $\subseteq \text{set (apply-ncrit crit data gs bs hs (ps @$
 $qs))$ (**is** $?l \subseteq ?r$)
<proof>

lemma *apply-ncrit-subset-aux*:

assumes $(ic, p, q) \in \text{set (fold}$
 $(\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs } ps' p q \text{ then } ps' \text{ else } (ic, p,$
 $q) \# ps') ps \text{ acc})$
shows $(ic, p, q) \in \text{set acc} \vee (\exists q\text{-in-bs}. (ic, q\text{-in-bs}, p, q) \in \text{set ps})$
<proof>

corollary *apply-ncrit-subset*:

assumes $(ic, p, q) \in \text{set (apply-ncrit crit data gs bs hs ps)}$
obtains $q\text{-in-bs}$ **where** $(ic, q\text{-in-bs}, p, q) \in \text{set ps}$
<proof>

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

lemma *not-in-apply-ncrit*:

assumes $(ic, p, q) \notin \text{set } (\text{apply-ncrit crit data } gs \ bs \ hs \ (xs \ @ \ ((ic, q\text{-in-}bs, p, q) \ # \ ys)))$
shows $\text{crit data } gs \ bs \ hs \ q\text{-in-}bs \ (\text{rev } (\text{apply-ncrit crit data } gs \ bs \ hs \ xs)) \ p \ q$
 $\langle \text{proof} \rangle$

lemma $(\text{in } -) \ \text{setE}$:

assumes $x \in \text{set } xs$
obtains $ys \ zs$ **where** $xs = ys \ @ \ (x \ # \ zs)$
 $\langle \text{proof} \rangle$

lemma *apply-ncrit-connectible*:

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

6.3.5 Applying Criteria to Old Pairs

definition *apply-ocrit* $:: ('t, 'b, 'c, 'd) \ \text{ocritT} \Rightarrow (\text{nat} \times 'd) \Rightarrow ('t, 'b, 'c) \ \text{pdata list}$
 \Rightarrow

$(\text{bool} \times ('t, 'b, 'c) \ \text{pdata-pair}) \ \text{list} \Rightarrow ('t, 'b, 'c) \ \text{pdata-pair list}$
 \Rightarrow
 $(('t, 'b, 'c) \ \text{pdata-pair list})$

where *apply-ocrit crit data hs ps' ps* $= (\text{let } c = \text{crit data } hs \ ps' \ \text{in } [(p, q) \leftarrow ps . \neg c \ p \ q])$

lemma *set-apply-ocrit*:

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

corollary *set-apply-ocrit-iff*:

$(p, q) \in \text{set } (\text{apply-ocrit crit data } hs \ ps' \ ps) \iff ((p, q) \in \text{set } ps \wedge \neg \text{crit data } hs \ ps' \ p \ q)$
 $\langle \text{proof} \rangle$

lemma *apply-ocrit-connectible*:

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

6.3.6 Creating Final List of Pairs

context

fixes *np::('t, 'b::field, 'c, 'd) npT*
and *icrit::('t, 'b, 'c, 'd) icritT*
and *ncrit::('t, 'b, 'c, 'd) ncritT*
and *ocrit::('t, 'b, 'c, 'd) ocritT*
and *comb::('t, 'b, 'c) pdata-pair list \Rightarrow ('t, 'b, 'c) pdata-pair list \Rightarrow ('t, 'b, 'c)*
pdata-pair list
begin

definition *add-pairs :: ('t, 'b, 'c, 'd) apT*

where *add-pairs gs bs ps hs data =*
 $(\text{let } ps1 = \text{apply-ncrit } \text{ncrit } \text{data } \text{gs } \text{bs } \text{hs } (\text{apply-icrit } \text{icrit } \text{data } \text{gs } \text{bs } \text{hs } (np \ \text{gs } \text{bs } \text{hs } \text{data}));$
 $ps2 = \text{apply-ocrit } \text{ocrit } \text{data } \text{hs } ps1 \ \text{ps } \text{in } \text{comb } (\text{map } \text{snd } [x \leftarrow ps1 \ . \ \neg$
 $\text{fst } x]) \ ps2)$

lemma *set-add-pairs*:

assumes $\bigwedge xs \ ys. \ \text{set } (\text{comb } xs \ ys) = \text{set } xs \cup \text{set } ys$
assumes $ps1 = \text{apply-ncrit } \text{ncrit } \text{data } \text{gs } \text{bs } \text{hs } (\text{apply-icrit } \text{icrit } \text{data } \text{gs } \text{bs } \text{hs } (np \ \text{gs } \text{bs } \text{hs } \text{data}))$
shows $\text{set } (\text{add-pairs } \text{gs } \text{bs } \text{ps } \text{hs } \text{data}) =$
 $\{(p, q) \mid p \ q. \ (\text{False}, p, q) \in \text{set } ps1 \vee ((p, q) \in \text{set } ps \wedge \neg \text{ocrit } \text{data } \text{hs } ps1 \ p \ q)\}$
 $\langle \text{proof} \rangle$

lemma *set-add-pairs-iff*:

assumes $\bigwedge xs \ ys. \ \text{set } (\text{comb } xs \ ys) = \text{set } xs \cup \text{set } ys$
assumes $ps1 = \text{apply-ncrit } \text{ncrit } \text{data } \text{gs } \text{bs } \text{hs } (\text{apply-icrit } \text{icrit } \text{data } \text{gs } \text{bs } \text{hs } (np \ \text{gs } \text{bs } \text{hs } \text{data}))$
shows $((p, q) \in \text{set } (\text{add-pairs } \text{gs } \text{bs } \text{ps } \text{hs } \text{data})) \longleftrightarrow$
 $((\text{False}, p, q) \in \text{set } ps1 \vee ((p, q) \in \text{set } ps \wedge \neg \text{ocrit } \text{data } \text{hs } ps1 \ p \ q))$
 $\langle \text{proof} \rangle$

lemma *ap-spec-add-pairs*:

assumes *np-spec np and icrit-spec icrit and ncrit-spec ncrit and ocrit-spec ocrit*

and $\bigwedge xs\ ys.\ set\ (comb\ xs\ ys) = set\ xs \cup set\ ys$
shows *ap-spec add-pairs*
 ⟨*proof*⟩

end

abbreviation *add-pairs-canon* \equiv
add-pairs (new-pairs-sorted canon-pair-order) component-crit chain-ncrit chain-ocrit
canon-pair-comb

lemma *ap-spec-add-pairs-canon: ap-spec add-pairs-canon*
 ⟨*proof*⟩

6.4 Suitable Instances of the *completion* Parameter

definition *rcp-spec* :: ('t, 'b::field, 'c, 'd) *complT* \Rightarrow *bool*

where *rcp-spec rcp* \longleftrightarrow
 ($\forall gs\ bs\ ps\ sps\ data.$
 $0 \notin fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \wedge$
 $(\forall h\ b.\ h \in set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \longrightarrow b \in set\ gs \cup set\ bs \longrightarrow$
fst $b \neq 0 \longrightarrow$
 $\neg lt\ (fst\ b)\ adds_t\ lt\ (fst\ h)) \wedge$
 $(\forall d.\ dickson\ grading\ d \longrightarrow$
 $dgrad\text{-}p\text{-}set\text{-}le\ d\ (fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))\ (args\text{-}to\text{-}set$
 $(gs,\ bs,\ sps))) \wedge$
 $component\text{-}of\text{-}term\ 'Keys\ (fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))) \subseteq$
 $component\text{-}of\text{-}term\ 'Keys\ (args\text{-}to\text{-}set\ (gs,\ bs,\ sps)) \wedge$
 $(is\ Groebner\ basis\ (fst\ 'set\ gs) \longrightarrow unique\text{-}idx\ (gs\ @\ bs)\ data \longrightarrow$
 $(fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \subseteq pmdl\ (args\text{-}to\text{-}set\ (gs,\ bs,\ sps)))$
 \wedge
 $(\forall (p,\ q) \in set\ sps.\ set\ sps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow$
 $(red\ (fst\ 'set\ (set\ gs \cup set\ bs)) \cup fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))))^{**}$
 $(spoly\ (fst\ p)\ (fst\ q)\ 0))))$

Informally, *rcp-spec rcp* expresses that, for suitable *gs*, *bs* and *sps*, the value of *rcp gs bs ps sps*

- is a list consisting exclusively of non-zero polynomials contained in the module generated by $set\ bs \cup set\ gs$, whose leading terms are not divisible by the leading term of any non-zero $b \in set\ bs$, and
- contains sufficiently many new polynomials such that all S-polynomials originating from *sps* can be reduced to 0 modulo the enlarged list of polynomials.

lemma *rcp-specI*:

assumes $\bigwedge gs\ bs\ ps\ sps\ data.\ 0 \notin fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))$
assumes $\bigwedge gs\ bs\ ps\ sps\ h\ b\ data.\ h \in set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \Longrightarrow b \in set\ gs \cup set\ bs \Longrightarrow fst\ b \neq 0 \Longrightarrow$

$\neg lt (fst b) \text{ adds}_t lt (fst h)$

assumes $\bigwedge gs \ bs \ ps \ sps \ d \ data. \text{ dickson-grading } d \implies$
 $dgrad\text{-}p\text{-set}\text{-}le \ d \ (fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))) \ (args\text{-}to\text{-}set$
 $(gs, \ bs, \ sps))$

assumes $\bigwedge gs \ bs \ ps \ sps \ data. \text{ component-of-term } \ ' \ Keys \ (fst \ ' \ (set \ (fst \ (rcp \ gs \ bs$
 $ps \ sps \ data)))) \subseteq$
 $\text{component-of-term } \ ' \ Keys \ (args\text{-}to\text{-}set \ (gs, \ bs, \ sps))$

assumes $\bigwedge gs \ bs \ ps \ sps \ data. \text{ is-Groebner-basis } (fst \ ' \ set \ gs) \implies \text{ unique-idx } (gs$
 $@ \ bs) \ data \implies$
 $(fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))) \subseteq \text{ pmdl } (args\text{-}to\text{-}set \ (gs, \ bs, \ sps))$

\wedge

$(\forall (p, q) \in set \ sps. \ set \ sps \subseteq set \ bs \times (set \ gs \cup set \ bs) \longrightarrow$
 $(red \ (fst \ ' \ (set \ gs \cup set \ bs) \cup fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))))^{**}$
 $(spoly \ (fst \ p) \ (fst \ q)) \ 0))$

shows $rcp\text{-}spec \ rcp$
 $\langle proof \rangle$

lemma $rcp\text{-}specD1$:

assumes $rcp\text{-}spec \ rcp$
shows $0 \notin fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))$
 $\langle proof \rangle$

lemma $rcp\text{-}specD2$:

assumes $rcp\text{-}spec \ rcp$
and $h \in set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))$ **and** $b \in set \ gs \cup set \ bs$ **and** $fst \ b \neq 0$
shows $\neg lt (fst b) \text{ adds}_t lt (fst h)$
 $\langle proof \rangle$

lemma $rcp\text{-}specD3$:

assumes $rcp\text{-}spec \ rcp$ **and** $dickson\text{-}grading \ d$
shows $dgrad\text{-}p\text{-set}\text{-}le \ d \ (fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data))) \ (args\text{-}to\text{-}set \ (gs, \ bs,$
 $sps))$
 $\langle proof \rangle$

lemma $rcp\text{-}specD4$:

assumes $rcp\text{-}spec \ rcp$
shows $\text{component-of-term } \ ' \ Keys \ (fst \ ' \ (set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data)))) \subseteq$
 $\text{component-of-term } \ ' \ Keys \ (args\text{-}to\text{-}set \ (gs, \ bs, \ sps))$
 $\langle proof \rangle$

lemma $rcp\text{-}specD5$:

assumes $rcp\text{-}spec \ rcp$ **and** $\text{ is-Groebner-basis } (fst \ ' \ set \ gs)$ **and** $\text{ unique-idx } (gs \ @$
 $bs) \ data$
shows $fst \ ' \ set \ (fst \ (rcp \ gs \ bs \ ps \ sps \ data)) \subseteq \text{ pmdl } (args\text{-}to\text{-}set \ (gs, \ bs, \ sps))$
 $\langle proof \rangle$

lemma $rcp\text{-}specD6$:

assumes $rcp\text{-}spec \ rcp$ **and** $\text{ is-Groebner-basis } (fst \ ' \ set \ gs)$ **and** $\text{ unique-idx } (gs \ @$
 $bs) \ data$

and $set\ sps \subseteq set\ bs \times (set\ gs \cup set\ bs)$
and $(p, q) \in set\ sps$
shows $(red\ (fst\ ' (set\ gs \cup set\ bs) \cup fst\ ' set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))))^{**}$
 $(spoly\ (fst\ p)\ (fst\ q))\ 0$
 $\langle proof \rangle$

lemma *compl-struct-rcp*:
assumes *rcp-spec rcp*
shows *compl-struct rcp*
 $\langle proof \rangle$

lemma *compl-pmdl-rcp*:
assumes *rcp-spec rcp*
shows *compl-pmdl rcp*
 $\langle proof \rangle$

lemma *compl-conn-rcp*:
assumes *rcp-spec rcp*
shows *compl-conn rcp*
 $\langle proof \rangle$

end

6.5 Suitable Instances of the *add-basis* Parameter

definition *add-basis-naive* :: $('a, 'b, 'c, 'd)\ abT$
where *add-basis-naive* $gs\ bs\ ns\ data = bs\ @\ ns$

lemma *ab-spec-add-basis-naive*: *ab-spec add-basis-naive*
 $\langle proof \rangle$

definition *add-basis-sorted* :: $(nat \times 'd \Rightarrow ('a, 'b, 'c)\ pdata \Rightarrow ('a, 'b, 'c)\ pdata$
 $\Rightarrow bool) \Rightarrow ('a, 'b, 'c, 'd)\ abT$
where *add-basis-sorted* $rel\ gs\ bs\ ns\ data = merge-wrt\ (rel\ data)\ bs\ ns$

lemma *ab-spec-add-basis-sorted*: *ab-spec (add-basis-sorted rel)*
 $\langle proof \rangle$

definition *card-keys* :: $('a \Rightarrow_0 'b::zero) \Rightarrow nat$
where *card-keys* = $card \circ keys$

definition (in *ordered-term*) *canon-basis-order* :: $'d \Rightarrow ('t, 'b::zero, 'c)\ pdata \Rightarrow$
 $('t, 'b, 'c)\ pdata \Rightarrow bool$
where *canon-basis-order* $data\ p\ q \longleftrightarrow$
 $(let\ cp = card-keys\ (fst\ p); cq = card-keys\ (fst\ q)\ in$
 $cp < cq \vee (cp = cq \wedge lt\ (fst\ p) \prec_t\ lt\ (fst\ q)))$

abbreviation (in *ordered-term*) *add-basis-canon* $\equiv add-basis-sorted\ canon-basis-order$

6.6 Special Case: Scalar Polynomials

context *gd-powerprod*
begin

lemma *remdups-map-component-of-term-punit*:
 remdups (map (λ-. ()) (punit.Keys-to-list (map fst bs))) =
 (if (∀ b∈set bs. fst b = 0) then [] else [()])
 ⟨*proof*⟩

lemma *count-const-lt-components-punit* [*code*]:
 punit.count-const-lt-components hs =
 (if (∃ h∈set hs. punit.const-lt-component (fst h) = Some ()) then 1 else 0)
 ⟨*proof*⟩

lemma *count-rem-components-punit* [*code*]:
 punit.count-rem-components bs =
 (if (∀ b∈set bs. fst b = 0) then 0
 else
 if (∃ b∈set bs. fst b ≠ 0 ∧ punit.const-lt-component (fst b) = Some ()) then
 0 else 1)
 ⟨*proof*⟩

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

abbreviation *add-pairs-punit-canon* ≡
 punit.add-pairs (punit.new-pairs-sorted punit.canon-pair-order) punit.product-crit
 punit.chain-ncrit
 punit.chain-ocrit punit.canon-pair-comb

lemma *ap-spec-add-pairs-punit-canon*: *punit.ap-spec add-pairs-punit-canon*
 ⟨*proof*⟩

end

end

7 Buchberger's Algorithm

theory *Buchberger*
 imports *Algorithm-Schema*
begin

context *gd-term*
begin

7.1 Reduction

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

lemma $trdsp\text{-alt}$: $trdsp \text{ bs } (p, q) = trd \text{ bs } (\text{spoly } (\text{fst } p) (\text{fst } q))$
 $\langle \text{proof} \rangle$

lemma $trdsp\text{-in-pmdl}$: $trdsp \text{ bs } (p, q) \in \text{pmdl } (\text{insert } (\text{fst } p) (\text{insert } (\text{fst } q) (\text{set } \text{bs})))$
 $\langle \text{proof} \rangle$

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

lemma $components\text{-trdsp-subset}$:
 $component\text{-of-term } ' \text{ keys } (trdsp \text{ bs } (p, q)) \subseteq component\text{-of-term } ' \text{ Keys } (\text{insert } (\text{fst } p) (\text{insert } (\text{fst } q) (\text{set } \text{bs})))$
 $\langle \text{proof} \rangle$

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

Actually, $gb\text{-red-aux}$ is only called on singleton lists.

lemma $set\text{-gb-red-aux}$: $set (gb\text{-red-aux } \text{bs } \text{ps}) = (trdsp (\text{map } \text{fst } \text{bs})) ' \text{ set } \text{ps} - \{0\}$
 $\langle \text{proof} \rangle$

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

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

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

lemma $gb\text{-red-aux-irreducible}$:
assumes $h \in \text{set } (gb\text{-red-aux } \text{bs } \text{ps})$ **and** $b \in \text{set } \text{bs}$ **and** $\text{fst } b \neq 0$
shows $\neg lt (\text{fst } b) \text{ adds}_t \text{ lt } h$
 $\langle \text{proof} \rangle$

lemma *gb-red-aux-dgrad-p-set-le*:

assumes *dickson-grading* *d*

shows *dgrad-p-set-le* *d* (*set* (*gb-red-aux* *bs ps*)) (*args-to-set* (\square , *bs*, *ps*))

<proof>

lemma *components-gb-red-aux-subset*:

component-of-term ' *Keys* (*set* (*gb-red-aux* *bs ps*)) \subseteq *component-of-term* ' *Keys*
(*args-to-set* (\square , *bs*, *ps*))

<proof>

lemma *pmdl-gb-red-aux*: *set* (*gb-red-aux* *bs ps*) \subseteq *pmdl* (*args-to-set* (\square , *bs*, *ps*))

<proof>

lemma *gb-red-aux-spoly-reducible*:

assumes (*p*, *q*) \in *set* *ps*

shows (*red* (*fst* ' *set* *bs* \cup *set* (*gb-red-aux* *bs ps*)))** (*spoly* (*fst* *p*) (*fst* *q*)) 0

<proof>

definition *gb-red* :: ('*t*, '*b*::*field*, '*c*::*default*, '*d*) *complT*

where *gb-red* *gs* *bs* *ps* *sps* *data* = (*map* ($\lambda h.$ (*h*, *default*)) (*gb-red-aux* (*gs* @ *bs*)
sps), *snd* *data*)

lemma *fst-set-fst-gb-red*: *fst* ' *set* (*fst* (*gb-red* *gs* *bs* *ps* *sps* *data*)) = *set* (*gb-red-aux*
(*gs* @ *bs*) *sps*)

<proof>

lemma *rcp-spec-gb-red*: *rcp-spec* *gb-red*

<proof>

lemmas *compl-struct-gb-red* = *compl-struct-rcp*[*OF* *rcp-spec-gb-red*]

lemmas *compl-pmdl-gb-red* = *compl-pmdl-rcp*[*OF* *rcp-spec-gb-red*]

lemmas *compl-conn-gb-red* = *compl-conn-rcp*[*OF* *rcp-spec-gb-red*]

7.2 Pair Selection

primrec *gb-sel* :: ('*t*, '*b*::*zero*, '*c*, '*d*) *selT* **where**

gb-sel *gs* *bs* [] *data* = []

gb-sel *gs* *bs* (*p* # *ps*) *data* = [*p*]

lemma *sel-spec-gb-sel*: *sel-spec* *gb-sel*

<proof>

7.3 Buchberger's Algorithm

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

<proof>

definition *gb-aux* :: ('*t*, '*b*, '*c*) *pdata* *list* \Rightarrow *nat* \times *nat* \times '*d* \Rightarrow ('*t*, '*b*, '*c*) *pdata*
list \Rightarrow

(t, b, c) pdata-pair list $\Rightarrow (t, b::\text{field}, c::\text{default})$ pdata list
where $gb\text{-aux} = gb\text{-schema-aux } gb\text{-sel } add\text{-pairs-canon } add\text{-basis-canon } gb\text{-red}$

lemmas $gb\text{-aux-simps } [code] = gb\text{-schema-aux-simps}[OF \text{ struct-spec-gb, folded } gb\text{-aux-def}]$

definition $gb :: (t, b, c)$ pdata' list $\Rightarrow d \Rightarrow (t, b::\text{field}, c::\text{default})$ pdata' list
where $gb = gb\text{-schema-direct } gb\text{-sel } add\text{-pairs-canon } add\text{-basis-canon } gb\text{-red}$

lemmas $gb\text{-simps } [code] = gb\text{-schema-direct-def}[of \text{ gb-sel } add\text{-pairs-canon } add\text{-basis-canon } gb\text{-red, folded } gb\text{-def } gb\text{-aux-def}]$

lemmas $gb\text{-isGB} = gb\text{-schema-direct-isGB}[OF \text{ struct-spec-gb } compl\text{-conn-gb-red, folded } gb\text{-def}]$

lemmas $gb\text{-pmdl} = gb\text{-schema-direct-pmdl}[OF \text{ struct-spec-gb } compl\text{-pmdl-gb-red, folded } gb\text{-def}]$

7.3.1 Special Case: punit

lemma (in *gd-term*) $struct\text{-spec-gb-punit}$: $punit.struct\text{-spec } punit.gb\text{-sel } add\text{-pairs-punit-canon } punit.add\text{-basis-canon } punit.gb\text{-red}$
<proof>

definition $gb\text{-aux-punit} :: (a, b, c)$ pdata list $\Rightarrow nat \times nat \times d \Rightarrow (a, b, c)$ pdata list \Rightarrow
 (a, b, c) pdata-pair list $\Rightarrow (a, b::\text{field}, c::\text{default})$ pdata list
where $gb\text{-aux-punit} = punit.gb\text{-schema-aux } punit.gb\text{-sel } add\text{-pairs-punit-canon } punit.add\text{-basis-canon } punit.gb\text{-red}$

lemmas $gb\text{-aux-punit-simps } [code] = punit.gb\text{-schema-aux-simps}[OF \text{ struct-spec-gb-punit, folded } gb\text{-aux-punit-def}]$

definition $gb\text{-punit} :: (a, b, c)$ pdata' list $\Rightarrow d \Rightarrow (a, b::\text{field}, c::\text{default})$ pdata' list
where $gb\text{-punit} = punit.gb\text{-schema-direct } punit.gb\text{-sel } add\text{-pairs-punit-canon } punit.add\text{-basis-canon } punit.gb\text{-red}$

lemmas $gb\text{-punit-simps } [code] = punit.gb\text{-schema-direct-def}[of \text{ punit.gb-sel } add\text{-pairs-punit-canon } punit.add\text{-basis-canon } punit.gb\text{-red, folded } gb\text{-punit-def } gb\text{-aux-punit-def}]$

lemmas $gb\text{-punit-isGB} = punit.gb\text{-schema-direct-isGB}[OF \text{ struct-spec-gb-punit } punit.compl\text{-conn-gb-red, folded } gb\text{-punit-def}]$

lemmas $gb\text{-punit-pmdl} = punit.gb\text{-schema-direct-pmdl}[OF \text{ struct-spec-gb-punit } punit.compl\text{-pmdl-gb-red, folded } gb\text{-punit-def}]$

end

end

8 Benchmark Problems for Computing Gröbner Bases

```
theory Benchmarks
  imports Polynomials.MPoly-Type-Class-OAlist
begin
```

This theory defines various well-known benchmark problems for computing Gröbner bases. The actual tests of the different algorithms on these problems are contained in the theories whose names end with *-Examples*.

