

Gröbner Bases Theory

Fabian Immler and Alexander Maletzky*

February 23, 2021

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> | 10 |
| 2.1.5 | <i>map-idx</i> | 10 |
| 2.1.6 | <i>map-dup</i> | 11 |
| 2.1.7 | Filtering Minimal Elements | 12 |
| 3 | Properties of Binary Relations | 14 |
| 3.1 | <i>wfp-on</i> | 14 |
| 3.2 | Relations | 14 |
| 3.3 | Setup for Connection to Theory <i>Abstract–Rewriting</i> . <i>Abstract-Rewriting</i> | 16 |

*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 | 39 |
| 5.3 | Buchberger's Criteria for Avoiding Useless Pairs | 40 |
| 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 | 44 |
| 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 | 87 |
| 7 | Buchberger's Algorithm | 88 |
| 7.1 | Reduction | 88 |
| 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 | 101 |
| 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 | 104 |
| 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 | 138 |
| 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*: $\text{set} (\text{zip} (\text{map } f \text{ } xs) (\text{map } g \text{ } xs)) = (\lambda x. (f \ x, g \ x)) \text{ ` } (\text{set } xs)$
 <proof>

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

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

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

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

2.1.1 *max-list*

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

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*

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

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

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

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

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

where $\text{insert-list } x \text{ } xs = (\text{if } x \in \text{set } xs \text{ then } xs \text{ else } x \# xs)$

lemma $\text{set-insert-list: set } (\text{insert-list } x \text{ } xs) = \text{insert } x \text{ } (\text{set } xs)$
 $\langle \text{proof} \rangle$

2.1.4 remdups-wrt

primrec $\text{remdups-wrt} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{remdups-wrt-base: remdups-wrt } - [] = [] \mid$
 $\text{remdups-wrt-rec: remdups-wrt } f (x \# xs) = (\text{if } f x \in f \text{ ' set } xs \text{ then remdups-wrt } f \text{ } xs \text{ else } x \# \text{remdups-wrt } f \text{ } xs)$

lemma $\text{set-remdups-wrt: } f \text{ ' set } (\text{remdups-wrt } f \text{ } xs) = f \text{ ' set } xs$
 $\langle \text{proof} \rangle$

lemma $\text{subset-remdups-wrt: set } (\text{remdups-wrt } f \text{ } xs) \subseteq \text{set } xs$
 $\langle \text{proof} \rangle$

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

lemma $\text{distinct-remdups-wrt: distinct } (\text{remdups-wrt } f \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma $\text{map-remdups-wrt: map } f \text{ } (\text{remdups-wrt } f \text{ } xs) = \text{remdups } (\text{map } f \text{ } xs)$
 $\langle \text{proof} \rangle$

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

2.1.5 map-idx

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

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

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

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

lemma *map-idx-nth*:

assumes $i < \text{length } xs$

shows $(\text{map-idx } f \text{ } xs \ n) \ ! \ i = f \ (xs \ ! \ i) \ (n + i)$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<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* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list}$ **where**

map-dup - - $[] = []$

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 $rel :: 'a \Rightarrow 'a \Rightarrow bool$

begin

primrec $filter\text{-}min\text{-}aux :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$filter\text{-}min\text{-}aux []\ ys = ys$

$filter\text{-}min\text{-}aux (x \# xs)\ ys =$

(if $(\exists y \in (set\ xs \cup set\ ys). rel\ y\ x)$ then $(filter\text{-}min\text{-}aux\ xs\ ys)$

else $(filter\text{-}min\text{-}aux\ xs\ (x \# ys))$)

definition $filter\text{-}min :: 'a\ list \Rightarrow 'a\ list$

where $filter\text{-}min\ xs = filter\text{-}min\text{-}aux\ xs\ []$

definition $filter\text{-}min\text{-}append :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$

where $filter\text{-}min\text{-}append\ xs\ ys =$

(let $P = (\lambda zs. \lambda x. \neg (\exists z \in set\ zs. rel\ z\ x))$; $ys1 = filter\ (P\ xs)\ ys$ in