8.1 Cyclic

```
definition cycl-pp :: nat ⇒ nat ⇒ nat ⇒ (nat, nat) pp
  where cycl-pp n d i = sparse0 (map (λk. (modulo (k + i) n, 1)) [0..

```

```
definition cyclic :: (nat, nat) pp nat-term-order ⇒ nat ⇒ ((nat, nat) pp ⇒0
'a::{zero,one,uminus}) list
  where cyclic to n =
    (let xs = [0..0 to (map (λi. (cycl-pp n d i, 1)) xs)) [1..0 to [(cycl-pp n n 0, 1), (0, -1)]]
    )
```

cyclic n is a system of n polynomials in n indeterminates, with maximum degree n .

8.2 Katsura

```
definition katsura-poly :: (nat, nat) pp nat-term-order ⇒ nat ⇒ nat ⇒ ((nat,
nat) pp ⇒0 'a::comm-ring-1)
  where katsura-poly to n i =
    change-ord to ((∑ j::int=-int n..0
(nat (abs j)) * V0 (nat (abs (i - j))) else 0) - V0 i)
```

```
definition katsura :: (nat, nat) pp nat-term-order ⇒ nat ⇒ ((nat, nat) pp ⇒0
'a::comm-ring-1) list
  where katsura to n =
    (let xs = [0..0 to ((sparse0 [(0, 1)], 1) # (map (λi. (sparse0 [(Suc i, 1)], 2))
xs) @ [(0, -1)])) #
      (map (katsura-poly to n) xs)
    )
```

For $1 \leq n$, *katsura n* is a system of $n + 1$ polynomials in $n + 1$ indeterminates, with maximum degree 2.

8.3 Eco

definition *eco-poly* :: (nat, nat) pp nat-term-order ⇒ nat ⇒ nat ⇒ ((nat, nat) pp ⇒₀ 'a::comm-ring-1)

where *eco-poly to m i* =

$$\text{distr}_0 \text{ to } ((\text{sparse}_0 [(i, 1), (m, 1)], 1) \# \text{map } (\lambda j. (\text{sparse}_0 [(j, 1), (j + i + 1, 1), (m, 1)], 1)) [0..<m - i - 1])$$

definition *eco* :: (nat, nat) pp nat-term-order ⇒ nat ⇒ ((nat, nat) pp ⇒₀ 'a::comm-ring-1) list

where *eco to n* =

$$\begin{aligned} & (\text{let } m = n - 1 \text{ in} \\ & (\text{distr}_0 \text{ to } ((\text{map } (\lambda j. (\text{sparse}_0 [(j, 1)], 1)) [0..<m]) @ [(0, 1)])) \# \\ & (\text{distr}_0 \text{ to } [(\text{sparse}_0 [(m-1, 1), (m, 1)], 1), (0, - \text{of-nat } m)]) \# \\ & (\text{rev } (\text{map } (\text{eco-poly to } m) [0..<m-1])) \\ &) \end{aligned}$$

For $(2::'a) \leq n$, *eco n* is a system of n polynomials in n indeterminates, with maximum degree 3.

8.4 Noon

definition *noon-poly* :: (nat, nat) pp nat-term-order ⇒ nat ⇒ nat ⇒ ((nat, nat) pp ⇒₀ 'a::comm-ring-1)

where *noon-poly to n i* =

$$\begin{aligned} & (\text{let } \text{ten} = \text{of-nat } 10; \text{eleven} = - \text{of-nat } 11 \text{ in} \\ & \text{distr}_0 \text{ to } ((\text{map } (\lambda j. \text{if } j = i \text{ then } (\text{sparse}_0 [(i, 1)], \text{eleven}) \text{ else } (\text{sparse}_0 [(j, 2), (i, 1)], \text{ten})) [0..<n]) @ \\ & [(0, \text{ten})]) \end{aligned}$$

definition *noon* :: (nat, nat) pp nat-term-order ⇒ nat ⇒ ((nat, nat) pp ⇒₀ 'a::comm-ring-1) list

where *noon to n* = (*noon-poly to n 1*) # (*noon-poly to n 0*) # (*map (noon-poly to n) [2..<n]*)

For $(2::'a) \leq n$, *noon n* is a system of n polynomials in n indeterminates, with maximum degree 3.

end

9 Code Equations Related to the Computation of Gröbner Bases

theory *Algorithm-Schema-Impl*

imports *Algorithm-Schema Benchmarks*

begin

lemma *card-keys-MP-oalist* [code]: *card-keys (MP-oalist xs) = length (fst (list-of-oalist-ntm xs))*

<proof>

end

theory *Code-Target-Rat*

imports *Complex-Main HOL-Library.Code-Target-Numeral*

begin

Mapping type *rat* to type "Rat.rat" in Isabelle/ML. Serialization for other target languages will be provided in the future.

context includes *integer.lifting* **begin**

lift-definition *rat-of-integer* :: *integer* \Rightarrow *rat* **is** *Rat.of-int* *<proof>*

lift-definition *quotient-of'* :: *rat* \Rightarrow *integer* \times *integer* **is** *quotient-of* *<proof>*

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

lemma [*code-unfold*]: *quotient-of* = ($\lambda x.$ *map-prod int-of-integer int-of-integer* (*quotient-of'* *x*))
<proof>

end

code-printing

type-constructor *rat* \rightarrow
(*Eval*) *Rat.rat* |
constant *plus* :: *rat* \Rightarrow - \Rightarrow - \rightarrow
(*Eval*) *Rat.add* |
constant *minus* :: *rat* \Rightarrow - \Rightarrow - \rightarrow
(*Eval*) *Rat.add* ((-)) (*Rat.neg* ((-))) |
constant *times* :: *rat* \Rightarrow - \Rightarrow - \rightarrow
(*Eval*) *Rat.mult* |
constant *inverse* :: *rat* \Rightarrow - \rightarrow
(*Eval*) *Rat.inv* |
constant *divide* :: *rat* \Rightarrow - \Rightarrow - \rightarrow
(*Eval*) *Rat.mult* ((-)) (*Rat.inv* ((-))) |
constant *rat-of-integer* :: *integer* \Rightarrow *rat* \rightarrow
(*Eval*) *Rat.of'-int* |
constant *abs* :: *rat* \Rightarrow - \rightarrow
(*Eval*) *Rat.abs* |
constant *0* :: *rat* \rightarrow
(*Eval*) !(*Rat.make* (0, 1)) |
constant *1* :: *rat* \rightarrow
(*Eval*) !(*Rat.make* (1, 1)) |
constant *uminus* :: *rat* \Rightarrow *rat* \rightarrow
(*Eval*) *Rat.neg* |

```

constant HOL.equal :: rat ⇒ - →
  (Eval) !((- : Rat.rat) = -) |
constant quotient-of' →
  (Eval) Rat.dest

```

end

10 Sample Computations with Buchberger's Algorithm

```

theory Buchberger-Examples
  imports Buchberger Algorithm-Schema-Impl Code-Target-Rat
begin

```

```

lemma (in gd-term) compute-trd-aux [code]:
  trd-aux fs p r =
    (if is-zero p then
      r
    else
      case find-adds fs (lt p) of
        None ⇒ trd-aux fs (tail p) (plus-monomial-less r (lc p) (lt p))
      | Some f ⇒ trd-aux fs (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail
f)) r
    )
  ⟨proof⟩

```

10.1 Scalar Polynomials

```

global-interpretation punit': gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit
cmp-term

```

```

  rewrites punit.adds-term = (adds)
  and punit.pp-of-term = ( $\lambda x. x$ )
  and punit.component-of-term = ( $\lambda-. ()$ )
  and punit.monom-mult = monom-mult-punit
  and punit.mult-scalar = mult-scalar-punit
  and punit'.punit.min-term = min-term-punit
  and punit'.punit.lt = lt-punit cmp-term
  and punit'.punit.lc = lc-punit cmp-term
  and punit'.punit.tail = tail-punit cmp-term
  and punit'.punit.ord-p = ord-p-punit cmp-term
  and punit'.punit.ord-strict-p = ord-strict-p-punit cmp-term
  for cmp-term :: ('a::nat, 'b::{nat,add-wellorder}) pp nat-term-order

```

```

  defines find-adds-punit = punit'.punit.find-adds
  and trd-aux-punit = punit'.punit.trd-aux

```

and *trd-punit* = *punit'.punit.trd*
and *spoly-punit* = *punit'.punit.spoly*
and *count-const-lt-components-punit* = *punit'.punit.count-const-lt-components*
and *count-rem-components-punit* = *punit'.punit.count-rem-components*
and *const-lt-component-punit* = *punit'.punit.const-lt-component*
and *full-gb-punit* = *punit'.punit.full-gb*
and *add-pairs-single-sorted-punit* = *punit'.punit.add-pairs-single-sorted*
and *add-pairs-punit* = *punit'.punit.add-pairs*
and *canon-pair-order-aux-punit* = *punit'.punit.canon-pair-order-aux*
and *canon-basis-order-punit* = *punit'.punit.canon-basis-order*
and *new-pairs-sorted-punit* = *punit'.punit.new-pairs-sorted*
and *product-crit-punit* = *punit'.punit.product-crit*
and *chain-ncrit-punit* = *punit'.punit.chain-ncrit*
and *chain-ocrit-punit* = *punit'.punit.chain-ocrit*
and *apply-icrit-punit* = *punit'.punit.apply-icrit*
and *apply-ncrit-punit* = *punit'.punit.apply-ncrit*
and *apply-ocrit-punit* = *punit'.punit.apply-ocrit*
and *trdsp-punit* = *punit'.punit.trdsp*
and *gb-sel-punit* = *punit'.punit.gb-sel*
and *gb-red-aux-punit* = *punit'.punit.gb-red-aux*
and *gb-red-punit* = *punit'.punit.gb-red*
and *gb-aux-punit* = *punit'.punit.gb-aux-punit*
and *gb-punit* = *punit'.punit.gb-punit* — Faster, because incorporates product
criterion.
<proof>

lemma *compute-spoly-punit* [code]:
spoly-punit to p q = (let t1 = lt-punit to p; t2 = lt-punit to q; l = lcs t1 t2 in
(monom-mult-punit (1 / lc-punit to p) (l - t1) p) - (monom-mult-punit
(1 / lc-punit to q) (l - t2) q))
<proof>

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

experiment begin interpretation *trivariate₀-rat* *<proof>*

lemma
lt-punit DRLEX $(X^2 * Z^3 + 3 * X^2 * Y) = \text{sparse}_0 [(0, 2), (2, 3)]$
<proof>

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

lemma
tail-punit DRLEX $(X^2 * Z^3 + 3 * X^2 * Y) = 3 * X^2 * Y$
<proof>

lemma

ord-strict-p-punit DRLEX $(X^2 * Z^4 - 2 * Y^3 * Z^2) (X^2 * Z^7 + 2 * Y^3 * Z^2)$
<proof>

lemma

trd-punit DRLEX $[Y^2 * Z + 2 * Y * Z^3] (X^2 * Z^4 - 2 * Y^3 * Z^3) =$
 $X^2 * Z^4 + Y^4 * Z$
<proof>

lemma

spoly-punit DRLEX $(X^2 * Z^4 - 2 * Y^3 * Z^2) (Y^2 * Z + 2 * Z^3) =$
 $-2 * Y^3 * Z^2 - (C_0 (1 / 2)) * X^2 * Y^2 * Z^2$
<proof>

lemma

gb-punit DRLEX
[
 $(X^2 * Z^4 - 2 * Y^3 * Z^2, ()),$
 $(Y^2 * Z + 2 * Z^3, ())$
] $() =$
[
 $(-2 * Y^3 * Z^2 - (C_0 (1 / 2)) * X^2 * Y^2 * Z^2, ()),$
 $(X^2 * Z^4 - 2 * Y^3 * Z^2, ()),$
 $(Y^2 * Z + 2 * Z^3, ()),$
 $(-(C_0 (1 / 2)) * X^2 * Y^4 * Z - 2 * Y^5 * Z, ())$
]
<proof>

lemma

gb-punit DRLEX
[
 $(X^2 * Z^2 - Y, ()),$
 $(Y^2 * Z - 1, ())$
] $() =$
[
 $(-(Y^3) + X^2 * Z, ()),$
 $(X^2 * Z^2 - Y, ()),$
 $(Y^2 * Z - 1, ())$
]
<proof>

lemma

gb-punit DRLEX
[
 $(X^3 - X * Y * Z^2, ()),$
 $(Y^2 * Z - 1, ())$
]

```

] () =
[
  (- (X ^ 3 * Y) + X * Z, ()),
  (X ^ 3 - X * Y * Z^2, ()),
  (Y^2 * Z - 1, ()),
  (- (X * Z ^ 3) + X ^ 5, ())
]
⟨proof⟩

```

lemma

```

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

```

end

value [code] length (gb-punit DRLEX (map (λp. (p, ())) ((katsura DRLEX 2)::(-
⇒₀ rat) list)) ()))

value [code] length (gb-punit DRLEX (map (λp. (p, ())) ((cyclic DRLEX 5)::(-
⇒₀ rat) list)) ()))

10.2 Vector Polynomials

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

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

definition *monom-mult-pprod* :: 'c::semiring-0 ⇒ ('a::nat, 'b::nat) pp ⇒ -
where *monom-mult-pprod* = pprod.monom-mult

definition *mult-scalar-pprod* :: (('a::nat, 'b::nat) pp ⇒₀ 'c::semiring-0) ⇒ -
where *mult-scalar-pprod* = pprod.mult-scalar

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

global-interpretation *pprod'*: *gd-nat-term* λx::('a, 'b) pp × 'c. x λx. x *cmp-term*
rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*

```

and pprod.splus = splus-pprod
and pprod.monom-mult = monom-mult-pprod
and pprod.mult-scalar = mult-scalar-pprod
and pprod.adds-term = adds-term-pprod
for cmp-term :: (('a::nat, 'b::nat) pp × 'c::{nat,the-min}) nat-term-order
defines shift-map-keys-pprod = pprod'.shift-map-keys
and min-term-pprod = pprod'.min-term
and lt-pprod = pprod'.lt
and lc-pprod = pprod'.lc
and tail-pprod = pprod'.tail
and comp-opt-p-pprod = pprod'.comp-opt-p
and ord-p-pprod = pprod'.ord-p
and ord-strict-p-pprod = pprod'.ord-strict-p
and find-adds-pprod = pprod'.find-adds
and trd-aux-pprod = pprod'.trd-aux
and trd-pprod = pprod'.trd
and spoly-pprod = pprod'.spoly
and count-const-lt-components-pprod = pprod'.count-const-lt-components
and count-rem-components-pprod = pprod'.count-rem-components
and const-lt-component-pprod = pprod'.const-lt-component
and full-gb-pprod = pprod'.full-gb
and keys-to-list-pprod = pprod'.keys-to-list
and Keys-to-list-pprod = pprod'.Keys-to-list
and add-pairs-single-sorted-pprod = pprod'.add-pairs-single-sorted
and add-pairs-pprod = pprod'.add-pairs
and canon-pair-order-aux-pprod = pprod'.canon-pair-order-aux
and canon-basis-order-pprod = pprod'.canon-basis-order
and new-pairs-sorted-pprod = pprod'.new-pairs-sorted
and component-crit-pprod = pprod'.component-crit
and chain-ncrit-pprod = pprod'.chain-ncrit
and chain-ocrit-pprod = pprod'.chain-ocrit
and apply-icrit-pprod = pprod'.apply-icrit
and apply-ncrit-pprod = pprod'.apply-ncrit
and apply-ocrit-pprod = pprod'.apply-ocrit
and trdsp-pprod = pprod'.trdsp
and gb-sel-pprod = pprod'.gb-sel
and gb-red-aux-pprod = pprod'.gb-red-aux
and gb-red-pprod = pprod'.gb-red
and gb-aux-pprod = pprod'.gb-aux
and gb-pprod = pprod'.gb
⟨proof⟩

```

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

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

lemma *compute-shift-map-keys-pprod* [code abstract]:

list-of-oalist-ntm (*shift-map-keys-pprod* $t f xs$) = *map-raw* ($\lambda(k, v).$ (*splus-pprod* $t k, f v$)) (*list-of-oalist-ntm* xs)
 ⟨proof⟩

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

⟨proof⟩

lemmas [code] = *conversep-iff*

definition $Vec_0 :: nat \Rightarrow (('a, nat) pp \Rightarrow_0 'b) \Rightarrow (('a::nat, nat) pp \times nat) \Rightarrow_0 'b::semiring-1$ **where**

$Vec_0 i p = mult-scalar-pprod p (Poly-Mapping.single (0, i) 1)$

experiment begin interpretation *trivariate₀-rat* ⟨proof⟩

lemma

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

lemma

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

lemma

lt-pprod (*POT DRLEX*) ($Vec_0 1 (X^2 * Z) + Vec_0 0 (2 * Y^3 * Z^2)$) = (*sparse₀* [(0, 2), (2, 1)], 1)
 ⟨proof⟩

lemma

keys ($Vec_0 0 (X^2 * Z^3) + Vec_0 1 (2 * Y^3 * Z^2)$) =
 {(*sparse₀* [(0, 2), (2, 3)], 0), (*sparse₀* [(1, 3), (2, 2)], 1)}
 ⟨proof⟩

lemma

keys ($Vec_0 0 (X^2 * Z^3) + Vec_0 2 (2 * Y^3 * Z^2)$) =
 {(*sparse₀* [(0, 2), (2, 3)], 0), (*sparse₀* [(1, 3), (2, 2)], 2)}
 ⟨proof⟩

lemma

$Vec_0 1 (X^2 * Z^7 + 2 * Y^3 * Z^2) + Vec_0 3 (X^2 * Z^4) + Vec_0 1 (- 2 * Y^3 * Z^2) =$
 $Vec_0 1 (X^2 * Z^7) + Vec_0 3 (X^2 * Z^4)$
 ⟨proof⟩

lemma

$lookup (Vec_0 0 (X^2 * Z^\gamma) + Vec_0 1 (2 * Y^3 * Z^2 + 2)) (sparse_0 [(0, 2), (2, \gamma)], 0) = 1$
 ⟨proof⟩

lemma

$lookup (Vec_0 0 (X^2 * Z^\gamma) + Vec_0 1 (2 * Y^3 * Z^2 + 2)) (sparse_0 [(0, 2), (2, \gamma)], 1) = 0$
 ⟨proof⟩

lemma

$Vec_0 0 (0 * X^2 * Z^\gamma) + Vec_0 1 (0 * Y^3 * Z^2) = 0$
 ⟨proof⟩

lemma

$monom-mult-pprod 3 (sparse_0 [(1, 2::nat)]) (Vec_0 0 (X^2 * Z) + Vec_0 1 (2 * Y^3 * Z^2)) =$
 $Vec_0 0 (3 * Y^2 * Z * X^2) + Vec_0 1 (6 * Y^5 * Z^2)$
 ⟨proof⟩

lemma

$trd-pprod DRLEX [Vec_0 0 (Y^2 * Z + 2 * Y * Z^3)] (Vec_0 0 (X^2 * Z^4 - 2 * Y^3 * Z^3)) =$
 $Vec_0 0 (X^2 * Z^4 + Y^4 * Z)$
 ⟨proof⟩

lemma

$length (gb-pprod (POT DRLEX$
 [
 ($Vec_0 0 (X^2 * Z^4 - 2 * Y^3 * Z^2)$), ($()$),
 ($Vec_0 0 (Y^2 * Z + 2 * Z^3)$), ($()$)
]) ($()$) = 4
 ⟨proof⟩

end

end

11 Further Properties of Multivariate Polynomials

theory *More-MPoly-Type-Class*

imports *Polynomials.MPoly-Type-Class-Ordered General*

begin

Some further general properties of (ordered) multivariate polynomials needed for Gröbner bases. This theory is an extension of *Polynomials.MPoly-Type-Class-Ordered*.

11.1 Modules and Linear Hulls

context *module*
begin

lemma *span-listE*:

assumes $p \in \text{span } (\text{set } bs)$

obtains qs **where** $\text{length } qs = \text{length } bs$ **and** $p = \text{sum-list } (\text{map2 } (*s) \text{ } qs \text{ } bs)$

<proof>

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

<proof>

end

lemma (**in** *term-powerprod*) *monomial-1-in-pmdlI*:

assumes $(f::\Rightarrow_0 \text{'b}::\text{field}) \in \text{pmdl } F$ **and** $\text{keys } f = \{t\}$

shows $\text{monomial } 1 \text{ } t \in \text{pmdl } F$

<proof>

11.2 Ordered Polynomials

context *ordered-term*
begin

11.2.1 Sets of Leading Terms and -Coefficients

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

$\text{lt-set } F = \text{lt } \text{' } (F - \{0\})$

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

$\text{lc-set } F = \text{lc } \text{' } (F - \{0\})$

lemma *lt-setI*:

assumes $f \in F$ **and** $f \neq 0$

shows $\text{lt } f \in \text{lt-set } F$

<proof>

lemma *lt-setE*:

assumes $t \in \text{lt-set } F$

obtains f **where** $f \in F$ **and** $f \neq 0$ **and** $\text{lt } f = t$

<proof>

lemma *lt-set-iff*:

shows $t \in \text{lt-set } F \iff (\exists f \in F. f \neq 0 \wedge \text{lt } f = t)$

<proof>

lemma *lc-setI*:

assumes $f \in F$ **and** $f \neq 0$

shows $\text{lc } f \in \text{lc-set } F$

<proof>

lemma *lc-setE*:

assumes $c \in \text{lc-set } F$

obtains f where $f \in F$ and $f \neq 0$ and $\text{lc } f = c$

<proof>

lemma *lc-set-iff*:

shows $c \in \text{lc-set } F \iff (\exists f \in F. f \neq 0 \wedge \text{lc } f = c)$

<proof>

lemma *lc-set-nonzero*:

shows $0 \notin \text{lc-set } F$

<proof>

lemma *lt-sum-distinct-eq-Max*:

assumes *finite* I and $\text{sum } p \ I \neq 0$

and $\bigwedge i1 \ i2. i1 \in I \implies i2 \in I \implies p \ i1 \neq 0 \implies p \ i2 \neq 0 \implies \text{lt } (p \ i1) = \text{lt } (p \ i2) \implies i1 = i2$

shows $\text{lt } (\text{sum } p \ I) = \text{ord-term-lin.Max } (\text{lt-set } (p \ 'I))$

<proof>

lemma *lt-sum-distinct-in-lt-set*:

assumes *finite* I and $\text{sum } p \ I \neq 0$

and $\bigwedge i1 \ i2. i1 \in I \implies i2 \in I \implies p \ i1 \neq 0 \implies p \ i2 \neq 0 \implies \text{lt } (p \ i1) = \text{lt } (p \ i2) \implies i1 = i2$

shows $\text{lt } (\text{sum } p \ I) \in \text{lt-set } (p \ 'I)$

<proof>

11.2.2 Monicity

definition *monic* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::\text{field})$ where

$\text{monic } p = \text{monom-mult } (1 / \text{lc } p) \ 0 \ p$

definition *is-monic-set* :: $('t \Rightarrow_0 'b::\text{field}) \ \text{set} \Rightarrow \text{bool}$ where

$\text{is-monic-set } B \equiv (\forall b \in B. b \neq 0 \longrightarrow \text{lc } b = 1)$

lemma *lookup-monic*: $\text{lookup } (\text{monic } p) \ v = (\text{lookup } p \ v) / \text{lc } p$

<proof>

lemma *lookup-monic-lt*:

assumes $p \neq 0$

shows $\text{lookup } (\text{monic } p) \ (\text{lt } p) = 1$

<proof>

lemma *monic-0 [simp]*: $\text{monic } 0 = 0$

<proof>

lemma *monic-0-iff*: $(\text{monic } p = 0) \iff (p = 0)$

<proof>

lemma *keys-monic* [*simp*]: $keys (monic\ p) = keys\ p$
<proof>

lemma *lt-monic* [*simp*]: $lt (monic\ p) = lt\ p$
<proof>

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

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

lemma *is-monic-setI*:
 assumes $\bigwedge b. b \in B \implies b \neq 0 \implies lc\ b = 1$
 shows *is-monic-set* B
<proof>

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

lemma *Keys-image-monic* [*simp*]: $Keys (monic\ 'A) = Keys\ A$
<proof>

lemma *image-monic-is-monic-set*: *is-monic-set* $(monic\ 'A)$
<proof>

lemma *pmdl-image-monic* [*simp*]: $pmdl (monic\ 'B) = pmdl\ B$
<proof>

end

end

12 Auto-reducing Lists of Polynomials

theory *Auto-Reduction*
 imports *Reduction More-MPoly-Type-Class*
begin

12.1 Reduction and Monic Sets

context *ordered-term*

begin

lemma *is-red-monic*: $is-red\ B\ (monic\ p) \longleftrightarrow is-red\ B\ p$
 ⟨*proof*⟩

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

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

12.2 Minimal Bases and Auto-reduced Bases

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

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

lemma *is-auto-reducedD*:

assumes *is-auto-reduced* B **and** $b \in B$

shows $\neg is-red\ (B - \{b\})\ b$

⟨*proof*⟩

The converse of the following lemma is only true if B is minimal!

lemma *image-monic-is-auto-reduced*:

assumes *is-auto-reduced* B

shows *is-auto-reduced* $(monic\ 'B)$

⟨*proof*⟩

lemma *is-minimal-basisI*:

assumes $\bigwedge p. p \in B \Longrightarrow p \neq 0$ **and** $\bigwedge p\ q. p \in B \Longrightarrow q \in B \Longrightarrow p \neq q \Longrightarrow \neg lt\ p\ adds_t\ lt\ q$

shows *is-minimal-basis* B

⟨*proof*⟩

lemma *is-minimal-basisD1*:

assumes *is-minimal-basis* B **and** $p \in B$

shows $p \neq 0$

⟨*proof*⟩

lemma *is-minimal-basisD2*:

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

shows $\neg lt\ p\ adds_t\ lt\ q$

⟨*proof*⟩

lemma *is-minimal-basisD3*:

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

shows $\neg \text{lt } q \text{ adds}_t \text{ lt } p$

<proof>

lemma *is-minimal-basis-subset*:

assumes *is-minimal-basis* B **and** $A \subseteq B$

shows *is-minimal-basis* A

<proof>

lemma *nadds-red*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **and** *red*: *red* B p r

shows $r \neq 0 \wedge \text{lt } r = \text{lt } p$

<proof>

lemma *nadds-red-nonzero*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **and** *red* B p r

shows $r \neq 0$

<proof>

lemma *nadds-red-lt*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **and** *red* B p r

shows $\text{lt } r = \text{lt } p$

<proof>

lemma *nadds-red-rtrancl-lt*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **and** *rtrancl*: $(\text{red } B)^{**} p r$

shows $\text{lt } r = \text{lt } p$

<proof>

lemma *nadds-red-rtrancl-nonzero*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **and** $p \neq 0$ **and** *rtrancl*: $(\text{red } B)^{**} p r$

shows $r \neq 0$

<proof>

lemma *minimal-basis-red-rtrancl-nonzero*:

assumes *is-minimal-basis* B **and** $p \in B$ **and** $(\text{red } (B - \{p\}))^{**} p r$

shows $r \neq 0$

<proof>

lemma *minimal-basis-red-rtrancl-lt*:

assumes *is-minimal-basis* B **and** $p \in B$ **and** $(\text{red } (B - \{p\}))^{**} p r$

shows $\text{lt } r = \text{lt } p$

<proof>

lemma *is-minimal-basis-replace*:

assumes *major*: *is-minimal-basis* B **and** $p \in B$ **and** *red*: $(\text{red } (B - \{p\}))^{**} p r$

shows *is-minimal-basis* $(\text{insert } r (B - \{p\}))$

<proof>

12.3 Computing Minimal Bases

definition *comp-min-basis* :: ('t \Rightarrow_0 'b) list \Rightarrow ('t \Rightarrow_0 'b::zero) list **where**
 comp-min-basis xs = *filter-min* ($\lambda x y. lt\ x\ adds_t\ lt\ y$) (*filter* ($\lambda x. x \neq 0$) xs)

lemma *comp-min-basis-subset'*: set (*comp-min-basis* xs) \subseteq {x \in set xs. x \neq 0}
<proof>

lemma *comp-min-basis-subset*: set (*comp-min-basis* xs) \subseteq set xs
<proof>

lemma *comp-min-basis-nonzero*: p \in set (*comp-min-basis* xs) \implies p \neq 0
<proof>

lemma *comp-min-basis-adds*:
 assumes p \in set xs **and** p \neq 0
 obtains q **where** q \in set (*comp-min-basis* xs) **and** lt q *adds_t* lt p
<proof>

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

lemma *comp-min-basis-nadds*:
 assumes p \in set (*comp-min-basis* xs) **and** q \in set (*comp-min-basis* xs) **and** p \neq q
 shows \neg lt q *adds_t* lt p
<proof>

lemma *comp-min-basis-is-minimal-basis*: *is-minimal-basis* (set (*comp-min-basis* xs))
<proof>

lemma *comp-min-basis-distinct*: *distinct* (*comp-min-basis* xs)
<proof>

end

12.4 Auto-Reduction

context *gd-term*
begin

lemma *is-minimal-basis-trd-is-minimal-basis*:
 assumes *is-minimal-basis* (set (x # xs)) **and** x \notin set xs
 shows *is-minimal-basis* (set ((trd xs x) # xs))
<proof>

lemma *is-minimal-basis-trd-distinct*:

assumes *min*: *is-minimal-basis* (set (x # xs)) **and** *dist*: *distinct* (x # xs)

shows *distinct* ((trd xs x) # xs)

<proof>

primrec *comp-red-basis-aux* :: ('t \Rightarrow_0 'b) list \Rightarrow ('t \Rightarrow_0 'b) list \Rightarrow ('t \Rightarrow_0 'b::field) list **where**

comp-red-basis-aux-base: *comp-red-basis-aux* Nil ys = ys|

comp-red-basis-aux-rec: *comp-red-basis-aux* (x # xs) ys = *comp-red-basis-aux* xs ((trd (xs @ ys) x) # ys)

lemma *subset-comp-red-basis-aux*: set ys \subseteq set (*comp-red-basis-aux* xs ys)

<proof>

lemma *comp-red-basis-aux-nonzero*:

assumes *is-minimal-basis* (set (xs @ ys)) **and** *distinct* (xs @ ys) **and** $p \in$ set (*comp-red-basis-aux* xs ys)

shows $p \neq 0$

<proof>

lemma *comp-red-basis-aux-lt*:

assumes *is-minimal-basis* (set (xs @ ys)) **and** *distinct* (xs @ ys)

shows lt ' set (xs @ ys) = lt ' set (*comp-red-basis-aux* xs ys)

<proof>

lemma *comp-red-basis-aux-pmdl*:

assumes *is-minimal-basis* (set (xs @ ys)) **and** *distinct* (xs @ ys)

shows pmdl (set (*comp-red-basis-aux* xs ys)) \subseteq pmdl (set (xs @ ys))

<proof>

lemma *comp-red-basis-aux-irred*:

assumes *is-minimal-basis* (set (xs @ ys)) **and** *distinct* (xs @ ys)

and $\bigwedge y. y \in$ set ys $\implies \neg$ *is-red* (set (xs @ ys) - {y}) y

and $p \in$ set (*comp-red-basis-aux* xs ys)

shows \neg *is-red* (set (*comp-red-basis-aux* xs ys) - {p}) p

<proof>

lemma *comp-red-basis-aux-dgrad-p-set-le*:

assumes *dickson-grading* d

shows *dgrad-p-set-le* d (set (*comp-red-basis-aux* xs ys)) (set xs \cup set ys)

<proof>

definition *comp-red-basis* :: ('t \Rightarrow_0 'b) list \Rightarrow ('t \Rightarrow_0 'b::field) list

where *comp-red-basis* xs = *comp-red-basis-aux* (*comp-min-basis* xs) \square

lemma *comp-red-basis-nonzero*:

assumes $p \in$ set (*comp-red-basis* xs)

shows $p \neq 0$

<proof>

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

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

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

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

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

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

12.5 Auto-Reduction and Monicity

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

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

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

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

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

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

⟨proof⟩

lemma *comp-red-monic-basis-dgrad-p-set-le*:

assumes *dickson-grading* d

shows *dgrad-p-set-le* d (set (comp-red-monic-basis xs)) (set xs)

⟨proof⟩

end

end

13 Reduced Gröbner Bases

theory *Reduced-GB*

imports *Groebner-Bases Auto-Reduction*

begin

lemma (in *gd-term*) *GB-image-monic*: *is-Groebner-basis* (monic ‘ G) \longleftrightarrow *is-Groebner-basis* G

⟨proof⟩

13.1 Definition and Uniqueness of Reduced Gröbner Bases

context *ordered-term*

begin

definition *is-reduced-GB* :: (*t* \Rightarrow_0 *b*::field) set \Rightarrow bool **where**

is-reduced-GB B \equiv *is-Groebner-basis* B \wedge *is-auto-reduced* B \wedge *is-monic-set* B \wedge 0 \notin B

lemma *reduced-GB-D1*:

assumes *is-reduced-GB* G

shows *is-Groebner-basis* G

⟨proof⟩

lemma *reduced-GB-D2*:

assumes *is-reduced-GB* G

shows *is-auto-reduced* G

⟨proof⟩

lemma *reduced-GB-D3*:

assumes *is-reduced-GB* G

shows *is-monic-set* G

⟨proof⟩

lemma *reduced-GB-D4*:

assumes *is-reduced-GB* G **and** $g \in G$

shows $g \neq 0$
 ⟨*proof*⟩

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

end

context *gd-term*
begin

lemma *is-reduced-GB-subsetI*:
assumes *Ared*: *is-reduced-GB* A **and** *BGB*: *is-Groebner-basis* B **and** *Bmon*:
is-monic-set B
and $*$: $\bigwedge a\ b. a \in A \implies b \in B \implies a \neq 0 \implies b \neq 0 \implies a - b \neq 0 \implies lt\ (a - b) \in keys\ b \implies lt\ (a - b) \prec_t\ lt\ b \implies False$
and *id-eq*: $pmdl\ A = pmdl\ B$
shows $A \subseteq B$
 ⟨*proof*⟩

lemma *is-reduced-GB-unique'*:
assumes *Ared*: *is-reduced-GB* A **and** *Bred*: *is-reduced-GB* B **and** *id-eq*: $pmdl\ A = pmdl\ B$
shows $A \subseteq B$
 ⟨*proof*⟩

theorem *is-reduced-GB-unique*:
assumes *Ared*: *is-reduced-GB* A **and** *Bred*: *is-reduced-GB* B **and** *id-eq*: $pmdl\ A = pmdl\ B$
shows $A = B$
 ⟨*proof*⟩

13.2 Computing Reduced Gröbner Bases by Auto-Reduction

13.2.1 Minimal Bases

lemma *minimal-basis-is-reduced-GB*:
assumes *is-minimal-basis* B **and** *is-monic-set* B **and** *is-reduced-GB* G **and** $G \subseteq B$
and $pmdl\ B = pmdl\ G$
shows $B = G$
 ⟨*proof*⟩

13.2.2 Computing Minimal Bases

lemma *comp-min-basis-pmdl*:
assumes *is-Groebner-basis* (set xs)
shows $pmdl\ (set\ (comp-min-basis\ xs)) = pmdl\ (set\ xs)$ (is $pmdl\ (set\ ?ys) = -$)

<proof>

lemma *comp-min-basis-GB*:

assumes *is-Groebner-basis* (set *xs*)

shows *is-Groebner-basis* (set (comp-min-basis *xs*)) (**is** *is-Groebner-basis* (set ?*ys*))

<proof>

13.2.3 Computing Reduced Bases

lemma *comp-red-basis-pmdl*:

assumes *is-Groebner-basis* (set *xs*)

shows *pmdl* (set (comp-red-basis *xs*)) = *pmdl* (set *xs*)

<proof>

lemma *comp-red-basis-GB*:

assumes *is-Groebner-basis* (set *xs*)

shows *is-Groebner-basis* (set (comp-red-basis *xs*))

<proof>

13.2.4 Computing Reduced Gröbner Bases

lemma *comp-red-monic-basis-pmdl*:

assumes *is-Groebner-basis* (set *xs*)

shows *pmdl* (set (comp-red-monic-basis *xs*)) = *pmdl* (set *xs*)

<proof>

lemma *comp-red-monic-basis-GB*:

assumes *is-Groebner-basis* (set *xs*)

shows *is-Groebner-basis* (set (comp-red-monic-basis *xs*))

<proof>

lemma *comp-red-monic-basis-is-reduced-GB*:

assumes *is-Groebner-basis* (set *xs*)

shows *is-reduced-GB* (set (comp-red-monic-basis *xs*))

<proof>

lemma *ex-finite-reduced-GB-dgrad-p-set*:

assumes *dickson-grading* *d* **and** *finite* (component-of-term ‘ *Keys F*’) **and** $F \subseteq$
dgrad-p-set *d m*

obtains *G* **where** $G \subseteq$ *dgrad-p-set* *d m* **and** *finite* *G* **and** *is-reduced-GB* *G* **and**
 $pmdl\ G = pmdl\ F$

<proof>

theorem *ex-unique-reduced-GB-dgrad-p-set*:

assumes *dickson-grading* *d* **and** *finite* (component-of-term ‘ *Keys F*’) **and** $F \subseteq$
dgrad-p-set *d m*

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

<proof>

corollary *ex-unique-reduced-GB-dgrad-p-set'*:

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows $\exists! G. \text{ finite } G \wedge \text{ is-reduced-GB } G \wedge \text{ pmdl } G = \text{ pmdl } F$
(proof)

definition *reduced-GB :: ('t \Rightarrow_0 'b) set \Rightarrow ('t \Rightarrow_0 'b::field) set*

where *reduced-GB B = (THE G. finite G \wedge is-reduced-GB G \wedge pmdl G = pmdl B)*

reduced-GB returns the unique reduced Gröbner basis of the given set, provided its Dickson grading is bounded. Combining *comp-red-monic-basis* with any function for computing Gröbner bases, e.g. *gb* from theory "Buchberger", makes *reduced-GB* computable.

lemma *finite-reduced-GB-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows *finite (reduced-GB F)*
(proof)

lemma *reduced-GB-is-reduced-GB-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows *is-reduced-GB (reduced-GB F)*
(proof)

lemma *reduced-GB-is-GB-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows *is-Groebner-basis (reduced-GB F)*
(proof)

lemma *reduced-GB-is-auto-reduced-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows *is-auto-reduced (reduced-GB F)*
(proof)

lemma *reduced-GB-is-monic-set-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows *is-monic-set (reduced-GB F)*
(proof)

lemma *reduced-GB-nonzero-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ' Keys F) and $F \subseteq$ dgrad-p-set d m*

shows $0 \notin \text{ reduced-GB } F$
(proof)

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

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

lemma *reduced-GB-dgrad-p-set:*
assumes *dickson-grading* d **and** *finite* (component-of-term ‘ *Keys* F) **and** $F \subseteq$
dgrad-p-set d m
shows $\text{reduced-GB } F \subseteq \text{dgrad-p-set } d$ m
<proof>

lemma *reduced-GB-unique:*
assumes *finite* G **and** *is-reduced-GB* G **and** $\text{pmdl } G = \text{pmdl } F$
shows $\text{reduced-GB } F = G$
<proof>

lemma *is-reduced-GB-empty:* $\text{is-reduced-GB } \{\}$
<proof>

lemma *is-reduced-GB-singleton:* $\text{is-reduced-GB } \{f\} \longleftrightarrow \text{lc } f = 1$
<proof>

lemma *reduced-GB-empty:* $\text{reduced-GB } \{\} = \{\}$
<proof>

lemma *reduced-GB-singleton:* $\text{reduced-GB } \{f\} = (\text{if } f = 0 \text{ then } \{\} \text{ else } \{\text{monic } f\})$
<proof>

lemma *ex-unique-reduced-GB-finite:* $\text{finite } F \implies (\exists! G. \text{finite } G \wedge \text{is-reduced-GB } G \wedge \text{pmdl } G = \text{pmdl } F)$
<proof>

lemma *finite-reduced-GB-finite:* $\text{finite } F \implies \text{finite}(\text{reduced-GB } F)$
<proof>

lemma *reduced-GB-is-reduced-GB-finite:* $\text{finite } F \implies \text{is-reduced-GB}(\text{reduced-GB } F)$
<proof>

lemma *reduced-GB-is-GB-finite:* $\text{finite } F \implies \text{is-Groebner-basis}(\text{reduced-GB } F)$

<proof>

lemma *reduced-GB-is-auto-reduced-finite*: *finite F* \implies *is-auto-reduced (reduced-GB F)*

<proof>

lemma *reduced-GB-is-monic-set-finite*: *finite F* \implies *is-monic-set (reduced-GB F)*

<proof>

lemma *reduced-GB-nonzero-finite*: *finite F* \implies $0 \notin$ *reduced-GB F*

<proof>

lemma *reduced-GB-pmdl-finite*: *finite F* \implies *pmdl (reduced-GB F) = pmdl F*

<proof>

lemma *reduced-GB-unique-finite*: *finite F* \implies *is-reduced-GB G* \implies *pmdl G = pmdl F* \implies *reduced-GB F = G*

<proof>

end

13.2.5 Properties of the Reduced Gröbner Basis of an Ideal

context *gd-powerprod*

begin

lemma *ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set*:

assumes *dickson-grading d* **and** $F \subseteq$ *punit.dgrad-p-set d m*

shows *ideal F = UNIV* \longleftrightarrow *punit.reduced-GB F = {1}*

<proof>

lemmas *ideal-eq-UNIV-iff-reduced-GB-eq-one-finite =*

ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set[OF dickson-grading-dgrad-dummy punit.dgrad-p-set-exhaust-expl]

end

13.2.6 Context *od-term*

context *od-term*

begin

lemmas *ex-unique-reduced-GB =*

ex-unique-reduced-GB-dgrad-p-set'[OF dickson-grading-zero - subset-dgrad-p-set-zero]

lemmas *finite-reduced-GB =*

finite-reduced-GB-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]

lemmas *reduced-GB-is-reduced-GB =*

reduced-GB-is-reduced-GB-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]

lemmas *reduced-GB-is-GB =*

reduced-GB-is-GB-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]

```

lemmas reduced-GB-is-auto-reduced =
  reduced-GB-is-auto-reduced-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]
lemmas reduced-GB-is-monic-set =
  reduced-GB-is-monic-set-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]
lemmas reduced-GB-nonzero =
  reduced-GB-nonzero-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]
lemmas reduced-GB-pmdl =
  reduced-GB-pmdl-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]
lemmas reduced-GB-unique =
  reduced-GB-unique-dgrad-p-set[OF dickson-grading-zero - subset-dgrad-p-set-zero]

end

end

```

14 Sample Computations of Reduced Gröbner Bases

```

theory Reduced-GB-Examples
  imports Buchberger Reduced-GB Polynomials.MPoly-Type-Class-OAlist Code-Target-Rat
begin

```

```

context gd-term
begin

```

```

definition rgb :: ('t  $\Rightarrow_0$  'b) list  $\Rightarrow$  ('t  $\Rightarrow_0$  'b::field) list
  where rgb bs = comp-red-monic-basis (map fst (gb (map ( $\lambda$ b. (b, ())) bs) ()))