$(filter\ (P\ ys1)\ xs) @ ys1$)

lemma $filter\text{-}min\text{-}aux\text{-}supset$: $set\ ys \subseteq set\ (filter\text{-}min\text{-}aux\ xs\ ys)$

<proof>

lemma $filter\text{-}min\text{-}aux\text{-}subset$: $set\ (filter\text{-}min\text{-}aux\ xs\ ys) \subseteq set\ xs \cup set\ ys$

<proof>

lemma $filter\text{-}min\text{-}aux\text{-}relE$:

assumes $transp\ rel$ **and** $x \in set\ xs$ **and** $x \notin set\ (filter\text{-}min\text{-}aux\ xs\ ys)$

obtains y **where** $y \in set\ (filter\text{-}min\text{-}aux\ xs\ ys)$ **and** $rel\ y\ x$

<proof>

lemma $filter\text{-}min\text{-}aux\text{-}minimal$:

assumes $transp\ rel$ **and** $x \in set\ (filter\text{-}min\text{-}aux\ xs\ ys)$ **and** $y \in set\ (filter\text{-}min\text{-}aux\ xs\ ys)$

and $rel\ x\ y$

assumes $\bigwedge a\ b. a \in set\ xs \cup set\ ys \Longrightarrow b \in set\ ys \Longrightarrow rel\ a\ b \Longrightarrow a = b$

shows $x = y$

<proof>

lemma $filter\text{-}min\text{-}aux\text{-}distinct$:

assumes $reflp\ rel$ **and** $distinct\ ys$

shows $distinct\ (filter\text{-}min\text{-}aux\ xs\ ys)$

<proof>

lemma $filter\text{-}min\text{-}subset$: $set\ (filter\text{-}min\ xs) \subseteq set\ xs$

<proof>

lemma $filter\text{-}min\text{-}cases$:

assumes $transp\ rel$ **and** $x \in set\ xs$

assumes $x \in set\ (filter\text{-}min\ xs) \Longrightarrow thesis$

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

corollary *filter-min-relE*:

assumes *transp rel* **and** *reflp rel* **and** $x \in \text{set } xs$
obtains *y* **where** $y \in \text{set } (\text{filter-min } xs)$ **and** $\text{rel } y \ x$
 ⟨*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 $x \in \text{set } (\text{filter-min-append } xs \ ys) \implies$ *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 rel y x \implies 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** $\text{rel } y \ x$
 ⟨*proof*⟩

lemma *filter-min-append-minimal*:

assumes $\bigwedge x' \ y'. x' \in \text{set } xs \implies y' \in \text{set } xs \implies \text{rel } x' \ y' \implies x' = y'$
and $\bigwedge x' \ y'. x' \in \text{set } ys \implies y' \in \text{set } ys \implies \text{rel } x' \ y' \implies x' = y'$
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 *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** $\rightarrow 50$)

begin

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

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

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** \leftrightarrow^* 50) **where**
 $srtc\ a\ b \equiv sc^{**}\ a\ b$

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

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 $dw-closed\ A$
<proof>

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

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

lemma $dw-closed-empty$: $dw-closed\ \{\}$
<proof>

lemma $dw-closed-UNIV$: $dw-closed\ UNIV$

$\langle proof \rangle$

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

abbreviation (*input*) $relset::('a * 'a)$ set where

$relset \equiv \{(x, y). x \rightarrow y\}$

lemma *rtc-rtranclI*:

assumes $a \rightarrow^* b$

shows $(a, b) \in relset^*$

$\langle proof \rangle$

lemma *final-NF*: $(is-final\ a) = (a \in NF\ relset)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *confluent-CR*: $is-confluent = CR\ relset$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *loc-confluent-WCR*:

shows $is-loc-confluent = WCR\ relset$

$\langle proof \rangle$

lemma *wf-converse*:

shows $(wfP\ r^{\hat{-}-1}) = (wf\ (relset^{-1}))$

$\langle proof \rangle$

lemma *wf-SN*:

shows $(wfP\ r^{\hat{-}-1}) = (SN\ relset)$

$\langle proof \rangle$

3.4 Simple Lemmas

lemma *rtrancl-is-final*:

assumes $a \rightarrow^* b$ and *is-final* a

shows $a = b$

$\langle proof \rangle$

lemma *cs-refl*:

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

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

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

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

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

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

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

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

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

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

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

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

lemma *unique-nf-imp-confluence-on*:

assumes *major*: $\bigwedge a\ b1\ b2. a \in A \implies (a \rightarrow^* b1) \implies (a \rightarrow^* b2) \implies \textit{is-final}\ b1$
 $\implies \textit{is-final}\ b2 \implies b1 = b2$
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
 ⟨*proof*⟩

corollary *unique-nf-imp-confluence*:

assumes $\bigwedge a\ b1\ b2. (a \rightarrow^* b1) \implies (a \rightarrow^* b2) \implies \textit{is-final}\ b1 \implies \textit{is-final}\ b2 \implies b1 = b2$
and *wfP* $((\rightarrow)^{-1-1})$
shows *is-confluent*
 ⟨*proof*⟩

end

3.5 Advanced Results and the Generalized Newman Lemma

definition *relbelow-on*: $'a\ \textit{set} \implies ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool})$

where *relbelow-on* $A\ \textit{ord}\ z\ \textit{rel}\ a\ b \equiv (a \in A \wedge b \in A \wedge \textit{rel}\ a\ b \wedge \textit{ord}\ a\ z \wedge \textit{ord}\ b\ z)$

definition *cbelow-on-1*: $'a\ \textit{set} \implies ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool})$

where *cbelow-on-1* $A\ \textit{ord}\ z\ \textit{rel} \equiv (\textit{relbelow-on}\ A\ \textit{ord}\ z\ \textit{rel})^{++}$

definition *cbelow-on*: $'a\ \textit{set} \implies ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool})$

where *cbelow-on* $A\ \textit{ord}\ z\ \textit{rel}\ a\ b \equiv (a = b \wedge b \in A \wedge \textit{ord}\ b\ z) \vee \textit{cbelow-on-1}\ A\ \textit{ord}\ z\ \textit{rel}\ a\ b$

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*: $'a\ \textit{set} \implies ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$

where *is-loc-connective-on* $A\ \textit{ord}\ r \iff (\forall a \in A. \forall b1\ b2. r\ a\ b1 \wedge r\ a\ b2 \longrightarrow \textit{cbelow-on}\ A\ \textit{ord}\ a\ (\textit{relation.sc}\ r)\ b1\ b2)$

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

lemma *cbelow-on-first-below*:
assumes *cbelow-on A ord z rel a b*
shows *ord a z*
<proof>

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

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

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

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

lemma *cbelow-on-induct [consumes 1, case-names base step]*:
assumes *a: cbelow-on A ord z rel a b*
and *base: $a \in A \implies ord a z \implies P a$*
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 *$P b$*
<proof>

lemma *cbelow-on-symmetric*:
assumes *main: cbelow-on A ord z rel a b and symp rel*
shows *cbelow-on A ord z rel b a*
<proof>

lemma *cbelow-on-transitive*:
assumes *cbelow-on A ord z rel a b and cbelow-on A ord z rel b c*
shows *cbelow-on A ord z rel a c*
<proof>

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

locale *relation-order = relation +*

```

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

lemma relation-refines:
  assumes a → b
  shows ord b a
  ⟨proof⟩

lemma relation-wf: wfp-on (→)-1-1 A
  ⟨proof⟩

lemma rtc-implies-cbelow-on:
  assumes dw-closed A and main: a →* b and a ∈ A and ord a c
  shows cbelow-on A ord c (↔) a b
  ⟨proof⟩

lemma cs-implies-cbelow-on:
  assumes dw-closed A and a ↓* b and a ∈ A and b ∈ A and ord a c and ord b
  c
  shows cbelow-on A ord c (↔) a b
  ⟨proof⟩

The generalized Newman lemma, taken from [17]:

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

end

theorem loc-connectivity-equiv-ChurchRosser:
  assumes relation-order r ord UNIV
  shows relation.is-ChurchRosser r = is-loc-connective-on UNIV ord r
  ⟨proof⟩

end