```

```

definition rgb-punit :: ('a  $\Rightarrow_0$  'b) list  $\Rightarrow$  ('a  $\Rightarrow_0$  'b::field) list
  where rgb-punit bs = punit.comp-red-monic-basis (map fst (gb-punit (map ( $\lambda$ b.
  (b, ())) bs) ()))

```

```

lemma compute-trd-aux [code]:
  trd-aux fs p r =
    (if is-zero p then
     r
    else
     case find-adds fs (lt p) of
       None  $\Rightarrow$  trd-aux fs (tail p) (plus-monomial-less r (lc p) (lt p))
     | Some f  $\Rightarrow$  trd-aux fs (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail
f)) r
    )
  <proof>

```

```

end

```

We only consider scalar polynomials here, but vector-polynomials could be handled, too.

```

global-interpretation punit': gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit
cmp-term

```

```

rewrites punit.adds-term = (adds)
and punit.pp-of-term = ( $\lambda x. x$ )
and punit.component-of-term = ( $\lambda-. ()$ )
and punit.monom-mult = monom-mult-punit
and punit.mult-scalar = mult-scalar-punit
and punit'.punit.min-term = min-term-punit
and punit'.punit.lt = lt-punit cmp-term
and punit'.punit.lc = lc-punit cmp-term
and punit'.punit.tail = tail-punit cmp-term
and punit'.punit.ord-p = ord-p-punit cmp-term
and punit'.punit.ord-strict-p = ord-strict-p-punit cmp-term
for cmp-term :: ('a::nat, 'b::{nat,add-wellorder}) pp nat-term-order

```

```

defines find-adds-punit = punit'.punit.find-adds
and trd-aux-punit = punit'.punit.trd-aux
and trd-punit = punit'.punit.trd
and spoly-punit = punit'.punit.spoly
and count-const-lt-components-punit = punit'.punit.count-const-lt-components
and count-rem-components-punit = punit'.punit.count-rem-components
and const-lt-component-punit = punit'.punit.const-lt-component
and full-gb-punit = punit'.punit.full-gb
and add-pairs-single-sorted-punit = punit'.punit.add-pairs-single-sorted
and add-pairs-punit = punit'.punit.add-pairs
and canon-pair-order-aux-punit = punit'.punit.canon-pair-order-aux
and canon-basis-order-punit = punit'.punit.canon-basis-order
and new-pairs-sorted-punit = punit'.punit.new-pairs-sorted
and product-crit-punit = punit'.punit.product-crit
and chain-ncrit-punit = punit'.punit.chain-ncrit
and chain-ocrit-punit = punit'.punit.chain-ocrit
and apply-icrit-punit = punit'.punit.apply-icrit
and apply-ncrit-punit = punit'.punit.apply-ncrit
and apply-ocrit-punit = punit'.punit.apply-ocrit
and trdsp-punit = punit'.punit.trdsp
and gb-sel-punit = punit'.punit.gb-sel
and gb-red-aux-punit = punit'.punit.gb-red-aux
and gb-red-punit = punit'.punit.gb-red
and gb-aux-punit = punit'.punit.gb-aux-punit
and gb-punit = punit'.punit.gb-punit — Faster, because incorporates product
criterion.
and comp-min-basis-punit = punit'.punit.comp-min-basis
and comp-red-basis-aux-punit = punit'.punit.comp-red-basis-aux
and comp-red-basis-punit = punit'.punit.comp-red-basis
and monic-punit = punit'.punit.monic
and comp-red-monic-basis-punit = punit'.punit.comp-red-monic-basis
and rgb-punit = punit'.punit.rgb-punit
<proof>

```

lemma *compute-spoly-punit* [code]:

spoly-punit to p q = (let t1 = lt-punit to p; t2 = lt-punit to q; l = lcs t1 t2 in

(*monom-mult-punit (1 / lc-punit to p) (l - t1) p*) - (*monom-mult-punit (1 / lc-punit to q) (l - t2) q*)
 ⟨*proof*⟩

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

experiment begin interpretation *trivariate₀-rat* ⟨*proof*⟩

lemma

rgb-punit DRLEX

[
 $X^3 - X * Y * Z^2,$
 $Y^2 * Z - 1$
] =
 [
 $X^3 * Y - X * Z,$
 $-(X^3) + X * Y * Z^2,$
 $Y^2 * Z - 1,$
 $-(X * Z^3) + X^5$
]

⟨*proof*⟩

lemma

rgb-punit DRLEX

[
 $X^2 + Y^2 + Z^2 - 1,$
 $X * Y - Z - 1,$
 $Y^2 + X,$
 $Z^2 + X$
] =
 [
 1
]

⟨*proof*⟩

Note: The above computations have been cross-checked with Mathematica 11.1.

end

end

15 Macaulay Matrices

theory *Macaulay-Matrix*

imports *More-MPoly-Type-Class Jordan-Normal-Form.Gauss-Jordan-Elimination*

begin

We build upon vectors and matrices represented by dimension and characteristic function, because later on we need to quantify the dimensions of certain matrices existentially. This is not possible (at least not easily possible) with a type-based approach, as in HOL-Multivariate Analysis.

15.1 More about Vectors

lemma *vec-of-list-alt*: $vec\text{-of-list } xs = vec\ (length\ xs)\ (nth\ xs)$
<proof>

lemma *vec-cong*:
assumes $n = m$ **and** $\bigwedge i. i < m \implies f\ i = g\ i$
shows $vec\ n\ f = vec\ m\ g$
<proof>

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

lemma *vec-scalar-mult-fun*: $vec\ n\ (\lambda x. c * f\ x) = c \cdot_v\ vec\ n\ f$
<proof>

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

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

lemma *dim-resize-vec[simp]*: $dim\text{-vec } (resize\text{-vec } n\ v) = n$
<proof>

lemma *resize-vec-carrier*: $resize\text{-vec } n\ v \in carrier\text{-vec } n$
<proof>

lemma *resize-vec-dim[simp]*: $resize\text{-vec } (dim\text{-vec } v)\ v = v$
<proof>

lemma *resize-vec-index*:
assumes $i < n$
shows $resize\text{-vec } n\ v\ \$\ i = v\ \$\ i$
<proof>

lemma *mult-mat-vec-resize*:
 $v \cdot_v A = (resize\text{-vec } (dim\text{-row } A)\ v) \cdot_v A$
<proof>

lemma *assoc-mult-vec-mat*:
assumes $v \in carrier\text{-vec } n1$ **and** $A \in carrier\text{-mat } n1\ n2$ **and** $B \in carrier\text{-mat}$

n2 n3

shows $v \cdot_v (A * B) = (v \cdot_v A) \cdot_v B$
<proof>

lemma *mult-vec-mat-transpose*:

assumes $\dim\text{-vec } v = \dim\text{-row } A$

shows $v \cdot_v A = (\text{transpose-mat } A) \cdot_v (v :: 'a :: \text{comm-semiring-0 vec})$
<proof>

15.2 More about Matrices

definition *nzrows* :: $'a :: \text{zero mat} \Rightarrow 'a \text{ vec list}$

where $\text{nzrows } A = \text{filter } (\lambda r. r \neq 0_v (\dim\text{-col } A)) (\text{rows } A)$

definition *row-space* :: $'a \text{ mat} \Rightarrow 'a :: \text{semiring-0 vec set}$

where $\text{row-space } A = (\lambda v. \text{mult-vec-mat } v A) ` (\text{carrier-vec } (\dim\text{-row } A))$

definition *row-echelon* :: $'a \text{ mat} \Rightarrow 'a :: \text{field mat}$

where $\text{row-echelon } A = \text{fst } (\text{gauss-jordan } A (1_m (\dim\text{-row } A)))$

15.2.1 *nzrows*

lemma *length-nzrows*: $\text{length } (\text{nzrows } A) \leq \dim\text{-row } A$

<proof>

lemma *set-nzrows*: $\text{set } (\text{nzrows } A) = \text{set } (\text{rows } A) - \{0_v (\dim\text{-col } A)\}$

<proof>

lemma *nzrows-nth-not-zero*:

assumes $i < \text{length } (\text{nzrows } A)$

shows $\text{nzrows } A ! i \neq 0_v (\dim\text{-col } A)$

<proof>

15.2.2 *row-space*

lemma *row-spaceI*:

assumes $x = v \cdot_v A$

shows $x \in \text{row-space } A$

<proof>

lemma *row-spaceE*:

assumes $x \in \text{row-space } A$

obtains v **where** $v \in \text{carrier-vec } (\dim\text{-row } A)$ **and** $x = v \cdot_v A$

<proof>

lemma *row-space-alt*: $\text{row-space } A = \text{range } (\lambda v. \text{mult-vec-mat } v A)$

<proof>

lemma *row-space-mult*:

assumes $A \in \text{carrier-mat } nr \ nc$ **and** $B \in \text{carrier-mat } nr \ nr$

shows $\text{row-space } (B * A) \subseteq \text{row-space } A$
(proof)

lemma *row-space-mult-unit*:

assumes $P \in \text{Units } (\text{ring-mat } \text{TYPE}('a::\text{semiring-1}) (\text{dim-row } A) b)$
shows $\text{row-space } (P * A) = \text{row-space } A$
(proof)

15.2.3 row-echelon

lemma *row-eq-zero-iff-pivot-fun*:

assumes $\text{pivot-fun } A f (\text{dim-col } A)$ **and** $i < \text{dim-row } (A::'a::\text{zero-neq-one mat})$
shows $(\text{row } A \ i = 0_v (\text{dim-col } A)) \longleftrightarrow (f \ i = \text{dim-col } A)$
(proof)

lemma *row-not-zero-iff-pivot-fun*:

assumes $\text{pivot-fun } A f (\text{dim-col } A)$ **and** $i < \text{dim-row } (A::'a::\text{zero-neq-one mat})$
shows $(\text{row } A \ i \neq 0_v (\text{dim-col } A)) \longleftrightarrow (f \ i < \text{dim-col } A)$
(proof)

lemma *pivot-fun-stabilizes*:

assumes $\text{pivot-fun } A f \ nc$ **and** $i1 \leq i2$ **and** $i2 < \text{dim-row } A$ **and** $nc \leq f \ i1$
shows $f \ i2 = nc$
(proof)

lemma *pivot-fun-mono-strict*:

assumes $\text{pivot-fun } A f \ nc$ **and** $i1 < i2$ **and** $i2 < \text{dim-row } A$ **and** $f \ i1 < nc$
shows $f \ i1 < f \ i2$
(proof)

lemma *pivot-fun-mono*:

assumes $\text{pivot-fun } A f \ nc$ **and** $i1 \leq i2$ **and** $i2 < \text{dim-row } A$
shows $f \ i1 \leq f \ i2$
(proof)

lemma *row-echelon-carrier*:

assumes $A \in \text{carrier-mat } nr \ nc$
shows $\text{row-echelon } A \in \text{carrier-mat } nr \ nc$
(proof)

lemma *dim-row-echelon[simp]*:

shows $\text{dim-row } (\text{row-echelon } A) = \text{dim-row } A$ **and** $\text{dim-col } (\text{row-echelon } A) = \text{dim-col } A$
(proof)

lemma *row-echelon-transform*:

obtains P **where** $P \in \text{Units } (\text{ring-mat } \text{TYPE}('a::\text{field}) (\text{dim-row } A) b)$ **and**
 $\text{row-echelon } A = P * A$
(proof)

lemma *row-space-row-echelon[simp]*: $\text{row-space } (\text{row-echelon } A) = \text{row-space } A$
 ⟨proof⟩

lemma *row-echelon-pivot-fun*:
obtains f **where** $\text{pivot-fun } (\text{row-echelon } A) f (\text{dim-col } (\text{row-echelon } A))$
 ⟨proof⟩

lemma *distinct-nzrows-row-echelon*: $\text{distinct } (\text{nzrows } (\text{row-echelon } A))$
 ⟨proof⟩

15.3 Converting Between Polynomials and Macaulay Matrices

definition *poly-to-row* :: $'a \text{ list} \Rightarrow ('a \Rightarrow_0 'b::\text{zero}) \Rightarrow 'b \text{ vec}$ **where**
 $\text{poly-to-row } ts \ p = \text{vec-of-list } (\text{map } (\text{lookup } p) \ ts)$

definition *polys-to-mat* :: $'a \text{ list} \Rightarrow ('a \Rightarrow_0 'b::\text{zero}) \text{ list} \Rightarrow 'b \text{ mat}$ **where**
 $\text{polys-to-mat } ts \ ps = \text{mat-of-rows } (\text{length } ts) (\text{map } (\text{poly-to-row } ts) \ ps)$

definition *list-to-fun* :: $'a \text{ list} \Rightarrow ('b::\text{zero}) \text{ list} \Rightarrow 'a \Rightarrow 'b$ **where**
 $\text{list-to-fun } ts \ cs \ t = (\text{case map-of } (\text{zip } ts \ cs) \ t \text{ of } \text{Some } c \Rightarrow c \mid \text{None} \Rightarrow 0)$

definition *list-to-poly* :: $'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \Rightarrow_0 'b::\text{zero})$ **where**
 $\text{list-to-poly } ts \ cs = \text{Abs-poly-mapping } (\text{list-to-fun } ts \ cs)$

definition *row-to-poly* :: $'a \text{ list} \Rightarrow 'b \text{ vec} \Rightarrow ('a \Rightarrow_0 'b::\text{zero})$ **where**
 $\text{row-to-poly } ts \ r = \text{list-to-poly } ts (\text{list-of-vec } r)$

definition *mat-to-polys* :: $'a \text{ list} \Rightarrow 'b \text{ mat} \Rightarrow ('a \Rightarrow_0 'b::\text{zero}) \text{ list}$ **where**
 $\text{mat-to-polys } ts \ A = \text{map } (\text{row-to-poly } ts) (\text{rows } A)$

lemma *dim-poly-to-row*: $\text{dim-vec } (\text{poly-to-row } ts \ p) = \text{length } ts$
 ⟨proof⟩

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

context *term-powerprod*
begin

lemma *poly-to-row-scalar-mult*:
assumes $\text{keys } p \subseteq \text{set } ts$
shows $\text{row-to-poly } ts \ (c \cdot_v (\text{poly-to-row } ts \ p)) = c \cdot p$
 ⟨proof⟩

lemma *poly-to-row-to-poly*:

assumes $keys\ p \subseteq set\ ts$
shows $row\text{-}to\text{-}poly\ ts\ (poly\text{-}to\text{-}row\ ts\ p) = (p::'t \Rightarrow_0 'b::semiring\text{-}1)$
 $\langle proof \rangle$

lemma *lookup-list-to-poly*: $lookup\ (list\text{-}to\text{-}poly\ ts\ cs) = list\text{-}to\text{-}fun\ ts\ cs$
 $\langle proof \rangle$

lemma *list-to-fun-Nil* [simp]: $list\text{-}to\text{-}fun\ []\ cs = 0$
 $\langle proof \rangle$

lemma *list-to-poly-Nil* [simp]: $list\text{-}to\text{-}poly\ []\ cs = 0$
 $\langle proof \rangle$

lemma *row-to-poly-Nil* [simp]: $row\text{-}to\text{-}poly\ []\ r = 0$
 $\langle proof \rangle$

lemma *lookup-row-to-poly*:
assumes *distinct* ts **and** $dim\text{-}vec\ r = length\ ts$ **and** $i < length\ ts$
shows $lookup\ (row\text{-}to\text{-}poly\ ts\ r)\ (ts\ !\ i) = r\ \$\ i$
 $\langle proof \rangle$

lemma *keys-row-to-poly*: $keys\ (row\text{-}to\text{-}poly\ ts\ r) \subseteq set\ ts$
 $\langle proof \rangle$

lemma *lookup-row-to-poly-not-zeroE*:
assumes $lookup\ (row\text{-}to\text{-}poly\ ts\ r)\ t \neq 0$
obtains i **where** $i < length\ ts$ **and** $t = ts\ !\ i$
 $\langle proof \rangle$

lemma *row-to-poly-zero* [simp]: $row\text{-}to\text{-}poly\ ts\ (0_v\ (length\ ts)) = (0::'t \Rightarrow_0 'b::zero)$
 $\langle proof \rangle$

lemma *row-to-poly-zeroD*:
assumes *distinct* ts **and** $dim\text{-}vec\ r = length\ ts$ **and** $row\text{-}to\text{-}poly\ ts\ r = 0$
shows $r = 0_v\ (length\ ts)$
 $\langle proof \rangle$

lemma *row-to-poly-inj*:
assumes *distinct* ts **and** $dim\text{-}vec\ r1 = length\ ts$ **and** $dim\text{-}vec\ r2 = length\ ts$
and $row\text{-}to\text{-}poly\ ts\ r1 = row\text{-}to\text{-}poly\ ts\ r2$
shows $r1 = r2$
 $\langle proof \rangle$

lemma *row-to-poly-vec-plus*:
assumes *distinct* ts **and** $length\ ts = n$
shows $row\text{-}to\text{-}poly\ ts\ (vec\ n\ (f1 + f2)) = row\text{-}to\text{-}poly\ ts\ (vec\ n\ f1) + row\text{-}to\text{-}poly\ ts\ (vec\ n\ f2)$
 $\langle proof \rangle$

lemma *row-to-poly-vec-sum*:

assumes *distinct ts and length ts = n*

shows $\text{row-to-poly } ts \ (\text{vec } n \ (\lambda j. \sum_{i \in I}. f \ i \ j)) = ((\sum_{i \in I}. \text{row-to-poly } ts \ (\text{vec } n \ (f \ i)))::'t \Rightarrow_0 \ 'b::\text{comm-monoid-add})$

<proof>

lemma *row-to-poly-smult*:

assumes *distinct ts and dim-vec r = length ts*

shows $\text{row-to-poly } ts \ (c \cdot_v \ r) = c \cdot (\text{row-to-poly } ts \ r)$

<proof>

lemma *poly-to-row-Nil [simp]*: $\text{poly-to-row } [] \ p = \text{vec } 0 \ f$

<proof>

lemma *polys-to-mat-Nil [simp]*: $\text{polys-to-mat } ts \ [] = \text{mat } 0 \ (\text{length } ts) \ f$

<proof>

lemma *dim-row-polys-to-mat[simp]*: $\text{dim-row } (\text{polys-to-mat } ts \ ps) = \text{length } ps$

<proof>

lemma *dim-col-polys-to-mat[simp]*: $\text{dim-col } (\text{polys-to-mat } ts \ ps) = \text{length } ts$

<proof>

lemma *polys-to-mat-index*:

assumes $i < \text{length } ps$ **and** $j < \text{length } ts$

shows $(\text{polys-to-mat } ts \ ps) \ \$\$ \ (i, j) = \text{lookup } (ps \ ! \ i) \ (ts \ ! \ j)$

<proof>

lemma *row-polys-to-mat*:

assumes $i < \text{length } ps$

shows $\text{row } (\text{polys-to-mat } ts \ ps) \ i = \text{poly-to-row } ts \ (ps \ ! \ i)$

<proof>

lemma *col-polys-to-mat*:

assumes $j < \text{length } ts$

shows $\text{col } (\text{polys-to-mat } ts \ ps) \ j = \text{vec-of-list } (\text{map } (\lambda p. \text{lookup } p \ (ts \ ! \ j)) \ ps)$

<proof>

lemma *length-mat-to-polys[simp]*: $\text{length } (\text{mat-to-polys } ts \ A) = \text{dim-row } A$

<proof>

lemma *mat-to-polys-nth*:

assumes $i < \text{dim-row } A$

shows $(\text{mat-to-polys } ts \ A) \ ! \ i = \text{row-to-poly } ts \ (\text{row } A \ i)$

<proof>

lemma *Keys-mat-to-polys*: $\text{Keys } (\text{set } (\text{mat-to-polys } ts \ A)) \subseteq \text{set } ts$

<proof>

lemma *polys-to-mat-to-polys*:

assumes $Keys (set ps) \subseteq set ts$

shows $mat\text{-}to\text{-}polys\ ts (polys\text{-}to\text{-}mat\ ts\ ps) = (ps::('t \Rightarrow_0 'b::semiring-1)\ list)$

$\langle proof \rangle$

lemma *mat-to-polys-to-mat*:

assumes *distinct ts and length ts = dim-col A*

shows $(polys\text{-}to\text{-}mat\ ts (mat\text{-}to\text{-}polys\ ts\ A)) = A$

$\langle proof \rangle$

15.4 Properties of Macaulay Matrices

lemma *row-to-poly-vec-times*:

assumes *distinct ts and length ts = dim-col A*

shows $row\text{-}to\text{-}poly\ ts (v\ v^* A) = ((\sum i=0..<dim\text{-}row\ A. (v\ \$\ i) \cdot (row\text{-}to\text{-}poly\ ts (row\ A\ i))))::('t \Rightarrow_0 'b::comm\text{-}semiring-0)$

$\langle proof \rangle$

lemma *vec-times-polys-to-mat*:

assumes $Keys (set ps) \subseteq set ts$ **and** $v \in carrier\text{-}vec (length\ ps)$

shows $row\text{-}to\text{-}poly\ ts (v\ v^* (polys\text{-}to\text{-}mat\ ts\ ps)) = (\sum (c, p) \leftarrow zip (list\text{-}of\text{-}vec\ v) ps. c \cdot p)$

(**is** $?l = ?r$)

$\langle proof \rangle$

lemma *row-space-subset-phull*:

assumes $Keys (set ps) \subseteq set ts$

shows $row\text{-}to\text{-}poly\ ts \text{ ' row-space } (polys\text{-}to\text{-}mat\ ts\ ps) \subseteq phull (set\ ps)$

(**is** $?r \subseteq ?h$)

$\langle proof \rangle$

lemma *phull-subset-row-space*:

assumes $Keys (set ps) \subseteq set ts$

shows $phull (set\ ps) \subseteq row\text{-}to\text{-}poly\ ts \text{ ' row-space } (polys\text{-}to\text{-}mat\ ts\ ps)$

(**is** $?h \subseteq ?r$)

$\langle proof \rangle$

lemma *row-space-eq-phull*:

assumes $Keys (set ps) \subseteq set ts$

shows $row\text{-}to\text{-}poly\ ts \text{ ' row-space } (polys\text{-}to\text{-}mat\ ts\ ps) = phull (set\ ps)$

$\langle proof \rangle$

lemma *row-space-row-echelon-eq-phull*:

assumes $Keys (set ps) \subseteq set ts$

shows $row\text{-}to\text{-}poly\ ts \text{ ' row-space } (row\text{-}echelon (polys\text{-}to\text{-}mat\ ts\ ps)) = phull (set\ ps)$

$\langle proof \rangle$

lemma *phull-row-echelon*:

assumes $Keys (set ps) \subseteq set ts$ **and** $distinct ts$
shows $phull (set (mat-to-polys ts (row-echelon (polys-to-mat ts ps)))) = phull$
 $(set ps)$
 $\langle proof \rangle$

lemma *pmdl-row-echelon*:

assumes $Keys (set ps) \subseteq set ts$ **and** $distinct ts$
shows $pmdl (set (mat-to-polys ts (row-echelon (polys-to-mat ts ps)))) = pmdl$
 $(set ps)$
(is $?l = ?r$
 $\langle proof \rangle$

end

context *ordered-term*

begin

lemma *lt-row-to-poly-pivot-fun*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** $pivot-fun A f (dim-col$
 $A)$
and $i < dim-row A$ **and** $f i < dim-col A$
shows $lt ((mat-to-polys (pps-to-list S) A) ! i) = (pps-to-list S) ! (f i)$
 $\langle proof \rangle$

lemma *lc-row-to-poly-pivot-fun*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** $pivot-fun A f (dim-col$
 $A)$
and $i < dim-row A$ **and** $f i < dim-col A$
shows $lc ((mat-to-polys (pps-to-list S) A) ! i) = 1$
 $\langle proof \rangle$

lemma *lt-row-to-poly-pivot-fun-less*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** $pivot-fun A f (dim-col$
 $A)$
and $i1 < i2$ **and** $i2 < dim-row A$ **and** $f i1 < dim-col A$ **and** $f i2 < dim-col A$
shows $(pps-to-list S) ! (f i2) \prec_t (pps-to-list S) ! (f i1)$
 $\langle proof \rangle$

lemma *lt-row-to-poly-pivot-fun-eqD*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** $pivot-fun A f (dim-col$
 $A)$
and $i1 < dim-row A$ **and** $i2 < dim-row A$ **and** $f i1 < dim-col A$ **and** $f i2 <$
 $dim-col A$
and $(pps-to-list S) ! (f i1) = (pps-to-list S) ! (f i2)$
shows $i1 = i2$
 $\langle proof \rangle$

lemma *lt-row-to-poly-pivot-in-keysD*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** $pivot-fun A f (dim-col$

A)
and $i1 < \text{dim-row } A$ **and** $i2 < \text{dim-row } A$ **and** $f\ i1 < \text{dim-col } A$
and $(\text{pps-to-list } S) ! (f\ i1) \in \text{keys } ((\text{mat-to-polys } (\text{pps-to-list } S)\ A) ! i2)$
shows $i1 = i2$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *in-Macaulay-listE*:
assumes $p \in \text{set } (\text{Macaulay-list } ps)$
and *pivot-fun* $(\text{row-echelon } (\text{Macaulay-mat } ps))\ f\ (\text{dim-col } (\text{row-echelon } (\text{Macaulay-mat}$
 $ps)))$
obtains i **where** $i < \text{dim-row } (\text{row-echelon } (\text{Macaulay-mat } ps))$
and $p = (\text{mat-to-polys } (\text{Keys-to-list } ps) (\text{row-echelon } (\text{Macaulay-mat } ps))) ! i$
and $f\ i < \text{dim-col } (\text{row-echelon } (\text{Macaulay-mat } ps))$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

end

end

16 Faugère's F4 Algorithm

theory *F4*
 imports *Macaulay-Matrix Algorithm-Schema*
 begin

This theory implements Faugère's F4 algorithm based on *gd-term.gb-schema-direct*.

16.1 Symbolic Preprocessing

context *gd-term*
 begin

definition *sym-preproc-aux-term1* :: $('a \Rightarrow nat) \Rightarrow (((t \Rightarrow_0 'b) list \times 't list \times 't list \times (t \Rightarrow_0 'b) list) \times ((t \Rightarrow_0 'b) list \times 't list \times 't list \times (t \Rightarrow_0 'b) list)) set$
 where $sym-preproc-aux-term1\ d = \{((gs1, ks1, ts1, fs1), (gs2::(t \Rightarrow_0 'b) list, ks2, ts2, fs2)). \exists t2 \in set\ ts2. \forall t1 \in set\ ts1. t1 \prec_t t2\}$

definition *sym-preproc-aux-term2* :: ('a ⇒ nat) ⇒ (((t ⇒₀ 'b)::zero) list × 't list × 't list × (t ⇒₀ 'b) list) × ((t ⇒₀ 'b) list × 't list × 't list × (t ⇒₀ 'b) list) set
where *sym-preproc-aux-term2* d =
 {((gs1, ks1, ts1, fs1), (gs2::(t ⇒₀ 'b) list, ks2, ts2, fs2)). gs1 = gs2 ∧ dgrad-set-le d (pp-of-term ' set ts1) (pp-of-term ' (Keys (set gs2) ∪ set ts2))}

definition *sym-preproc-aux-term*
where *sym-preproc-aux-term* d = *sym-preproc-aux-term1* d ∩ *sym-preproc-aux-term2* d

lemma *wfp-on-ord-term-strict*:
assumes *dickson-grading* d
shows *wfp-on* (<_t) (pp-of-term - ' dgrad-set d m)
 <proof>

lemma *sym-preproc-aux-term1-wf-on*:
assumes *dickson-grading* d
shows *wfp-on* (λx y. (x, y) ∈ *sym-preproc-aux-term1* d) {x. set (fst (snd (snd x))) ⊆ pp-of-term - ' dgrad-set d m}
 <proof>

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

primrec *sym-preproc-addnew* :: (t ⇒₀ 'b)::semiring-1) list ⇒ 't list ⇒ (t ⇒₀ 'b) list ⇒ 't ⇒
 (t list × (t ⇒₀ 'b) list) **where**
sym-preproc-addnew [] vs fs - = (vs, fs)|
sym-preproc-addnew (g # gs) vs fs v =
 (if lt g adds_t v then
 (let f = monom-mult 1 (pp-of-term v - lp g) g in
sym-preproc-addnew gs (merge-wrt (>_t) vs (keys-to-list (tail f))) (insert-list f fs) v
)
 else
sym-preproc-addnew gs vs fs v
)

lemma *fst-sym-preproc-addnew-less*:
assumes ∧u. u ∈ set vs ⇒ u <_t v
and u ∈ set (fst (*sym-preproc-addnew* gs vs fs v))
shows u <_t v
 <proof>

lemma *fst-sym-preproc-addnew-dgrad-set-le*:

assumes *dickson-grading* *d*

shows *dgrad-set-le* *d* (*pp-of-term* ' *set* (*fst* (*sym-preproc-addnew* *gs vs fs v*)))
(*pp-of-term* ' (*Keys* (*set gs*) \cup *insert v* (*set vs*)))
<*proof*>

lemma *components-fst-sym-preproc-addnew-subset*:

component-of-term ' *set* (*fst* (*sym-preproc-addnew* *gs vs fs v*)) \subseteq *component-of-term*
' (*Keys* (*set gs*) \cup *insert v* (*set vs*))
<*proof*>

lemma *fst-sym-preproc-addnew-superset*: *set vs* \subseteq *set* (*fst* (*sym-preproc-addnew* *gs vs fs v*))
<*proof*>

lemma *snd-sym-preproc-addnew-superset*: *set fs* \subseteq *set* (*snd* (*sym-preproc-addnew* *gs vs fs v*))
<*proof*>

lemma *in-snd-sym-preproc-addnewE*:

assumes $p \in \text{set } (\text{snd } (\text{sym-preproc-addnew } gs \text{ vs } fs \ v))$

assumes 1: $p \in \text{set } fs \implies \text{thesis}$

assumes 2: $\bigwedge g \ s. \ g \in \text{set } gs \implies p = \text{monom-mult } 1 \ s \ g \implies \text{thesis}$

shows *thesis*

<*proof*>

lemma *sym-preproc-addnew-pmdl*:

pmdl (*set gs* \cup *set* (*snd* (*sym-preproc-addnew* *gs vs fs v*))) = *pmdl* (*set gs* \cup *set*
fs)

(**is** *pmdl* (*set gs* \cup ?*l*) = ?*r*)

<*proof*>

lemma *Keys-snd-sym-preproc-addnew*:

Keys (*set* (*snd* (*sym-preproc-addnew* *gs vs fs v*))) \cup *insert v* (*set vs*) =

Keys (*set fs*) \cup *insert v* (*set* (*fst* (*sym-preproc-addnew* *gs vs fs*:('t \Rightarrow_0 'b::semiring-1-no-zero-divisors)
list) *v*)))

<*proof*>

lemma *sym-preproc-addnew-complete*:

assumes $g \in \text{set } gs$ **and** $lt \ g \ \text{adds}_t \ v$

shows *monom-mult* 1 (*pp-of-term* $v - lp \ g$) $g \in \text{set } (\text{snd } (\text{sym-preproc-addnew}$
gs vs fs v))

<*proof*>

function *sym-preproc-aux* :: ('t \Rightarrow_0 'b::semiring-1) *list* \Rightarrow 't *list* \Rightarrow ('t *list* \times ('t
 \Rightarrow_0 'b) *list*) \Rightarrow

('t *list* \times ('t \Rightarrow_0 'b) *list*) **where**

sym-preproc-aux *gs ks* (*vs*, *fs*) =

(*if* *vs* = [] *then*

(ks, fs)
else
 $let\ v = ord-term-lin.max-list\ vs; vs' = removeAll\ v\ vs\ in$
 $sym-preproc-aux\ gs\ (ks\ @\ [v])\ (sym-preproc-addnew\ gs\ vs'\ fs\ v)$
 $)$
 $\langle proof \rangle$
termination $\langle proof \rangle$

lemma *sym-preproc-aux-Nil*: $sym-preproc-aux\ gs\ ks\ ([], fs) = (ks, fs)$
 $\langle proof \rangle$

lemma *sym-preproc-aux-sorted*:
assumes *sorted-wrt* $(\succ_t)\ (v \# vs)$
shows $sym-preproc-aux\ gs\ ks\ (v \# vs, fs) = sym-preproc-aux\ gs\ (ks\ @\ [v])$
 $(sym-preproc-addnew\ gs\ vs\ fs\ v)$
 $\langle proof \rangle$

lemma *sym-preproc-aux-induct* [*consumes 0, case-names base rec*]:
assumes *base*: $\bigwedge ks\ fs.\ P\ ks\ []\ fs\ (ks, fs)$
and *rec*: $\bigwedge ks\ vs\ fs\ v\ vs'.\ vs \neq [] \implies v = ord-term-lin.Max\ (set\ vs) \implies vs' =$
 $removeAll\ v\ vs \implies$
 $P\ (ks\ @\ [v])\ (fst\ (sym-preproc-addnew\ gs\ vs'\ fs\ v))\ (snd\ (sym-preproc-addnew$
 $gs\ vs'\ fs\ v))$
 $(sym-preproc-aux\ gs\ (ks\ @\ [v])\ (sym-preproc-addnew\ gs\ vs'\ fs\ v))$
 \implies
 $P\ ks\ vs\ fs\ (sym-preproc-aux\ gs\ (ks\ @\ [v])\ (sym-preproc-addnew\ gs\ vs'$
 $fs\ v))$
shows $P\ ks\ vs\ fs\ (sym-preproc-aux\ gs\ ks\ (vs, fs))$
 $\langle proof \rangle$

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

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

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

lemma *in-snd-sym-preproc-auxE*:
assumes $p \in set\ (snd\ (sym-preproc-aux\ gs\ ks\ (vs, fs)))$
assumes *1*: $p \in set\ fs \implies thesis$

assumes $2: \bigwedge g t. g \in \text{set } gs \implies p = \text{monom-mult } 1 t g \implies \text{thesis}$
shows *thesis*
 ⟨*proof*⟩

lemma *snd-sym-preproc-aux-pmdl*:

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

lemma *snd-sym-preproc-aux-dgrad-set-le*:

assumes *dickson-grading* d **and** $\text{set } vs \subseteq \text{Keys } (\text{set } (fs::('t \Rightarrow_0 'b::\text{semiring-1-no-zero-divisors}) \text{ list}))$

shows $d\text{grad-set-le } d (\text{pp-of-term } ' \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc-aux } gs \text{ ks } (vs, fs)))))) (\text{pp-of-term } ' \text{Keys } (\text{set } gs \cup \text{set } fs))$
 ⟨*proof*⟩

lemma *components-snd-sym-preproc-aux-subset*:

assumes $\text{set } vs \subseteq \text{Keys } (\text{set } (fs::('t \Rightarrow_0 'b::\text{semiring-1-no-zero-divisors}) \text{ list}))$

shows $\text{component-of-term } ' \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc-aux } gs \text{ ks } (vs, fs)))) \subseteq \text{component-of-term } ' \text{Keys } (\text{set } gs \cup \text{set } fs)$

⟨*proof*⟩

lemma *snd-sym-preproc-aux-complete*:

assumes $\bigwedge u' g'. u' \in \text{Keys } (\text{set } fs) \implies u' \notin \text{set } vs \implies g' \in \text{set } gs \implies \text{lt } g' \text{ adds}_t u' \implies$

$\text{monom-mult } 1 (\text{pp-of-term } u' - \text{lp } g') g' \in \text{set } fs$

assumes $u \in \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc-aux } gs \text{ ks } (vs, fs))))$ **and** $g \in \text{set } gs$ **and** $\text{lt } g \text{ adds}_t u$

shows $\text{monom-mult } (1::'b::\text{semiring-1-no-zero-divisors}) (\text{pp-of-term } u - \text{lp } g) g \in$

$\text{set } (\text{snd } (\text{sym-preproc-aux } gs \text{ ks } (vs, fs)))$

⟨*proof*⟩

definition *sym-preproc* :: $('t \Rightarrow_0 'b::\text{semiring-1}) \text{ list} \Rightarrow ('t \Rightarrow_0 'b) \text{ list} \Rightarrow ('t \text{ list} \times ('t \Rightarrow_0 'b) \text{ list})$

where $\text{sym-preproc } gs \ fs = \text{sym-preproc-aux } gs \ [] (\text{Keys-to-list } fs, fs)$

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

⟨*proof*⟩

lemma *fst-sym-preproc*:

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

⟨*proof*⟩

lemma *snd-sym-preproc-superset*: $\text{set } fs \subseteq \text{set } (\text{snd } (\text{sym-preproc } gs \ fs))$

⟨*proof*⟩

lemma *in-snd-sym-preprocE*:

assumes $p \in \text{set } (\text{snd } (\text{sym-preproc } gs \ fs))$
assumes 1: $p \in \text{set } fs \implies \text{thesis}$
assumes 2: $\bigwedge g \ t. \ g \in \text{set } gs \implies p = \text{monom-mult } 1 \ t \ g \implies \text{thesis}$
shows *thesis*
 <proof>

lemma *snd-sym-preproc-pmdl*: $\text{pmdl } (\text{set } gs \cup \text{set } (\text{snd } (\text{sym-preproc } gs \ fs))) =$
 $\text{pmdl } (\text{set } gs \cup \text{set } fs)$
 <proof>

lemma *snd-sym-preproc-dgrad-set-le*:
assumes *dickson-grading* d
shows $\text{dgrad-set-le } d \ (\text{pp-of-term } ' \ \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc } gs \ fs))))$
 $(\text{pp-of-term } ' \ \text{Keys } (\text{set } gs \cup \text{set } (fs::('t \Rightarrow_0 \ 'b::\text{semiring-1-no-zero-divisors})$
 $\text{list})))$
 <proof>

corollary *snd-sym-preproc-dgrad-p-set-le*:
assumes *dickson-grading* d
shows $\text{dgrad-p-set-le } d \ (\text{set } (\text{snd } (\text{sym-preproc } gs \ fs))) \ (\text{set } gs \cup \text{set } (fs::('t \Rightarrow_0$
 $'b::\text{semiring-1-no-zero-divisors}) \ \text{list}))$
 <proof>

lemma *components-snd-sym-preproc-subset*:
 $\text{component-of-term } ' \ \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc } gs \ fs))) \subseteq$
 $\text{component-of-term } ' \ \text{Keys } (\text{set } gs \cup \text{set } (fs::('t \Rightarrow_0 \ 'b::\text{semiring-1-no-zero-divisors})$
 $\text{list}))$
 <proof>

lemma *snd-sym-preproc-complete*:
assumes $v \in \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc } gs \ fs)))$ **and** $g \in \text{set } gs$ **and** $lt \ g \ \text{adds}_t$
 v
shows $\text{monom-mult } (1::'b::\text{semiring-1-no-zero-divisors}) \ (\text{pp-of-term } v - lp \ g) \ g$
 $\in \text{set } (\text{snd } (\text{sym-preproc } gs \ fs))$
 <proof>

end

16.2 *lin-red*

context *ordered-term*
begin

definition *lin-red* $:: ('t \Rightarrow_0 \ 'b::\text{field}) \ \text{set} \Rightarrow ('t \Rightarrow_0 \ 'b) \Rightarrow ('t \Rightarrow_0 \ 'b) \Rightarrow \text{bool}$
where $\text{lin-red } F \ p \ q \equiv (\exists f \in F. \ \text{red-single } p \ q \ f \ 0)$

lin-red is a restriction of *red*, where the reductor (f) may only be multiplied by a constant factor, i. e. where the power-product is 0 .

lemma *lin-redI*:

assumes $f \in F$ **and** $\text{red-single } p \ q \ f \ 0$
shows $\text{lin-red } F \ p \ q$
 $\langle \text{proof} \rangle$

lemma lin-redE :
assumes $\text{lin-red } F \ p \ q$
obtains $f :: 't \Rightarrow_0 'b :: \text{field}$ **where** $f \in F$ **and** $\text{red-single } p \ q \ f \ 0$
 $\langle \text{proof} \rangle$

lemma lin-red-imp-red :
assumes $\text{lin-red } F \ p \ q$
shows $\text{red } F \ p \ q$
 $\langle \text{proof} \rangle$

lemma lin-red-Un : $\text{lin-red } (F \cup G) \ p \ q = (\text{lin-red } F \ p \ q \vee \text{lin-red } G \ p \ q)$
 $\langle \text{proof} \rangle$

lemma $\text{lin-red-imp-red-rtrancl}$:
assumes $(\text{lin-red } F)^{**} \ p \ q$
shows $(\text{red } F)^{**} \ p \ q$
 $\langle \text{proof} \rangle$

lemma $\text{phull-closed-lin-red}$:
assumes $\text{phull } B \subseteq \text{phull } A$ **and** $p \in \text{phull } A$ **and** $\text{lin-red } B \ p \ q$
shows $q \in \text{phull } A$
 $\langle \text{proof} \rangle$

16.3 Reduction

definition $\text{Macaulay-red} :: 't \ \text{list} \Rightarrow ('t \Rightarrow_0 'b) \ \text{list} \Rightarrow ('t \Rightarrow_0 'b :: \text{field}) \ \text{list}$
where $\text{Macaulay-red } vs \ fs =$
 $(\text{let } lts = \text{map } lt \ (\text{filter } (\lambda p. p \neq 0) \ fs) \ \text{in}$
 $\text{filter } (\lambda p. p \neq 0 \wedge lt \ p \notin \text{set } lts) \ (\text{mat-to-polys } vs \ (\text{row-echelon } (\text{polys-to-mat}$
 $\text{vs } fs)))$
 $)$

$\text{Macaulay-red } vs \ fs$ auto-reduces (w. r. t. lin-red) the given list fs and returns those non-zero polynomials whose leading terms are not in $lt\text{-set } (set \ fs)$. Argument vs is expected to be $\text{Keys-to-list } fs$; this list is passed as an argument to Macaulay-red , because it can be efficiently computed by symbolic preprocessing.

lemma Macaulay-red-alt :
 $\text{Macaulay-red } (\text{Keys-to-list } fs) \ fs = \text{filter } (\lambda p. lt \ p \notin lt\text{-set } (set \ fs)) \ (\text{Macaulay-list } fs)$
 $\langle \text{proof} \rangle$

lemma set-Macaulay-red :
 $\text{set } (\text{Macaulay-red } (\text{Keys-to-list } fs) \ fs) = \text{set } (\text{Macaulay-list } fs) - \{p. lt \ p \in lt\text{-set } (set \ fs)\}$

<proof>

lemma *Keys-Macaulay-red*: $Keys (set (Macaulay-red (Keys-to-list fs) fs)) \subseteq Keys (set fs)$
<proof>

end

context *gd-term*
begin

lemma *Macaulay-red-reducible*:
 assumes $f \in phull (set fs)$ **and** $F \subseteq set fs$ **and** $lt-set F = lt-set (set fs)$
 shows $(lin-red (F \cup set (Macaulay-red (Keys-to-list fs) fs)))^{**} f 0$
<proof>

primrec *pdata-pairs-to-list* :: $(t, 'b::field, 'c)$ *pdata-pair list* $\Rightarrow (t \Rightarrow_0 'b)$ *list*
where
 $pdata-pairs-to-list [] = []$
 $pdata-pairs-to-list (p \# ps) =$
 $(let f = fst (fst p); g = fst (snd p); lf = lp f; lg = lp g; l = lcs lf lg in$
 $(monom-mult (1 / lc f) (l - lf) f) \# (monom-mult (1 / lc g) (l - lg) g) \#$
 $(pdata-pairs-to-list ps)$
)

lemma *in-pdata-pairs-to-listI1*:
 assumes $(f, g) \in set ps$
 shows $monom-mult (1 / lc (fst f)) ((lcs (lp (fst f)) (lp (fst g))) - (lp (fst f)))$
 $(fst f) \in set (pdata-pairs-to-list ps)$ **(is ?m \in -)**
<proof>

lemma *in-pdata-pairs-to-listI2*:
 assumes $(f, g) \in set ps$
 shows $monom-mult (1 / lc (fst g)) ((lcs (lp (fst f)) (lp (fst g))) - (lp (fst g)))$
 $(fst g) \in set (pdata-pairs-to-list ps)$ **(is ?m \in -)**
<proof>

lemma *in-pdata-pairs-to-listE*:
 assumes $h \in set (pdata-pairs-to-list ps)$
 obtains $f g$ **where** $(f, g) \in set ps \vee (g, f) \in set ps$
 and $h = monom-mult (1 / lc (fst f)) ((lcs (lp (fst f)) (lp (fst g))) - (lp (fst f)))$
 $(fst f)$
<proof>

definition *f4-red-aux* :: $(t, 'b::field, 'c)$ *pdata list* $\Rightarrow (t, 'b, 'c)$ *pdata-pair list* $\Rightarrow (t \Rightarrow_0 'b)$ *list*
 where $f4-red-aux bs ps =$
 $(let aux = sym-preproc (map fst bs) (pdata-pairs-to-list ps) in Macaulay-red (fst aux) (snd aux))$

f_4 -red-aux only takes two arguments, since it does not distinguish between those elements of the current basis that are known to be a Gröbner basis (called gs in *Groebner-Bases.Algorithm-Schema*) and the remaining ones.