```

4 Polynomial Reduction

```

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

```

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

context *ordered-term*
begin

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

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

definition *is-red* :: ($'t \Rightarrow_0 'b :: \text{field}$) *set* \Rightarrow ($'t \Rightarrow_0 'b$) \Rightarrow *bool*
where *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 $\text{red } F\ p\ q$
<proof>

lemma *red-setE*:
assumes $\text{red } F\ p\ q$
obtains f **and** t **where** $f \in F$ **and** $\text{red-single } p\ q\ f\ t$
<proof>

lemma *red-empty*: $\neg \text{red } \{\} p\ q$
<proof>

lemma *red-singleton-zero*: $\neg \text{red } \{0\} p\ q$
<proof>

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

lemma *red-unionI1*:
assumes $\text{red } F\ p\ q$
shows $\text{red } (F \cup G)\ p\ q$
<proof>

lemma *red-unionI2*:
assumes $\text{red } G\ p\ q$
shows $\text{red } (F \cup G)\ p\ q$
<proof>

lemma *red-subset*:
assumes $\text{red } G\ p\ q$ **and** $G \subseteq F$
shows $\text{red } F\ p\ q$
<proof>

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

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

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

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

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

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

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

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

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

lemma *red-single-self*:
assumes $p \neq 0$
shows *red-single* $p 0 p 0$
<proof>

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

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*
 \langle *proof* \rangle

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

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)*
 \langle *proof* \rangle

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

lemma *red-indE*:
assumes *red F p q*
shows $(\exists f \in F. f \neq 0 \wedge lt f adds_t lt p \wedge$
 $(q = p - monom-mult (lc p / lc f) (lp p - lp f) f)) \vee$
 $red F (tail p) (q - monomial (lc p) (lt p))$
 \langle *proof* \rangle

lemma *is-redI*:
assumes *red F a b*
shows *is-red F a*
 \langle *proof* \rangle

lemma *is-redE*:
assumes *is-red F a*
obtains b **where** *red F a b*
 \langle *proof* \rangle

lemma *is-red-alt*:
shows *is-red F a* \longleftrightarrow $(\exists b. red F a b)$
 \langle *proof* \rangle

lemma *is-red-singletonI*:
assumes *is-red F q*
obtains p **where** $p \in F$ **and** *is-red {p} q*

<proof>

lemma *is-red-singletonD*:

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

shows *is-red* *F* *q*

<proof>

lemma *is-red-singleton-trans*:

assumes *is-red* {*f*} *p* **and** *lt* *g* *adds_t* *lt* *f* **and** $g \neq 0$

shows *is-red* {*g*} *p*

<proof>

lemma *is-red-singleton-not-0*:

assumes *is-red* {*f*} *p*

shows $f \neq 0$

<proof>

lemma *irred-0*:

shows \neg *is-red* *F* 0

<proof>

lemma *is-red-indI1*:

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

shows *is-red* *F* *p*

<proof>

lemma *is-red-indI2*:

assumes $p \neq 0$ **and** *is-red* *F* (*tail* *p*)

shows *is-red* *F* *p*

<proof>

lemma *is-red-indE*:

assumes *is-red* *F* *p*

shows $(\exists f \in F. f \neq 0 \wedge$ *lt* *f* *adds_t* *lt* *p*) \vee *is-red* *F* (*tail* *p*)

<proof>

lemma *rtrancl-0*:

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

shows $x = 0$

<proof>

lemma *red-rtrancl-ord*:

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

shows $q \preceq_p p$

<proof>

lemma *components-red-subset*:

assumes *red* *F* *p* *q*

shows *component-of-term* ‘ *keys* *q* \subseteq *component-of-term* ‘ *keys* *p* \cup *compo-*

nent-of-term ‘ *Keys F* ’
 ⟨*proof*⟩

corollary *components-red-rtrancl-subset*:

assumes $(\text{red } F)^{**} p q$
shows *component-of-term* ‘ *keys* $q \subseteq \text{component-of-term}$ ‘ *keys* $p \cup \text{component-of-term}$ ‘ *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* $(\text{monom-mult } c s p) (\text{monom-mult } c s q) f (s + t)$
 ⟨*proof*⟩

lemma *red-single-plus-1*:

assumes *red-single* $p q f t$ **and** $t \oplus lt f \notin \text{keys } (p + r)$
shows *red-single* $(q + r) (p + r) f t$
 ⟨*proof*⟩

lemma *red-single-plus-2*:

assumes *red-single* $p q f t$ **and** $t \oplus lt f \notin \text{keys } (q + r)$
shows *red-single* $(p + r) (q + r) f t$
 ⟨*proof*⟩

lemma *red-single-plus-3*:

assumes *red-single* $p q f t$ **and** $t \oplus lt f \in \text{keys } (p + r)$ **and** $t \oplus lt f \in \text{keys } (q + r)$
shows $\exists s. \text{red-single } (p + r) s f t \wedge \text{red-single } (q + r) s f t$
 ⟨*proof*⟩

lemma *red-single-plus*:

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

lemma *red-single-diff*:

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

lemma *red-monom-mult*:

assumes $a: \text{red } F p q$ **and** $c \neq 0$
shows *red F* $(\text{monom-mult } c s p) (\text{monom-mult } c s q)$

$\langle proof \rangle$

lemma *red-plus-keys-disjoint*:

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

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

$\langle proof \rangle$

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

<proof>

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 $p - q \in pmdl\ F$

<proof>

lemma *in-pmdl-srtc*:

assumes $p \in pmdl\ F$
shows *relation.srtc* $(red\ F)$ p 0

<proof>

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

assumes $(red\ F)^{**}$ p q
shows $p - q \in pmdl\ F$

<proof>

corollary *red-diff-in-pmdl*:

assumes $red\ F$ p q
shows $p - q \in pmdl\ F$

<proof>

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

assumes $(red\ F)^{**}$ p 0
shows $p \in pmdl\ F$

<proof>

lemma *pmdl-closed-red*:

assumes $pmdl\ B \subseteq pmdl\ A$ **and** $p \in pmdl\ A$ **and** $red\ B$ p q
shows $q \in pmdl\ A$

<proof>

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

lemma *red-rtrancl-mult*:

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

<proof>

corollary *red-rtrancl-uminus*:

assumes $(red\ F)^{**}\ p\ q$
shows $(red\ F)^{**}\ (-p)\ (-q)$
<proof>

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

assumes $a: (red\ F)^{**}\ (p - q)\ r$
and cases: $P\ p\ p\ !!y\ z. [\ (red\ F)^{**}\ (p - q)\ z; red\ F\ z\ y; P\ p\ (q + z)] ==> P$
 $p\ (q + y)$
shows $P\ p\ (q + r)$
<proof>

lemma *red-rtrancl-diff-0-induct* [*consumes 1, case-names base step*]:

assumes $a: (red\ F)^{**}\ (p - q)\ 0$
and base: $P\ p\ p$ **and** *ind:* $\bigwedge y\ z. [\ (red\ F)^{**}\ (p - q)\ y; red\ F\ y\ z; P\ p\ (y + q)]$
 $==> P\ p\ (z + q)$
shows $P\ p\ q$
<proof>

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

<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 $\exists f \in F. \exists v \in \text{keys } p. f \neq 0 \wedge \text{lt } f \text{ adds}_t v$
 $\langle \text{proof} \rangle$

lemma *is-red-addsE*:
assumes *is-red* F p
obtains $f v$ **where** $f \in F$ **and** $v \in \text{keys } p$ **and** $f \neq 0$ **and** $\text{lt } f \text{ adds}_t v$
 $\langle \text{proof} \rangle$

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

lemma *is-red-subset*:
assumes *red*: *is-red* A p **and** *sub*: $A \subseteq B$
shows *is-red* B p
 $\langle \text{proof} \rangle$

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

lemma *red-single-mult-const*:
assumes *red-single* p q f t **and** $c \neq 0$
shows *red-single* p q (*monom-mult* c 0 f) t
 $\langle \text{proof} \rangle$

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 $(\text{red } F)^{**} (p + r)$ $(q + r)$
 $\langle \text{proof} \rangle$