lemma f_4 -red-aux-not-zero: $0 \notin \text{set } (f_4\text{-red-aux } bs \ ps)$
 ⟨proof⟩

lemma f_4 -red-aux-irreducible:
assumes $h \in \text{set } (f_4\text{-red-aux } bs \ ps)$ **and** $b \in \text{set } bs$ **and** $\text{fst } b \neq 0$
shows $\neg \text{lt } (\text{fst } b) \ \text{adds}_t \ \text{lt } h$
 ⟨proof⟩

lemma f_4 -red-aux-dgrad-p-set-le:
assumes *dickson-grading* d
shows $d\text{grad-p-set-le } d \ (\text{set } (f_4\text{-red-aux } bs \ ps)) \ (\text{args-to-set } ([], \ bs, \ ps))$
 ⟨proof⟩

lemma *components- f_4 -red-aux-subset*:
component-of-term ‘ *Keys* ($\text{set } (f_4\text{-red-aux } bs \ ps)$) \subseteq *component-of-term* ‘ *Keys*
 ($\text{args-to-set } ([], \ bs, \ ps)$)
 ⟨proof⟩

lemma $pmdl$ - f_4 -red-aux: $\text{set } (f_4\text{-red-aux } bs \ ps) \subseteq pmdl \ (\text{args-to-set } ([], \ bs, \ ps))$
 ⟨proof⟩

lemma f_4 -red-aux-phull-reducible:
assumes $\text{set } ps \subseteq \text{set } bs \times \text{set } bs$
and $f \in \text{phull } (\text{set } (pdata\text{-pairs-to-list } ps))$
shows $(\text{red } (\text{fst } ' \ \text{set } bs \cup \text{set } (f_4\text{-red-aux } bs \ ps)))^{**} \ f \ 0$
 ⟨proof⟩

corollary f_4 -red-aux-spoly-reducible:
assumes $\text{set } ps \subseteq \text{set } bs \times \text{set } bs$ **and** $(p, \ q) \in \text{set } ps$
shows $(\text{red } (\text{fst } ' \ \text{set } bs \cup \text{set } (f_4\text{-red-aux } bs \ ps)))^{**} \ (\text{spoly } (\text{fst } p) \ (\text{fst } q)) \ 0$
 ⟨proof⟩

definition f_4 -red :: ($'t, 'b::\text{field}, 'c::\text{default}, 'd$) *complT*
where $f_4\text{-red } gs \ bs \ ps \ sps \ data = (\text{map } (\lambda h. \ (h, \ \text{default})) \ (f_4\text{-red-aux } (gs \ @ \ bs) \ sps), \ \text{snd } data)$

lemma $\text{fst-set-fst- f_4 -red}$: $\text{fst } ' \ \text{set } (\text{fst } (f_4\text{-red } gs \ bs \ ps \ sps \ data)) = \text{set } (f_4\text{-red-aux } (gs \ @ \ bs) \ sps)$
 ⟨proof⟩

lemma $\text{rcp-spec- f_4 -red}$: $\text{rcp-spec } f_4\text{-red}$
 ⟨proof⟩

lemmas $\text{compl-struct- f_4 -red} = \text{compl-struct-rcp}[OF \ \text{rcp-spec- f_4 -red}]$

lemmas $\text{compl-pmdl- f_4 -red} = \text{compl-pmdl-rcp}[OF \ \text{rcp-spec- f_4 -red}]$

lemmas $\text{compl-conn- f_4 -red} = \text{compl-conn-rcp}[OF \ \text{rcp-spec- f_4 -red}]$

16.4 Pair Selection

primrec $f_4\text{-sel-aux} :: 'a \Rightarrow ('t, 'b::\text{zero}, 'c) \text{pdata-pair list} \Rightarrow ('t, 'b, 'c) \text{pdata-pair list}$ **where**

```

 $f_4\text{-sel-aux} \text{ - } [] = []$ 
 $f_4\text{-sel-aux} \text{ t (p \# ps) =$ 
  (if (lcs (lp (fst (fst p))) (lp (fst (snd p)))) = t then
    p \# (f_4-sel-aux t ps)
  else
    []
  )

```

lemma $f_4\text{-sel-aux-subset: set (f_4-sel-aux t ps) \subseteq set ps$
<proof>

primrec $f_4\text{-sel} :: ('t, 'b::\text{zero}, 'c, 'd) \text{selT}$ **where**

```

 $f_4\text{-sel} \text{ gs bs [] data = []}$ 
 $f_4\text{-sel} \text{ gs bs (p \# ps) data = p \# (f_4-sel-aux (lcs (lp (fst (fst p))) (lp (fst (snd p)))) ps)$ 

```

lemma $\text{sel-spec-f}_4\text{-sel: sel-spec } f_4\text{-sel}$
<proof>

16.5 The F4 Algorithm

The F4 algorithm is just *gb-schema-direct* with parameters instantiated by suitable functions.

lemma $\text{struct-spec-f}_4: \text{struct-spec } f_4\text{-sel add-pairs-canon add-basis-canon } f_4\text{-red}$
<proof>

definition $f_4\text{-aux} :: ('t, 'b, 'c) \text{pdata list} \Rightarrow \text{nat} \times \text{nat} \times 'd \Rightarrow ('t, 'b, 'c) \text{pdata list}$
 \Rightarrow

```

  ( $'t, 'b, 'c$ )  $\text{pdata-pair list} \Rightarrow ('t, 'b::\text{field}, 'c::\text{default}) \text{pdata list}$ 
  where  $f_4\text{-aux} = \text{gb-schema-aux } f_4\text{-sel add-pairs-canon add-basis-canon } f_4\text{-red}$ 

```

lemmas $f_4\text{-aux-simps [code] = gb-schema-aux-simps[OF struct-spec-f}_4, \text{folded } f_4\text{-aux-def}]$

definition $f_4 :: ('t, 'b, 'c) \text{pdata}' \text{list} \Rightarrow 'd \Rightarrow ('t, 'b::\text{field}, 'c::\text{default}) \text{pdata}' \text{list}$
where $f_4 = \text{gb-schema-direct } f_4\text{-sel add-pairs-canon add-basis-canon } f_4\text{-red}$

lemmas $f_4\text{-simps [code] = gb-schema-direct-def[of } f_4\text{-sel add-pairs-canon add-basis-canon } f_4\text{-red, folded } f_4\text{-def } f_4\text{-aux-def}]$

lemmas $f_4\text{-isGB} = \text{gb-schema-direct-isGB[OF struct-spec-f}_4 \text{ compl-conn-f}_4\text{-red, folded } f_4\text{-def}]$

lemmas $f_4\text{-pmdl} = \text{gb-schema-direct-pmdl[OF struct-spec-f}_4 \text{ compl-pmdl-f}_4\text{-red, folded } f_4\text{-def}]$

16.5.1 Special Case: *punit*

lemma (in *gd-term*) *struct-spec-f4-punit*: *punit.struct-spec punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*
⟨*proof*⟩

definition *f4-aux-punit* :: ('a, 'b, 'c) *pdata list* ⇒ *nat × nat × 'd* ⇒ ('a, 'b, 'c) *pdata list* ⇒

('a, 'b, 'c) *pdata-pair list* ⇒ ('a, 'b::field, 'c::default) *pdata list*
where *f4-aux-punit* = *punit.gb-schema-aux punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*

lemmas *f4-aux-punit-simps* [code] = *punit.gb-schema-aux-simps*[OF *struct-spec-f4-punit*,
folded f4-aux-punit-def]

definition *f4-punit* :: ('a, 'b, 'c) *pdata' list* ⇒ 'd ⇒ ('a, 'b::field, 'c::default) *pdata'*
list

where *f4-punit* = *punit.gb-schema-direct punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*

lemmas *f4-punit-simps* [code] = *punit.gb-schema-direct-def*[of *punit.f4-sel add-pairs-punit-canon punit.add-basis-canon punit.f4-red*,
folded f4-punit-def f4-aux-punit-def]

lemmas *f4-punit-isGB* = *punit.gb-schema-direct-isGB*[OF *struct-spec-f4-punit punit.compl-conn-f4-red*,
folded f4-punit-def]

lemmas *f4-punit-pmdl* = *punit.gb-schema-direct-pmdl*[OF *struct-spec-f4-punit punit.compl-pmdl-f4-red*,
folded f4-punit-def]

end

end

17 Sample Computations with the F4 Algorithm

theory *F4-Examples*

imports

F4

Algorithm-Schema-Impl

Jordan-Normal-Form.Gauss-Jordan-IArray-Impl

Code-Target-Rat

begin

We only consider scalar polynomials here, but vector-polynomials could be handled, too.

17.1 Preparations

primrec *remdups-wrt-rev* :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a list **where**
remdups-wrt-rev f [] vs = [] |
remdups-wrt-rev f (x # xs) vs =
 (let fx = f x in if List.member vs fx then *remdups-wrt-rev* f xs vs else x #
 (*remdups-wrt-rev* f xs (fx # vs)))

lemma *remdups-wrt-rev-notin*: $v \in \text{set } vs \implies v \notin f' \text{ set } (\text{remdups-wrt-rev } f \text{ xs } vs)$
 <proof>

lemma *distinct-remdups-wrt-rev*: *distinct* (map f (*remdups-wrt-rev* f xs vs))
 <proof>

lemma *map-of-remdups-wrt-rev'*:
map-of (*remdups-wrt-rev* fst xs vs) k = *map-of* (filter (λx. fst x ∉ set vs) xs) k
 <proof>

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

global-interpretation *punit'*: *gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit*
cmp-term

rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = (λx. x)
and *punit.component-of-term* = (λ-. ())
and *punit.monom-mult* = *monom-mult-punit*
and *punit.mult-scalar* = *mult-scalar-punit*
and *punit'.punit.min-term* = *min-term-punit*
and *punit'.punit.lt* = *lt-punit cmp-term*
and *punit'.punit.lc* = *lc-punit cmp-term*
and *punit'.punit.tail* = *tail-punit cmp-term*
and *punit'.punit.ord-p* = *ord-p-punit cmp-term*
and *punit'.punit.ord-strict-p* = *ord-strict-p-punit cmp-term*
and *punit'.punit.keys-to-list* = *keys-to-list-punit cmp-term*
for *cmp-term* :: ('a::nat, 'b::{nat,add-wellorder}) pp nat-term-order

defines *max-punit* = *punit'.ordered-powerprod-lin.max*
and *max-list-punit* = *punit'.ordered-powerprod-lin.max-list*
and *find-adds-punit* = *punit'.punit.find-adds*
and *trd-aux-punit* = *punit'.punit.trd-aux*
and *trd-punit* = *punit'.punit.trd*
and *spoly-punit* = *punit'.punit.spoly*
and *count-const-lt-components-punit* = *punit'.punit.count-const-lt-components*
and *count-rem-components-punit* = *punit'.punit.count-rem-components*
and *const-lt-component-punit* = *punit'.punit.const-lt-component*
and *full-gb-punit* = *punit'.punit.full-gb*
and *add-pairs-single-sorted-punit* = *punit'.punit.add-pairs-single-sorted*
and *add-pairs-punit* = *punit'.punit.add-pairs*

```

and canon-pair-order-aux-punit = punit'.punit.canon-pair-order-aux
and canon-basis-order-punit = punit'.punit.canon-basis-order
and new-pairs-sorted-punit = punit'.punit.new-pairs-sorted
and product-crit-punit = punit'.punit.product-crit
and chain-ncrit-punit = punit'.punit.chain-ncrit
and chain-ocrit-punit = punit'.punit.chain-ocrit
and apply-icrit-punit = punit'.punit.apply-icrit
and apply-ncrit-punit = punit'.punit.apply-ncrit
and apply-ocrit-punit = punit'.punit.apply-ocrit
and Keys-to-list-punit = punit'.punit.Keys-to-list
and sym-preproc-addnew-punit = punit'.punit.sym-preproc-addnew
and sym-preproc-aux-punit = punit'.punit.sym-preproc-aux
and sym-preproc-punit = punit'.punit.sym-preproc
and Macaulay-mat-punit = punit'.punit.Macaulay-mat
and Macaulay-list-punit = punit'.punit.Macaulay-list
and pdata-pairs-to-list-punit = punit'.punit.pdata-pairs-to-list
and Macaulay-red-punit = punit'.punit.Macaulay-red
and f4-sel-aux-punit = punit'.punit.f4-sel-aux
and f4-sel-punit = punit'.punit.f4-sel
and f4-red-aux-punit = punit'.punit.f4-red-aux
and f4-red-punit = punit'.punit.f4-red
and f4-aux-punit = punit'.punit.f4-aux-punit
and f4-punit = punit'.punit.f4-punit
⟨proof⟩

```

```

lemma (in term-powerprod) compute-list-to-poly:
  list-to-poly ts cs = distr0 DRLEX (remdups-wrt-rev fst (zip ts cs)) []
⟨proof⟩

```

```

declare punit.compute-list-to-poly [code]

```

```

lemma (in ordered-term) compute-Macaulay-list:
  Macaulay-list ps =
    (let ts = Keys-to-list ps in
     filter (λp. p ≠ 0) (mat-to-polys ts (row-echelon (polys-to-mat ts ps)))
    )
⟨proof⟩

```

```

declare punit'.punit.compute-Macaulay-list [code]

```

```

declare conversesep-iff [code]

```

17.2 Computations

```

experiment begin interpretation trivariate0-rat ⟨proof⟩

```

```

lemma
  lt-punit DRLEX (X2 * Z ^ 3 + 3 * X2 * Y) = sparse0 [(0, 2), (2, 3)]
⟨proof⟩

```

lemma

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

lemma

tail-punit DRLEX $(X^2 * Z^3 + 3 * X^2 * Y) = 3 * X^2 * Y$
<proof>

lemma

ord-strict-p-punit DRLEX $(X^2 * Z^4 - 2 * Y^3 * Z^2) (X^2 * Z^7 + 2 * Y^3 * Z^2)$
<proof>

lemma

f4-punit DRLEX
[
 $(X^2 * Z^4 - 2 * Y^3 * Z^2, ()),$
 $(Y^2 * Z + 2 * Z^3, ()),$
]
() =
[
 $(X^2 * Y^2 * Z^2 + 4 * Y^3 * Z^2, ()),$
 $(X^2 * Z^4 - 2 * Y^3 * Z^2, ()),$
 $(Y^2 * Z + 2 * Z^3, ()),$
 $(X^2 * Y^4 * Z + 4 * Y^5 * Z, ()),$
]
<proof>

lemma

f4-punit DRLEX
[
 $(X^2 + Y^2 + Z^2 - 1, ()),$
 $(X * Y - Z - 1, ()),$
 $(Y^2 + X, ()),$
 $(Z^2 + X, ()),$
]
() =
[
 $(1, ()),$
]
<proof>

end

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

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

end

18 Syzygies of Multivariate Polynomials

theory *Syzygy*
 imports *Groebner-Bases More-MPoly-Type-Class*
begin

In this theory we first introduce the general concept of *syzygies* in modules, and then provide a method for computing Gröbner bases of syzygy modules of lists of multivariate vector-polynomials. Since syzygies in this context are themselves represented by vector-polynomials, this method can be applied repeatedly to compute bases of syzygy modules of syzygies, and so on.

instance *nat* :: *comm-powerprod* *<proof>*

18.1 Syzygy Modules Generated by Sets

context *module*
begin

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

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

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

end

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

context *module*
begin

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

lemma *rep-zero* [*simp*]: $rep\ 0 = 0$
<proof>

lemma *rep-uminus* [*simp*]: $rep\ (-\ r) = -\ rep\ r$
<proof>

lemma *rep-plus*: $rep\ (r + s) = rep\ r + rep\ s$
<proof>

lemma *rep-minus*: $\text{rep } (r - s) = \text{rep } r - \text{rep } s$
(proof)

lemma *rep-smult*: $\text{rep } (\text{monomial } c \ 0 * r) = c * s \ \text{rep } r$
(proof)

lemma *rep-in-span*: $\text{rep } r \in \text{span } (\text{keys } r)$
(proof)

lemma *spanE-rep*:
assumes $x \in \text{span } B$
obtains r where $\text{keys } r \subseteq B$ and $x = \text{rep } r$
(proof)

lemma *representsI*:
assumes $\text{keys } r \subseteq B$ and $\text{rep } r = x$
shows $\text{represents } B \ r \ x$
(proof)

lemma *representsD1*:
assumes $\text{represents } B \ r \ x$
shows $\text{keys } r \subseteq B$
(proof)

lemma *representsD2*:
assumes $\text{represents } B \ r \ x$
shows $x = \text{rep } r$
(proof)

lemma *represents-mono*:
assumes $\text{represents } B \ r \ x$ and $B \subseteq A$
shows $\text{represents } A \ r \ x$
(proof)

lemma *represents-self*: $\text{represents } \{x\} \ (\text{monomial } 1 \ x) \ x$
(proof)

lemma *represents-zero*: $\text{represents } B \ 0 \ 0$
(proof)

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

lemma *represents-uminus*:
assumes $\text{represents } B \ r \ x$
shows $\text{represents } B \ (- r) \ (- x)$
(proof)

lemma *represents-minus*:

assumes *represents* A r x **and** *represents* B s y

shows *represents* $(A \cup B)$ $(r - s)$ $(x - y)$

<proof>

lemma *represents-scale*:

assumes *represents* B r x

shows *represents* B $(\text{monomial } c \ 0 \ * \ r)$ $(c \ * \ s \ x)$

<proof>

lemma *represents-in-span*:

assumes *represents* B r x

shows $x \in \text{span } B$

<proof>

lemma *syzygy-module-iff*: $s \in \text{syzygy-module } B \iff \text{represents } B \ s \ 0$

<proof>

lemma *syzygy-moduleI*:

assumes *represents* B s 0

shows $s \in \text{syzygy-module } B$

<proof>

lemma *syzygy-moduleD*:

assumes $s \in \text{syzygy-module } B$

shows *represents* B s 0

<proof>

lemma *zero-in-syzygy-module*: $0 \in \text{syzygy-module } B$

<proof>

lemma *syzygy-module-closed-plus*:

assumes $s1 \in \text{syzygy-module } B$ **and** $s2 \in \text{syzygy-module } B$

shows $s1 + s2 \in \text{syzygy-module } B$

<proof>

lemma *syzygy-module-closed-minus*:

assumes $s1 \in \text{syzygy-module } B$ **and** $s2 \in \text{syzygy-module } B$

shows $s1 - s2 \in \text{syzygy-module } B$

<proof>

lemma *syzygy-module-closed-times-monomial*:

assumes $s \in \text{syzygy-module } B$

shows $\text{monomial } c \ 0 \ * \ s \in \text{syzygy-module } B$

<proof>

end

context *term-powerprod*
begin

lemma *keys-rep-subset*:

assumes $u \in \text{keys } (\text{pmdl.rep } r)$
obtains $t \ v$ **where** $t \in \text{Keys } (\text{Poly-Mapping.range } r)$ **and** $v \in \text{Keys } (\text{keys } r)$ **and**
 $u = t \oplus v$
 $\langle \text{proof} \rangle$

lemma *rep-mult-scalar*: $\text{pmdl.rep } (\text{punit.monom-mult } c \ 0 \ r) = c \odot \text{pmdl.rep } r$
 $\langle \text{proof} \rangle$

lemma *represents-mult-scalar*:

assumes $\text{pmdl.represents } B \ r \ x$
shows $\text{pmdl.represents } B \ (\text{punit.monom-mult } c \ 0 \ r) \ (c \odot x)$
 $\langle \text{proof} \rangle$

lemma *syzygy-module-closed-map-scale*: $s \in \text{pmdl.syzygy-module } B \implies c \cdot s \in \text{pmdl.syzygy-module } B$
 $\langle \text{proof} \rangle$

lemma *phull-syzygy-module*: $\text{phull } (\text{pmdl.syzygy-module } B) = \text{pmdl.syzygy-module } B$
 $\langle \text{proof} \rangle$

end

18.2 Polynomial Mappings on List-Indices

definition *pm-of-idx-pm* :: $('a \ \text{list}) \Rightarrow (\text{nat} \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b::\text{zero}$

where $\text{pm-of-idx-pm } xs \ f = \text{Abs-poly-mapping } (\lambda x. \text{lookup } f \ (\text{Min } \{i. i < \text{length } xs \wedge xs ! i = x\}))$ **when** $x \in \text{set } xs$

definition *idx-pm-of-pm* :: $('a \ \text{list}) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow \text{nat} \Rightarrow_0 'b::\text{zero}$

where $\text{idx-pm-of-pm } xs \ f = \text{Abs-poly-mapping } (\lambda i. \text{lookup } f \ (xs ! i))$ **when** $i < \text{length } xs$

lemma *lookup-pm-of-idx-pm*:

$\text{lookup } (\text{pm-of-idx-pm } xs \ f) = (\lambda x. \text{lookup } f \ (\text{Min } \{i. i < \text{length } xs \wedge xs ! i = x\}))$
when $x \in \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *lookup-pm-of-idx-pm-distinct*:

assumes $\text{distinct } xs$ **and** $i < \text{length } xs$
shows $\text{lookup } (\text{pm-of-idx-pm } xs \ f) \ (xs ! i) = \text{lookup } f \ i$
 $\langle \text{proof} \rangle$

lemma *keys-pm-of-idx-pm-subset*: $\text{keys } (\text{pm-of-idx-pm } xs \ f) \subseteq \text{set } xs$