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

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

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

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 (*monomial* c v)
 $\langle \text{proof} \rangle$

lemma *is-red-monomialD*:

assumes *is-red* F (*monomial* c v)

shows $c \neq 0$

<proof>

lemma *is-red-monomialE*:

assumes *is-red* F (*monomial* c v)

obtains f **where** $f \in F$ **and** $f \neq 0$ **and** *lt* f *adds_t* v

<proof>

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

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

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

<proof>

lemma *conversion-property*:

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

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

<proof>

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

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

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

<proof>

lemma *is-red-map-scale*:

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

shows *is-red* F p

<proof>

corollary *is-irred-map-scale*: \neg *is-red* F $p \implies \neg$ *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 $is-red\ F\ p \vee is-red\ F\ q$
 $\langle proof \rangle$

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

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 $dgrad-set-le\ d\ \{t\}$ ($pp-of-term\ 'keys\ p$)
 $\langle proof \rangle$

lemma $dgrad-p-set-le-red-single$:
assumes $dickson-grading\ d$ **and** $red-single\ p\ q\ f\ t$
shows $dgrad-p-set-le\ d\ \{q\}\ \{f,\ p\}$
 $\langle proof \rangle$

lemma $dgrad-p-set-le-red$:
assumes $dickson-grading\ d$ **and** $red\ F\ p\ q$
shows $dgrad-p-set-le\ d\ \{q\}$ ($insert\ p\ F$)
 $\langle proof \rangle$

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

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 $d\ t \leq m$
 $\langle proof \rangle$

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 $q \in dgrad-p-set\ d\ m$
 $\langle proof \rangle$

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** $F \subseteq \text{dgrad-p-set } d \ m$ **and** $p \in \text{dgrad-p-set } d \ m$
and $(\text{red } F)^{**} \ p \ q$
shows $q \in \text{dgrad-p-set } d \ m$
<proof>

lemma *red-rtrancl-repE*:

assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$ **and** *finite* G **and** $p \in \text{dgrad-p-set } d \ m$
and $(\text{red } G)^{**} \ p \ r$
obtains q **where** $p = r + (\sum_{g \in G}. q \ g \odot g)$ **and** $\bigwedge g. q \ g \in \text{punit.dgrad-p-set } d \ m$
and $\bigwedge g. \text{lt } (q \ g \odot g) \preceq_t \text{lt } p$
<proof>

lemma *is-relation-order-red*:

assumes *dickson-grading* d
shows *Confluence.relation-order* $(\text{red } F) \ (\prec_p) \ (\text{dgrad-p-set } d \ m)$
<proof>

lemma *red-wf-dgrad-p-set-aux*:

assumes *dickson-grading* d **and** $F \subseteq \text{dgrad-p-set } d \ m$
shows *wfp-on* $(\text{red } F)^{-1-1} \ (\text{dgrad-p-set } d \ m)$
<proof>

lemma *red-wf-dgrad-p-set*:

assumes *dickson-grading* d **and** $F \subseteq \text{dgrad-p-set } d \ m$
shows *wfP* $(\text{red } F)^{-1-1}$
<proof>

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

lemma *cbelow-on-monom-mult*:

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

lemma *cbelow-on-monom-mult-monomial*:

assumes $c \neq 0$
and *cbelow-on* $(\text{dgrad-p-set } d \ m) \ (\prec_p) \ (\text{monomial } c' \ v) \ (\lambda a \ b. \text{red } F \ a \ b \vee \text{red } F \ b \ a) \ p \ q$
shows *cbelow-on* $(\text{dgrad-p-set } d \ m) \ (\prec_p) \ (\text{monomial } c \ (t \oplus v)) \ (\lambda a \ b. \text{red } F \ a \ b \vee \text{red } F \ b \ a) \ p \ q$
<proof>

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

lemma *is-full-pmdlI-lt-dgrad-p-set*:
assumes *dickson-grading* d **and** $B \subseteq \text{dgrad-p-set } d \ m$
assumes $\bigwedge k. k \in \text{component-of-term } \text{'Keys } (B::(\text{'t} \Rightarrow_0 \text{'b}::\text{field}) \ \text{set}) \implies$
 $(\exists b \in B. b \neq 0 \wedge \text{component-of-term } (lt \ b) = k \wedge lp \ b = 0)$
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* :: $(\text{'t} \Rightarrow_0 \text{'b}) \ \text{list} \Rightarrow \text{'t} \Rightarrow (\text{'t} \Rightarrow_0 \text{'b}::\text{zero}) \ \text{option}$ **where**
 $\text{find-adds } [] \ - = \text{None}$
 $\text{find-adds } (f \ # \ fs) \ u = (\text{if } f \neq 0 \wedge lt \ f \ \text{adds}_t \ u \ \text{then } \text{Some } f \ \text{else } \text{find-adds } fs \ u)$

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 $lt \ f \ \text{adds}_t \ u$
 $\langle \text{proof} \rangle$

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** $f \neq 0 \implies \neg lt \ f \ \text{adds}_t \ u \implies \text{thesis}$

shows *thesis*
 ⟨*proof*⟩

lemma *find-adds-SomeD-red-single*:

assumes $p \neq 0$ **and** *find-adds fs (lt p) = Some f*
shows *red-single p (tail p - monom-mult (lc p / lc f) (lp p - lp f) (tail f)) f (lp p - lp f)*
 ⟨*proof*⟩

lemma *find-adds-SomeD-red*:

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

end

4.7.2 Function *trd*

context *gd-term*

begin

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

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

lemma *trd-term-wf*:

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

function *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 =
 (if $p = 0$ then
 r
 else
 case find-adds fs (lt p) of
 None \Rightarrow *trd-aux fs (tail p) (r + monomial (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*⟩

termination ⟨*proof*⟩

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

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

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

lemma *trd-aux-irred*:
assumes $\neg \text{is-red } (\text{set } fs) r$
shows $\neg \text{is-red } (\text{set } fs) (\text{trd-aux } fs p r)$
 <proof>

corollary *trd-irred*: $\neg \text{is-red } (\text{set } fs) (\text{trd } fs p)$
 <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* $(lt p) = \text{component-of-term } (lt q)$ then
 (*monom-mult* $(1 / lc p) ((lcs (lp p) (lp q)) - (lp p)) (\text{tail } p)$,
monom-mult $(1 / lc q) ((lcs (lp p) (lp q)) - (lp q)) (\text{tail } q)$)
 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 $(dgrad-p\text{-set } d m) (\prec_p)$

(*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* :: ($'t \Rightarrow_0 'b::\text{field}$) *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*
{proof}

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

assumes *component-of-term (lt p) \neq component-of-term (lt q)*
shows *crit-pair-cbelow-on d m F p q*
{proof}

lemma *crit-pair-cbelow-sym*:

assumes *crit-pair-cbelow-on d m F p q*
shows *crit-pair-cbelow-on d m F q p*
{proof}

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

assumes *dickson-grading d and* $F \subseteq \text{dgrad-p-set } d \ m$ **and** $p \in \text{dgrad-p-set } d \ m$
and $q \in \text{dgrad-p-set } d \ m$
and *relation.cs (red F) (fst (crit-pair p q)) (snd (crit-pair p q))*
shows *crit-pair-cbelow-on d m F p q*
{proof}

lemma *crit-pair-cbelow-mono*:

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

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

assumes $p \neq 0$ **and** *component-of-term (lt p) = component-of-term (lt q)*
defines $t1 \equiv lp \ p$
defines $t2 \equiv lp \ q$
shows *red-single (monomial (- 1) (term-of-pair (lcs t1 t2), component-of-term (lt p)))*
 $(fst (crit-pair p q)) \ p \ (lcs \ t1 \ t2 - t1)$
{proof}

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

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

defines $t1 \equiv lp\ p$
defines $t2 \equiv lp\ q$
shows $red_single\ (monomial\ (-\ 1)\ (term_of_pair\ (lcs\ t1\ t2,\ component_of_term\ (lt\ p))))$
 $(snd\ (crit_pair\ p\ q))\ q\ (lcs\ t1\ t2\ -\ t2)$
 $\langle proof \rangle$

lemma $GB_imp_crit_pair_cbelow_dgrad_p_set$:
assumes $dickson_grading\ d$ **and** $F \subseteq dgrad_p_set\ d\ m$ **and** $is_Groebner_basis\ F$
assumes $p \in F$ **and** $q \in F$ **and** $p \neq 0$ **and** $q \neq 0$
shows $crit_pair_cbelow_on\ d\ m\ F\ p\ q$
 $\langle proof \rangle$