$\langle \text{proof} \rangle$

lemma *range-pm-of-idx-pm-subset*: $Poly\text{-Mapping.range } (pm\text{-of-idx-pm } xs\ f) \subseteq \text{lookup } f\ \{0..<length\ xs\} - \{0\}$
 ⟨proof⟩

corollary *range-pm-of-idx-pm-subset'*: $Poly\text{-Mapping.range } (pm\text{-of-idx-pm } xs\ f) \subseteq Poly\text{-Mapping.range } f$
 ⟨proof⟩

lemma *pm-of-idx-pm-zero [simp]*: $pm\text{-of-idx-pm } xs\ 0 = 0$
 ⟨proof⟩

lemma *pm-of-idx-pm-plus*: $pm\text{-of-idx-pm } xs\ (f + g) = pm\text{-of-idx-pm } xs\ f + pm\text{-of-idx-pm } xs\ g$
 ⟨proof⟩

lemma *pm-of-idx-pm-uminus*: $pm\text{-of-idx-pm } xs\ (-f) = - pm\text{-of-idx-pm } xs\ f$
 ⟨proof⟩

lemma *pm-of-idx-pm-minus*: $pm\text{-of-idx-pm } xs\ (f - g) = pm\text{-of-idx-pm } xs\ f - pm\text{-of-idx-pm } xs\ g$
 ⟨proof⟩

lemma *pm-of-idx-pm-monom-mult*: $pm\text{-of-idx-pm } xs\ (punit.monom-mult\ c\ 0\ f) = punit.monom-mult\ c\ 0\ (pm\text{-of-idx-pm } xs\ f)$
 ⟨proof⟩

lemma *pm-of-idx-pm-monomial*:
 assumes *distinct xs*
 shows $pm\text{-of-idx-pm } xs\ (monomial\ c\ i) = (monomial\ c\ (xs\ !\ i))$ when $i < length\ xs$
 ⟨proof⟩

lemma *pm-of-idx-pm-take*:
 assumes $keys\ f \subseteq \{0..<j\}$
 shows $pm\text{-of-idx-pm } (take\ j\ xs)\ f = pm\text{-of-idx-pm } xs\ f$
 ⟨proof⟩

lemma *lookup-idx-pm-of-pm*: $lookup\ (idx\text{-pm-of-pm } xs\ f) = (\lambda i. lookup\ f\ (xs\ !\ i))$ when $i < length\ xs$
 ⟨proof⟩

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

lemma *idx-pm-of-pm-zero [simp]*: $idx\text{-pm-of-pm } xs\ 0 = 0$
 ⟨proof⟩

lemma *idx-pm-of-pm-plus*: $idx\text{-pm-of-pm } xs\ (f + g) = idx\text{-pm-of-pm } xs\ f + idx\text{-pm-of-pm } xs\ g$

xs g
⟨*proof*⟩

lemma *idx-pm-of-pm-minus*: *idx-pm-of-pm xs (f - g) = idx-pm-of-pm xs f - idx-pm-of-pm xs g*
⟨*proof*⟩

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

lemma *idx-pm-of-pm-of-idx-pm*:
assumes *distinct xs and keys f ⊆ {0..*length xs*}*
shows *idx-pm-of-pm xs (pm-of-idx-pm xs f) = f*
⟨*proof*⟩

18.3 POT Orders

context *ordered-term*
begin

definition *is-pot-ord* :: *bool*
where *is-pot-ord* $\longleftrightarrow (\forall u v. \text{component-of-term } u < \text{component-of-term } v \longrightarrow u \prec_t v)$

lemma *is-pot-ordI*:
assumes $\bigwedge u v. \text{component-of-term } u < \text{component-of-term } v \implies u \prec_t v$
shows *is-pot-ord*
⟨*proof*⟩

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

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

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

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

where $\text{map-component } f \ v = \text{term-of-pair } (\text{pp-of-term } v, f \ (\text{component-of-term } v))$

lemma $\text{pair-of-map-component}$ [*term-simps*]:
 $\text{pair-of-term } (\text{map-component } f \ v) = (\text{pp-of-term } v, f \ (\text{component-of-term } v))$
 ⟨*proof*⟩

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

lemma $\text{component-of-map-component}$ [*term-simps*]:
 $\text{component-of-term } (\text{map-component } f \ v) = f \ (\text{component-of-term } v)$
 ⟨*proof*⟩

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

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

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

lemma map-component-inj :
assumes $\text{inj } f$ **and** $\text{map-component } f \ u = \text{map-component } f \ v$
shows $u = v$
 ⟨*proof*⟩

end

18.4 Gröbner Bases of Syzygy Modules

locale $\text{gd-inf-term} =$
 $\text{gd-term } \text{pair-of-term } \text{term-of-pair } \text{ord } \text{ord-strict } \text{ord-term } \text{ord-term-strict}$
for $\text{pair-of-term}::'t \Rightarrow ('a::\text{graded-dickson-powerprod} \times \text{nat})$
and $\text{term-of-pair}::('a \times \text{nat}) \Rightarrow 't$
and $\text{ord}::'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infixl** $\langle \preceq \rangle$ 50)
and ord-strict (**infixl** $\langle \prec \rangle$ 50)
and $\text{ord-term}::'t \Rightarrow 't \Rightarrow \text{bool}$ (**infixl** $\langle \preceq_t \rangle$ 50)
and $\text{ord-term-strict}::'t \Rightarrow 't \Rightarrow \text{bool}$ (**infixl** $\langle \prec_t \rangle$ 50)
begin

In order to compute a Gröbner basis of the syzygy module of a list bs of polynomials, one first needs to “lift” bs to a new list bs' by adding further components, compute a Gröbner basis gs of bs' , and then filter out those elements of gs whose only non-zero components are those that were newly

added to bs . Function *init-syzygy-list* takes care of constructing bs' , and function *filter-syzygy-basis* does the filtering. Function *proj-orig-basis*, finally, projects the Gröbner basis gs of bs' to a Gröbner basis of the original list bs .

definition *lift-poly-syz* :: $nat \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow nat \Rightarrow ('t \Rightarrow_0 'b::semiring-1)$
where *lift-poly-syz* $n\ b\ i = Abs\text{-}poly\text{-}mapping$
 $(\lambda x. \text{if pair-of-term } x = (0, i) \text{ then } 1$
 $\text{else if } n \leq \text{component-of-term } x \text{ then lookup } b \text{ (map-component } (\lambda k. k - n) x)$
 $\text{else } 0)$

definition *proj-poly-syz* :: $nat \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::semiring-1)$
where *proj-poly-syz* $n\ b = Poly\text{-}Mapping.map\text{-}key (\lambda x. \text{map-component } (\lambda k. k + n) x) b$

definition *cofactor-list-syz* :: $nat \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b::semiring-1) list$
where *cofactor-list-syz* $n\ b = map (\lambda i. \text{proj-poly } i\ b) [0..<n]$

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

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

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

definition *syzygy-module-list* :: $('t \Rightarrow_0 'b) list \Rightarrow ('t \Rightarrow_0 'b::comm-ring-1) set$
where *syzygy-module-list* $bs = \text{atomize-poly } ' \text{ idx-pm-of-pm } bs \text{ ' pmdl.syzygy-module } (set\ bs)$

18.4.1 lift-poly-syz

lemma *keys-lift-poly-syz-aux*:

$\{x. (\text{if pair-of-term } x = (0, i) \text{ then } 1$
 $\text{else if } n \leq \text{component-of-term } x \text{ then lookup } b \text{ (map-component } (\lambda k. k - n)$
 $x)$
 $\text{else } 0\} \neq 0\} \subseteq \text{insert } (\text{term-of-pair } (0, i)) (\text{map-component } (\lambda k. k + n) ' \text{ keys } b)$
 $(\text{is } ?l \subseteq ?r) \text{ for } b::'t \Rightarrow_0 'b::semiring-1$
 $\langle \text{proof} \rangle$

lemma *lookup-lift-poly-syz*:

$\text{lookup } (\text{lift-poly-syz } n\ b\ i) =$
 $(\lambda x. \text{if pair-of-term } x = (0, i) \text{ then } 1 \text{ else if } n \leq \text{component-of-term } x \text{ then}$
 $\text{lookup } b \text{ (map-component } (\lambda k. k - n) x) \text{ else } 0)$
 $\langle \text{proof} \rangle$

corollary *lookup-lift-poly-syz-alt*:

$lookup (lift-poly-syz\ n\ b\ i) (term-of-pair\ (t, j)) =$
 $(if\ (t, j) = (0, i)\ then\ 1\ else\ if\ n \leq j\ then\ lookup\ b\ (term-of-pair\ (t, j -$
 $n))\ else\ 0)$
 $\langle proof \rangle$

lemma *keys-lift-poly-syz*:

$keys (lift-poly-syz\ n\ b\ i) = insert (term-of-pair\ (0, i)) (map-component\ (\lambda k. k +$
 $n)\ `keys\ b)$
 $\langle proof \rangle$

18.4.2 *proj-poly-syz*

lemma *inj-map-component-plus*: $inj (map-component\ (\lambda k. k + n))$

$\langle proof \rangle$

lemma *lookup-proj-poly-syz*: $lookup (proj-poly-syz\ n\ p)\ x = lookup\ p (map-component\ (\lambda k. k + n)\ x)$

$\langle proof \rangle$

lemma *lookup-proj-poly-syz-alt*:

$lookup (proj-poly-syz\ n\ p) (term-of-pair\ (t, i)) = lookup\ p (term-of-pair\ (t, i +$
 $n))$

$\langle proof \rangle$

lemma *keys-proj-poly-syz*: $keys (proj-poly-syz\ n\ p) = map-component\ (\lambda k. k + n)$
 $- `keys\ p$

$\langle proof \rangle$

lemma *proj-poly-syz-zero [simp]*: $proj-poly-syz\ n\ 0 = 0$

$\langle proof \rangle$

lemma *proj-poly-syz-plus*: $proj-poly-syz\ n\ (p + q) = proj-poly-syz\ n\ p + proj-poly-syz\ n\ q$

$\langle proof \rangle$

lemma *proj-poly-syz-sum*: $proj-poly-syz\ n\ (sum\ f\ A) = (\sum\ a \in A. proj-poly-syz\ n\ (f\ a))$

$\langle proof \rangle$

lemma *proj-poly-syz-sum-list*: $proj-poly-syz\ n\ (sum-list\ xs) = sum-list (map (proj-poly-syz\ n)\ xs)$

$\langle proof \rangle$

lemma *proj-poly-syz-monom-mult*:

$proj-poly-syz\ n (monom-mult\ c\ t\ p) = monom-mult\ c\ t (proj-poly-syz\ n\ p)$

$\langle proof \rangle$

lemma *proj-poly-syz-mult-scalar*:

$proj-poly-syz\ n (mult-scalar\ q\ p) = mult-scalar\ q (proj-poly-syz\ n\ p)$

<proof>

lemma *proj-poly-syz-lift-poly-syz*:

assumes $i < n$

shows $\text{proj-poly-syz } n (\text{lift-poly-syz } n p i) = p$

<proof>

lemma *proj-poly-syz-eq-zero-iff*: $\text{proj-poly-syz } n p = 0 \longleftrightarrow (\text{component-of-term } \text{keys } p \subseteq \{0..<n\})$

<proof>

lemma *component-of-lt-ge*:

assumes *is-pot-ord* **and** $\text{proj-poly-syz } n p \neq 0$

shows $n \leq \text{component-of-term } (\text{lt } p)$

<proof>

lemma *lt-proj-poly-syz*:

assumes *is-pot-ord* **and** $\text{proj-poly-syz } n p \neq 0$

shows $\text{lt } (\text{proj-poly-syz } n p) = \text{map-component } (\lambda k. k - n) (\text{lt } p)$ (**is - = ?l**)

<proof>

lemma *proj-proj-poly-syz*: $\text{proj-poly } k (\text{proj-poly-syz } n p) = \text{proj-poly } (k + n) p$

<proof>

lemma *poly-mapping-eqI-proj-syz*:

assumes $\text{proj-poly-syz } n p = \text{proj-poly-syz } n q$

and $\bigwedge k. k < n \implies \text{proj-poly } k p = \text{proj-poly } k q$

shows $p = q$

<proof>

18.4.3 cofactor-list-syz

lemma *length-cofactor-list-syz [simp]*: $\text{length } (\text{cofactor-list-syz } n p) = n$

<proof>

lemma *cofactor-list-syz-nth*:

assumes $i < n$

shows $(\text{cofactor-list-syz } n p) ! i = \text{proj-poly } i p$

<proof>

lemma *cofactor-list-syz-zero [simp]*: $\text{cofactor-list-syz } n 0 = \text{replicate } n 0$

<proof>

lemma *cofactor-list-syz-plus*:

$\text{cofactor-list-syz } n (p + q) = \text{map2 } (+) (\text{cofactor-list-syz } n p) (\text{cofactor-list-syz } n q)$

<proof>

18.4.4 *init-szygy-list*

lemma *length-init-szygy-list* [simp]: $\text{length } (\text{init-szygy-list } bs) = \text{length } bs$
<proof>

lemma *init-szygy-list-nth*:
assumes $i < \text{length } bs$
shows $(\text{init-szygy-list } bs) ! i = \text{lift-poly-syz } (\text{length } bs) (bs ! i)$
<proof>

lemma *Keys-init-szygy-list*:
 $\text{Keys } (\text{set } (\text{init-szygy-list } bs)) =$
 $\text{map-component } (\lambda k. k + \text{length } bs) \text{ ` } \text{Keys } (\text{set } bs) \cup (\lambda i. \text{term-of-pair } (0, i))$
 $\text{ ` } \{0..<\text{length } bs\}$
<proof>

lemma *pp-of-Keys-init-szygy-list-subset*:
 $\text{pp-of-term } \text{ ` } \text{Keys } (\text{set } (\text{init-szygy-list } bs)) \subseteq \text{insert } 0 (\text{pp-of-term } \text{ ` } \text{Keys } (\text{set } bs))$
<proof>

lemma *pp-of-Keys-init-szygy-list-superset*:
 $\text{pp-of-term } \text{ ` } \text{Keys } (\text{set } bs) \subseteq \text{pp-of-term } \text{ ` } \text{Keys } (\text{set } (\text{init-szygy-list } bs))$
<proof>

lemma *pp-of-Keys-init-szygy-list*:
assumes $bs \neq []$
shows $\text{pp-of-term } \text{ ` } \text{Keys } (\text{set } (\text{init-szygy-list } bs)) = \text{insert } 0 (\text{pp-of-term } \text{ ` } \text{Keys } (\text{set } bs))$
<proof>

lemma *component-of-Keys-init-szygy-list*:
 $\text{component-of-term } \text{ ` } \text{Keys } (\text{set } (\text{init-szygy-list } bs)) =$
 $(+) (\text{length } bs) \text{ ` } \text{component-of-term } \text{ ` } \text{Keys } (\text{set } bs) \cup \{0..<\text{length } bs\}$
<proof>

lemma *proj-lift-poly-syz*:
assumes $j < n$
shows $\text{proj-poly } j (\text{lift-poly-syz } n p i) = (1 \text{ when } j = i)$
<proof>

18.4.5 *proj-orig-basis*

lemma *length-proj-orig-basis* [simp]: $\text{length } (\text{proj-orig-basis } n bs) = \text{length } bs$
<proof>

lemma *proj-orig-basis-nth*:
assumes $i < \text{length } bs$
shows $(\text{proj-orig-basis } n bs) ! i = \text{proj-poly-syz } n (bs ! i)$
<proof>

lemma *proj-orig-basis-init-syzygy-list* [*simp*]:
 $\text{proj-orig-basis } (\text{length } bs) (\text{init-syzygy-list } bs) = bs$
 ⟨*proof*⟩

lemma *set-proj-orig-basis*: $\text{set } (\text{proj-orig-basis } n \ bs) = \text{proj-poly-syz } n \ \text{' } \text{set } bs$
 ⟨*proof*⟩

The following lemma could be generalized from *proj-poly-syz* to arbitrary module homomorphisms, i. e. functions respecting 0, addition and scalar multiplication.

lemma *pmdl-proj-orig-basis'*:
 $\text{pmdl } (\text{set } (\text{proj-orig-basis } n \ bs)) = \text{proj-poly-syz } n \ \text{' } \text{pmdl } (\text{set } bs) \ (\text{is } ?A = ?B)$
 ⟨*proof*⟩

18.4.6 *filter-syzygy-basis*

lemma *filter-syzygy-basis-alt*: $\text{filter-syzygy-basis } n \ bs = [b \leftarrow bs. \text{proj-poly-syz } n \ b = 0]$
 ⟨*proof*⟩

lemma *set-filter-syzygy-basis*:
 $\text{set } (\text{filter-syzygy-basis } n \ bs) = \{b \in \text{set } bs. \text{proj-poly-syz } n \ b = 0\}$
 ⟨*proof*⟩

18.4.7 *syzygy-module-list*

lemma *syzygy-module-listI*:
assumes $s' \in \text{pmdl.syzygy-module } (\text{set } bs)$ **and** $s = \text{atomize-poly } (\text{idx-pm-of-pm } bs \ s')$
shows $s \in \text{syzygy-module-list } bs$
 ⟨*proof*⟩

lemma *syzygy-module-listE*:
assumes $s \in \text{syzygy-module-list } bs$
obtains s' **where** $s' \in \text{pmdl.syzygy-module } (\text{set } bs)$ **and** $s = \text{atomize-poly } (\text{idx-pm-of-pm } bs \ s')$
 ⟨*proof*⟩

lemma *monom-mult-atomize*:
 $\text{monom-mult } c \ t \ (\text{atomize-poly } p) = \text{atomize-poly } (\text{MPoly-Type-Class.punit.monom-mult } (\text{monomial } c \ t) \ 0 \ p)$
 ⟨*proof*⟩

lemma *punit-monom-mult-monomial-idx-pm-of-pm*:
 $\text{MPoly-Type-Class.punit.monom-mult } (\text{monomial } c \ t) \ (0 :: \text{nat}) \ (\text{idx-pm-of-pm } bs \ s) =$
 $\text{idx-pm-of-pm } bs \ (\text{MPoly-Type-Class.punit.monom-mult } (\text{monomial } c \ t) \ (0 :: 't) \ \Rightarrow_0 \ 'b :: \text{ring-1}) \ s)$

<proof>

lemma *syzygy-module-list-closed-monom-mult*:

assumes $s \in \text{syzygy-module-list } bs$

shows $\text{monom-mult } c \ t \ s \in \text{syzygy-module-list } bs$

<proof>

lemma *pmdl-syzygy-module-list [simp]*: $\text{pmdl } (\text{syzygy-module-list } bs) = \text{syzygy-module-list } bs$

<proof>

The following lemma also holds without the distinctness constraint on bs , but then the proof becomes more difficult.

lemma *syzygy-module-listI'*:

assumes *distinct* bs **and** *sum-list* $(\text{map2 } \text{mult-scalar } (\text{cofactor-list-syz } (\text{length } bs) \ s) \ bs) = 0$

and *component-of-term* 'keys $s \subseteq \{0..<\text{length } bs\}$

shows $s \in \text{syzygy-module-list } bs$

<proof>

lemma *component-of-syzygy-module-list*:

assumes $s \in \text{syzygy-module-list } bs$

shows *component-of-term* 'keys $s \subseteq \{0..<\text{length } bs\}$

<proof>

lemma *map2-mult-scalar-proj-poly-syz*:

$\text{map2 } \text{mult-scalar } xs \ (\text{map } (\text{proj-poly-syz } n) \ ys) =$

$\text{map } (\text{proj-poly-syz } n \circ (\lambda(x, y). \text{mult-scalar } x \ y)) \ (\text{zip } xs \ ys)$

<proof>

lemma *map2-times-proj*:

$\text{map2 } (*) \ xs \ (\text{map } (\text{proj-poly } k) \ ys) = \text{map } (\text{proj-poly } k \circ (\lambda(x, y). \ x \odot \ y)) \ (\text{zip } xs \ ys)$

<proof>

Probably the following lemma also holds without the distinctness constraint on bs .

lemma *syzygy-module-list-subset*:

assumes *distinct* bs

shows $\text{syzygy-module-list } bs \subseteq \text{pmdl } (\text{set } (\text{init-syzygy-list } bs))$

<proof>

18.4.8 Cofactors

lemma *map2-mult-scalar-plus*:

$\text{map2 } (\odot) \ (\text{map2 } (+) \ xs \ ys) \ zs = \text{map2 } (+) \ (\text{map2 } (\odot) \ xs \ zs) \ (\text{map2 } (\odot) \ ys \ zs)$

<proof>

lemma *syz-cofactors*:

assumes $p \in \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows $\text{proj-poly-syz} (\text{length } bs) p = \text{sum-list} (\text{map2 mult-scalar} (\text{cofactor-list-syz} (\text{length } bs) p) bs)$
 ⟨proof⟩

18.4.9 Modules

lemma *pmdl-proj-orig-basis*:
assumes $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows $\text{pmdl} (\text{set} (\text{proj-orig-basis} (\text{length } bs) gs)) = \text{pmdl} (\text{set } bs)$
 ⟨proof⟩

lemma *pmdl-filter-syzygy-basis-subset*:
assumes *distinct* bs **and** $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows $\text{pmdl} (\text{set} (\text{filter-syzygy-basis} (\text{length } bs) gs)) \subseteq \text{pmdl} (\text{syzygy-module-list } bs)$
 ⟨proof⟩

lemma *ex-filter-syzygy-basis-adds-lt*:
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* $(\text{set } gs)$
and $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
and $f \in \text{pmdl} (\text{syzygy-module-list } bs)$ **and** $f \neq 0$
shows $\exists g \in \text{set} (\text{filter-syzygy-basis} (\text{length } bs) gs). g \neq 0 \wedge \text{lt } g \text{ adds}_t \text{ lt } f$
 ⟨proof⟩

lemma *pmdl-filter-syzygy-basis*:
fixes $bs::('t \Rightarrow_0 'b::\text{field}) \text{ list}$
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* $(\text{set } gs)$ **and**
 $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows $\text{pmdl} (\text{set} (\text{filter-syzygy-basis} (\text{length } bs) gs)) = \text{syzygy-module-list } bs$
 ⟨proof⟩

18.4.10 Gröbner Bases

lemma *proj-orig-basis-isGB*:
assumes *is-pot-ord* **and** *is-Groebner-basis* $(\text{set } gs)$ **and** $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows *is-Groebner-basis* $(\text{set} (\text{proj-orig-basis} (\text{length } bs) gs))$
 ⟨proof⟩

lemma *filter-syzygy-basis-isGB*:
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* $(\text{set } gs)$
and $\text{pmdl} (\text{set } gs) = \text{pmdl} (\text{set} (\text{init-syzygy-list } bs))$
shows *is-Groebner-basis* $(\text{set} (\text{filter-syzygy-basis} (\text{length } bs) gs))$
 ⟨proof⟩

end

end

19 Sample Computations of Syzygies

```

theory Syzygy-Examples
  imports Buchberger Algorithm-Schema-Impl Syzygy Code-Target-Rat
begin

```

19.1 Preparations

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

```

definition splus-pprod :: ('a::nat, 'b::nat) pp  $\Rightarrow$  -
  where splus-pprod = pprod.splus

```

```

definition monom-mult-pprod :: 'c::semiring-0  $\Rightarrow$  ('a::nat, 'b::nat) pp  $\Rightarrow$  (((('a, 'b)
pp  $\times$  nat)  $\Rightarrow_0$  'c)  $\Rightarrow$  -
  where monom-mult-pprod = pprod.monom-mult

```

```

definition mult-scalar-pprod :: (('a::nat, 'b::nat) pp  $\Rightarrow_0$  'c::semiring-0)  $\Rightarrow$  (((('a,
'b) pp  $\times$  nat)  $\Rightarrow_0$  'c)  $\Rightarrow$  -
  where mult-scalar-pprod = pprod.mult-scalar

```

```

definition adds-term-pprod :: ('a::nat, 'b::nat) pp  $\times$  -  $\Rightarrow$  -
  where adds-term-pprod = pprod.adds-term

```

lemma (in *gd-term*) *compute-trd-aux* [code]:

```

\Rightarrow trd-aux fs (tail p) (plus-monomial-less r (lc p) (lt p))
    | Some f  $\Rightarrow$  trd-aux fs (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail
f)) r
  )
  <proof>

```

```

locale gd-nat-inf-term = gd-nat-term pair-of-term term-of-pair cmp-term
  for pair-of-term::'t::nat-term  $\Rightarrow$  ('a::{nat-term, graded-dickson-powerprod}  $\times$ 
nat)
  and term-of-pair::('a  $\times$  nat)  $\Rightarrow$  't
  and cmp-term
begin

```

```

sublocale aux: gd-inf-term pair-of-term term-of-pair
   $\lambda$ s t. le-of-nat-term-order cmp-term (term-of-pair (s, the-min)) (term-of-pair
(t, the-min))
   $\lambda$ s t. lt-of-nat-term-order cmp-term (term-of-pair (s, the-min)) (term-of-pair
(t, the-min))
  le-of-nat-term-order cmp-term

```

lt-of-nat-term-order cmp-term \langle proof \rangle

definition *lift-keys* :: $\text{nat} \Rightarrow ('t, 'b) \text{ oalist-ntm} \Rightarrow ('t, 'b::\text{semiring-0}) \text{ oalist-ntm}$
where *lift-keys* *i xs* = *oalist-of-list-ntm* (*map-raw* ($\lambda kv. (\text{map-component } ((+) i) (fst kv), snd kv)$) (*list-of-oalist-ntm* *xs*))

lemma *list-of-oalist-lift-keys*:

list-of-oalist-ntm (*lift-keys* *i xs*) = (*map-raw* ($\lambda kv. (\text{map-component } ((+) i) (fst kv), snd kv)$) (*list-of-oalist-ntm* *xs*))
 \langle proof \rangle

Regardless of whether the above lemma holds (which might be the case) or not, we can use *lift-keys* in computations. Now, however, it is implemented rather inefficiently, because the list resulting from the application of *map-raw* is sorted again. That should not be a big problem though, since *lift-keys* is applied only once to every input polynomial before computing syzygies.

lemma *lookup-lift-keys-plus*:

lookup (*MP-oalist* (*lift-keys* *i xs*)) (*term-of-pair* (*t*, *i + k*)) = *lookup* (*MP-oalist* *xs*) (*term-of-pair* (*t*, *k*))
 (is ?l = ?r)
 \langle proof \rangle

lemma *keys-lift-keys-subset*:

keys (*MP-oalist* (*lift-keys* *i xs*)) \subseteq (*map-component* ($(+) i$)) ‘ *keys* (*MP-oalist* *xs*)
 (is ?l \subseteq ?r)
 \langle proof \rangle

end

global-interpretation *pprod'*: *gd-nat-inf-term* $\lambda x::('a, 'b) pp \times nat. x \lambda x. x \text{ cmp-term}$
rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.monom-mult* = *monom-mult-pprod*
and *pprod.mult-scalar* = *mult-scalar-pprod*
and *pprod.adds-term* = *adds-term-pprod*
for *cmp-term* :: $(('a::\text{nat}, 'b::\text{nat}) pp \times nat) \text{ nat-term-order}$
defines *shift-map-keys-pprod* = *pprod'.shift-map-keys*
and *lift-keys-pprod* = *pprod'.lift-keys*
and *min-term-pprod* = *pprod'.min-term*
and *lt-pprod* = *pprod'.lt*
and *lc-pprod* = *pprod'.lc*
and *tail-pprod* = *pprod'.tail*
and *comp-opt-p-pprod* = *pprod'.comp-opt-p*
and *ord-p-pprod* = *pprod'.ord-p*
and *ord-strict-p-pprod* = *pprod'.ord-strict-p*
and *find-adds-pprod* = *pprod'.find-adds*
and *trd-aux-pprod* = *pprod'.trd-aux*
and *trd-pprod* = *pprod'.trd*

and *spoly-pprod* = *pprod'.spoly*
and *count-const-lt-components-pprod* = *pprod'.count-const-lt-components*
and *count-rem-components-pprod* = *pprod'.count-rem-components*
and *const-lt-component-pprod* = *pprod'.const-lt-component*
and *full-gb-pprod* = *pprod'.full-gb*
and *keys-to-list-pprod* = *pprod'.keys-to-list*
and *Keys-to-list-pprod* = *pprod'.Keys-to-list*
and *add-pairs-single-sorted-pprod* = *pprod'.add-pairs-single-sorted*
and *add-pairs-pprod* = *pprod'.add-pairs*
and *canon-pair-order-aux-pprod* = *pprod'.canon-pair-order-aux*
and *canon-basis-order-pprod* = *pprod'.canon-basis-order*
and *new-pairs-sorted-pprod* = *pprod'.new-pairs-sorted*
and *component-crit-pprod* = *pprod'.component-crit*
and *chain-ncrit-pprod* = *pprod'.chain-ncrit*
and *chain-ocrit-pprod* = *pprod'.chain-ocrit*
and *apply-icrit-pprod* = *pprod'.apply-icrit*
and *apply-ncrit-pprod* = *pprod'.apply-ncrit*
and *apply-ocrit-pprod* = *pprod'.apply-ocrit*
and *trdsp-pprod* = *pprod'.trdsp*
and *gb-sel-pprod* = *pprod'.gb-sel*
and *gb-red-aux-pprod* = *pprod'.gb-red-aux*
and *gb-red-pprod* = *pprod'.gb-red*
and *gb-aux-pprod* = *pprod'.gb-aux*
and *gb-pprod* = *pprod'.gb*
and *filter-syzygy-basis-pprod* = *pprod'.aux.filter-syzygy-basis*
and *init-syzygy-list-pprod* = *pprod'.aux.init-syzygy-list*
and *lift-poly-syz-pprod* = *pprod'.aux.lift-poly-syz*
and *map-component-pprod* = *pprod'.map-component*
<proof>

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

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

lemma *compute-shift-map-keys-pprod* [*code abstract*]:
list-of-oalist-ntm (*shift-map-keys-pprod* *t f xs*) = *map-raw* ($\lambda(k, v).$ (*splus-pprod* *t k, f v*)) (*list-of-oalist-ntm* *xs*)
<proof>

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

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

lemma *POT-is-pot-ord*: *pprod'.is-pot-ord* (*TYPE('a::nat)*) (*TYPE('b::nat)*) (*POT*

to)
 ⟨proof⟩

definition $Vec_0 :: nat \Rightarrow (('a, nat) pp \Rightarrow_0 'b) \Rightarrow (('a::nat, nat) pp \times nat) \Rightarrow_0 'b::semiring-1$ **where**

$Vec_0\ i\ p = mult\text{-}scalar\text{-}pprod\ p\ (Poly\text{-}Mapping.single\ (0, i)\ 1)$

definition $syzygy\text{-}basis\ to\ bs =$

$filter\text{-}syzygy\text{-}basis\text{-}pprod\ (length\ bs)\ (map\ fst\ (gb\text{-}pprod\ (POT\ to)\ (map\ (\lambda p. (p, ()))\ (init\text{-}syzygy\text{-}list\text{-}pprod\ bs))\ ()))$

thm $pprod'.aux.filter\text{-}syzygy\text{-}basis\text{-}isGB[OF\ POT\text{-}is\text{-}pot\text{-}ord]$

lemma $lift\text{-}poly\text{-}syz\text{-}MP\text{-}oalist$ [code]:

$lift\text{-}poly\text{-}syz\text{-}pprod\ n\ (MP\text{-}oalist\ xs)\ i = MP\text{-}oalist\ (OAlist\text{-}insert\text{-}ntm\ ((0, i), 1)\ (lift\text{-}keys\text{-}pprod\ n\ xs))$
 ⟨proof⟩

19.2 Computations

experiment begin interpretation $trivariate_0\text{-}rat$ ⟨proof⟩

lemma

$syzygy\text{-}basis\ DRLEX [Vec_0\ 0\ (X^2 * Z^3 + 3 * X^2 * Y), Vec_0\ 0\ (X * Y * Z + 2 * Y^2)] =$
 $[Vec_0\ 0\ (C_0\ (1 / 3) * X * Y * Z + C_0\ (2 / 3) * Y^2) + Vec_0\ 1\ (C_0\ (-1 / 3) * X^2 * Z^3 - X^2 * Y)]$
 ⟨proof⟩

value [code] $syzygy\text{-}basis\ DRLEX [Vec_0\ 0\ (X^2 * Z^3 + 3 * X^2 * Y), Vec_0\ 0\ (X * Y * Z + 2 * Y^2), Vec_0\ 0\ (X - Y + 3 * Z)]$

lemma

$map\ fst\ (gb\text{-}pprod\ (POT\ DRLEX)\ (map\ (\lambda p. (p, ()))\ (init\text{-}syzygy\text{-}list\text{-}pprod\ [Vec_0\ 0\ (X^4 + 3 * X^2 * Y), Vec_0\ 0\ (Y^3 + 2 * X * Z), Vec_0\ 0\ (Z^2 - X - Y)]))\ ())) =$
 [$Vec_0\ 0\ 1 + Vec_0\ 3\ (X^4 + 3 * X^2 * Y),$
 $Vec_0\ 1\ 1 + Vec_0\ 3\ (Y^3 + 2 * X * Z),$
 $Vec_0\ 0\ (Y^3 + 2 * X * Z) - Vec_0\ 1\ (X^4 + 3 * X^2 * Y),$
 $Vec_0\ 2\ 1 + Vec_0\ 3\ (Z^2 - X - Y),$
 $Vec_0\ 1\ (Z^2 - X - Y) - Vec_0\ 2\ (Y^3 + 2 * X * Z),$
 $Vec_0\ 0\ (Z^2 - X - Y) - Vec_0\ 2\ (X^4 + 3 * X^2 * Y),$
 $Vec_0\ 0\ (- (Y^3 * Z^2) + Y^4 + X * Y^3 + 2 * X^2 * Z + 2 * X * Y * Z - 2 * X * Z^3) +$
 $Vec_0\ 1\ (X^4 * Z^2 - X^5 - X^4 * Y - 3 * X^3 * Y - 3 * X^2 * Y^2 + 3 * X^2 * Y * Z^2)$
]
 ⟨proof⟩

lemma

syzygy-basis DRLEX [$Vec_0\ 0\ (X^4 + 3 * X^2 * Y)$, $Vec_0\ 0\ (Y^3 + 2 * X * Z)$, $Vec_0\ 0\ (Z^2 - X - Y)$] =
 [
 $Vec_0\ 0\ (Y^3 + 2 * X * Z) - Vec_0\ 1\ (X^4 + 3 * X^2 * Y)$,
 $Vec_0\ 1\ (Z^2 - X - Y) - Vec_0\ 2\ (Y^3 + 2 * X * Z)$,
 $Vec_0\ 0\ (Z^2 - X - Y) - Vec_0\ 2\ (X^4 + 3 * X^2 * Y)$,
 $Vec_0\ 0\ (- (Y^3 * Z^2) + Y^4 + X * Y^3 + 2 * X^2 * Z + 2 * X * Y * Z - 2 * X * Z^3) +$
 $Vec_0\ 1\ (X^4 * Z^2 - X^5 - X^4 * Y - 3 * X^3 * Y - 3 * X^2 * Y^2 + 3 * X^2 * Y * Z^2)$
]
 ⟨*proof*⟩

value [*code*] *syzygy-basis DRLEX* [$Vec_0\ 0\ (X * Y - Z)$, $Vec_0\ 0\ (X * Z - Y)$, $Vec_0\ 0\ (Y * Z - X)$]

lemma

*map fst (gb-pprod (POT DRLEX) (map (λp. (p, ())) (init-syzygy-list-pprod [Vec_0 0 (X * Y - Z), Vec_0 0 (X * Z - Y), Vec_0 0 (Y * Z - X)])) ()) =*
 [
 $Vec_0\ 0\ 1 + Vec_0\ 3\ (X * Y - Z)$,
 $Vec_0\ 1\ 1 + Vec_0\ 3\ (X * Z - Y)$,
 $Vec_0\ 2\ 1 + Vec_0\ 3\ (Y * Z - X)$,
 $Vec_0\ 0\ (- X * Z + Y) + Vec_0\ 1\ (X * Y - Z)$,
 $Vec_0\ 0\ (- Y * Z + X) + Vec_0\ 2\ (X * Y - Z)$,
 $Vec_0\ 1\ (- Y * Z + X) + Vec_0\ 2\ (X * Z - Y)$,
 $Vec_0\ 1\ (-Y) + Vec_0\ 2\ (X) + Vec_0\ 3\ (Y^2 - X^2)$,
 $Vec_0\ 0\ (Z) + Vec_0\ 2\ (-X) + Vec_0\ 3\ (X^2 - Z^2)$,
 $Vec_0\ 0\ (Y - Y * Z^2) + Vec_0\ 1\ (Y^2 * Z - Z) + Vec_0\ 2\ (Y^2 - Z^2)$,
 $Vec_0\ 0\ (- Y) + Vec_0\ 1\ (- (X * Y)) + Vec_0\ 2\ (X^2 - 1) + Vec_0\ 3\ (X - X^3)$
]
 ⟨*proof*⟩

lemma

syzygy-basis DRLEX [$Vec_0\ 0\ (X * Y - Z)$, $Vec_0\ 0\ (X * Z - Y)$, $Vec_0\ 0\ (Y * Z - X)$] =
 [
 $Vec_0\ 0\ (- X * Z + Y) + Vec_0\ 1\ (X * Y - Z)$,
 $Vec_0\ 0\ (- Y * Z + X) + Vec_0\ 2\ (X * Y - Z)$,
 $Vec_0\ 1\ (- Y * Z + X) + Vec_0\ 2\ (X * Z - Y)$,
 $Vec_0\ 0\ (Y - Y * Z^2) + Vec_0\ 1\ (Y^2 * Z - Z) + Vec_0\ 2\ (Y^2 - Z^2)$
]
 ⟨*proof*⟩

end

end

theory *Groebner-PM*

imports *Polynomials.MPoly-PM Reduced-GB*

begin

We prove results that hold specifically for Gröbner bases in polynomial rings, where the polynomials really have *indeterminates*.

context *pm-powerprod*

begin

lemmas *finite-reduced-GB-Polys =*

punit.finite-reduced-GB-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-is-reduced-GB-Polys =*

punit.reduced-GB-is-reduced-GB-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-is-GB-Polys =*

punit.reduced-GB-is-GB-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-is-auto-reduced-Polys =*

punit.reduced-GB-is-auto-reduced-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-is-monic-set-Polys =*

punit.reduced-GB-is-monic-set-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-nonzero-Polys =*

punit.reduced-GB-nonzero-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-ideal-Polys =*

punit.reduced-GB-pmdl-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-unique-Polys =*

punit.reduced-GB-unique-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *reduced-GB-Polys =*

punit.reduced-GB-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

lemmas *ideal-eq-UNIV-iff-reduced-GB-eq-one-Polys =*

ideal-eq-UNIV-iff-reduced-GB-eq-one-dgrad-p-set[simplified, OF dickson-grading-varnum, where m=0, simplified dgrad-p-set-varnum]

19.3 Univariate Polynomials

lemma (in *—*) *adds-univariate-linear:*

assumes finite X and card X ≤ 1 and s ∈ .[X] and t ∈ .[X]

obtains $s \text{ adds } t \mid t \text{ adds } s$
<proof>

context

fixes $X :: 'x \text{ set}$

assumes $\text{fin-}X$: *finite* X **and** $\text{card-}X$: $\text{card } X \leq 1$

begin

lemma *ord-iff-adds-univariate*:

assumes $s \in .[X]$ **and** $t \in .[X]$

shows $s \preceq t \longleftrightarrow s \text{ adds } t$

<proof>

lemma *adds-iff-deg-le-univariate*:

assumes $s \in .[X]$ **and** $t \in .[X]$

shows $s \text{ adds } t \longleftrightarrow \text{deg-pm } s \leq \text{deg-pm } t$

<proof>

corollary *ord-iff-deg-le-univariate*: $s \in .[X] \implies t \in .[X] \implies s \preceq t \longleftrightarrow \text{deg-pm } s \leq \text{deg-pm } t$

<proof>

lemma *poly-deg-univariate*:

assumes $p \in P[X]$

shows $\text{poly-deg } p = \text{deg-pm } (\text{lpp } p)$

<proof>

lemma *reduced-GB-univariate-cases*:

assumes $F \subseteq P[X]$

obtains g **where** $g \in P[X]$ **and** $g \neq 0$ **and** $\text{lcf } g = 1$ **and** $\text{punit.reduced-GB } F = \{g\}$

$\text{punit.reduced-GB } F = \{\}$

<proof>

corollary *deg-reduced-GB-univariate-le*:

assumes $F \subseteq P[X]$ **and** $f \in \text{ideal } F$ **and** $f \neq 0$ **and** $g \in \text{punit.reduced-GB } F$

shows $\text{poly-deg } g \leq \text{poly-deg } f$

<proof>

end

19.4 Homogeneity

lemma *is-reduced-GB-homogeneous*:

assumes $\bigwedge f. f \in F \implies \text{homogeneous } f$ **and** $\text{punit.is-reduced-GB } G$ **and** $\text{ideal } G = \text{ideal } F$

and $g \in G$

shows *homogeneous* g

<proof>

lemma *lp-dehomogenize*:
assumes *is-hom-ord x and homogeneous p*
shows $lpp\ (dehomogenize\ x\ p) = except\ (lpp\ p)\ \{x\}$
 $\langle proof \rangle$

lemma *isGB-dehomogenize*:
assumes *is-hom-ord x and finite X and $G \subseteq P[X]$ and punit.is-Groebner-basis G*
and $\bigwedge g. g \in G \implies homogeneous\ g$
shows *punit.is-Groebner-basis (dehomogenize x ‘ G)*
 $\langle proof \rangle$

end

context *extended-ord-pm-powerprod*
begin

lemma *extended-ord-lp*:
assumes *None \notin indets p*
shows $restrict-indets-pp\ (extended-ord.lpp\ p) = lpp\ (restrict-indets\ p)$
 $\langle proof \rangle$

lemma *restrict-indets-reduced-GB*:
assumes *finite X and $F \subseteq P[X]$*
shows *punit.is-Groebner-basis (restrict-indets ‘ extended-ord.punit.reduced-GB (homogenize None ‘ extend-indets ‘ F))*
(is ?thesis1)
and *ideal (restrict-indets ‘ extended-ord.punit.reduced-GB (homogenize None ‘ extend-indets ‘ F)) = ideal F*
(is ?thesis2)
and *restrict-indets ‘ extended-ord.punit.reduced-GB (homogenize None ‘ extend-indets ‘ F) $\subseteq P[X]$*
(is ?thesis3)
 $\langle proof \rangle$

end

end

References

- [1] W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Society, July 1994.
- [2] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Mod-*

- ulo a Zero Dimensional Polynomial Ideal*). PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. English translation in *Journal of Symbolic Computation* 41(3–4):475–511, Special Issue on Logic, Mathematics, and Computer Science: Interactions.
- [3] B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems (An Algorithmic Criterion for the Solvability of an Algebraic System of Equations). *Aequationes Mathematicae*, pages 374–383, 1970. (English translation in *Gröbner Bases and Applications (Proceedings of the International Conference “33 Years of Gröbner Bases”, 1998)*, London Mathematical Society Lecture Note Series 251, Cambridge University Press, 1998, pages 535–545).
- [4] B. Buchberger. A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases. In E. W. Ng, editor, *Symbolic and Algebraic Computation (Proceedings of EUROSAM’79, Marseille, June 26–28)*, volume 72 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 1979.
- [5] B. Buchberger. Introduction to Gröbner Bases. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications*, number 251 in London Mathematical Society Lectures Notes Series, pages 3 – 31. Cambridge University Press, 1998.
- [6] B. Buchberger. Gröbner Rings in Theorema: A Case Study in Functors and Categories. Technical Report 2003-49, Johannes Kepler University Linz, Spezialforschungsbereich F013, November 2003.
- [7] A. Chaieb and M. Wenzel. Context aware Calculation and Deduction: Ring Equalities via Gröbner Bases in Isabelle. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (Proceedings of Calculemus’2007, Hagenberg, Austria, June 27–30)*, volume 4573 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2007.
- [8] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F_4). *Journal of Pure and Applied Algebra*, 139(1):61–88, 1999.
- [9] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases without Reduction to Zero (F_5). In T. Mora, editor, *Proceedings of ISSAC’02*, pages 61–88. ACM Press, 2002.
- [10] J. S. Jorge, V. M. Guilas, and J. L. Freire. Certifying properties of an efficient functional program for computing Gröbner bases. *Journal of Symbolic Computation*, 44(5):571–582, 2009.

- [11] M. Kreuzer and L. Robbiano. *Computational Commutative Algebra 1*. Springer-Verlag, 2000.
- [12] A. Maletzky. *Computer-Assisted Exploration of Gröbner Bases Theory in Theorema*. PhD thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, May 2016. To appear.
- [13] I. Medina-Bulo, F. Palomo-Lozano, and J.-L. Ruiz-Reina. A verified COMMON LISP implementation of Buchberger’s algorithm in ACL2. *Journal of Symbolic Computation*, 45(1):96–123, 2010.
- [14] T. Mora. An Introduction to Commutative and Non-Commutative Gröbner Bases. *Theoretical Computer Science*, 134(1):131–173, 1994.
- [15] C. Schwarzweiler. Gröbner Bases – Theory Refinement in the Mizar System. In M. Kohlhase, editor, *Mathematical Knowledge Management (4th International Conference, Bremen, Germany, July 15–17)*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 299–314. Springer, 2006.
- [16] L. Théry. A Machine-Checked Implementation of Buchberger’s Algorithm. *Journal of Automated Reasoning*, 26(2):107–137, 2001.
- [17] F. Winkler and B. Buchberger. A Criterion for Eliminating Unnecessary Reductions in the Knuth-Bendix Algorithm. In J. Demetrovics, G. Katona, and A. Salomaa, editors, *Proceedings of Algebra and Logic in Computer Science, Győr, Hungary*, volume 42 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 849–869. North Holland, 1983.