lemma $spoly_alt$:
assumes $p \neq 0$ **and** $q \neq 0$
shows $spoly\ p\ q = fst\ (crit_pair\ p\ q) - snd\ (crit_pair\ p\ q)$
 $\langle proof \rangle$

lemma $spoly_same$: $spoly\ p\ p = 0$
 $\langle proof \rangle$

lemma $spoly_swap$: $spoly\ p\ q = -\ spoly\ q\ p$
 $\langle proof \rangle$

lemma $spoly_red_zero_imp_crit_pair_cbelow_on$:
assumes $dickson_grading\ d$ **and** $F \subseteq dgrad_p_set\ d\ m$ **and** $p \in dgrad_p_set\ d\ m$
and $q \in dgrad_p_set\ d\ m$ **and** $p \neq 0$ **and** $q \neq 0$ **and** $(red\ F)^{**}\ (spoly\ p\ q)\ 0$
shows $crit_pair_cbelow_on\ d\ m\ F\ p\ q$
 $\langle proof \rangle$

lemma $dgrad_p_set_le_spoly_zero$: $dgrad_p_set_le\ d\ \{spoly\ p\ 0\}\ \{p\}$
 $\langle proof \rangle$

lemma $dgrad_p_set_le_spoly$:
assumes $dickson_grading\ d$
shows $dgrad_p_set_le\ d\ \{spoly\ p\ q\}\ \{p,\ q\}$
 $\langle proof \rangle$

lemma $dgrad_p_set_closed_spoly$:
assumes $dickson_grading\ d$ **and** $p \in dgrad_p_set\ d\ m$ **and** $q \in dgrad_p_set\ d\ m$
shows $spoly\ p\ q \in dgrad_p_set\ d\ m$
 $\langle proof \rangle$

lemma $components_spoly_subset$: $component_of_term\ 'keys\ (spoly\ p\ q) \subseteq component_of_term\ 'Keys\ \{p,\ q\}$
 $\langle proof \rangle$

lemma $pmdl_closed_spoly$:
assumes $p \in pmdl\ F$ **and** $q \in pmdl\ F$

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 $\text{dg: dickson-grading } d$ **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 $\text{relation.is-confluent-on } (\text{red } F) \ (\text{dgrad-p-set } d \ m)$

⟨proof⟩

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

assumes $\text{dickson-grading } d$ **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 $\text{is-Groebner-basis } F$

⟨proof⟩

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

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

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

shows $\text{is-Groebner-basis } F$

⟨proof⟩

lemmas $\text{Buchberger-criterion-finite} = \text{Buchberger-criterion-dgrad-p-set}[\text{OF } \text{dickson-grading-dgrad-dummy } \text{dgrad-p-set-exhaust-expl}]$

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

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

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

⟨proof⟩

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

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

shows $\text{is-red } G \ f$

⟨proof⟩

lemma $\text{is-Groebner-basis-empty: is-Groebner-basis } \{\}$

<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 $\text{gcs } (\text{punit.lt } p) (\text{punit.lt } q) = 0$*

shows *$\text{punit.crit-pair-cbelow-on } d \ m \ F \ p \ q$*

<proof>

lemma *chain-criterion:*

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

and *$p \neq 0$ and $q \neq 0$ and $\text{lp } r$ adds lcs $(\text{lp } p) (\text{lp } q)$*

and *component-of-term $(\text{lt } r) = \text{component-of-term } (\text{lt } p)$*

and *$\text{pr}: \text{crit-pair-cbelow-on } d \ m \ F \ p \ r$ and $\text{rq}: \text{crit-pair-cbelow-on } d \ m \ F \ r \ q$*

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

<proof>

5.4 Weak and Strong Gröbner Bases

lemma *ord-p-wf-on:*

assumes *dickson-grading d*

shows *$\text{wfp-on } (\prec_p) (\text{dgrad-p-set } d \ m)$*

<proof>

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

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

assumes *$\text{pmdl } B \subseteq \text{pmdl } A$ and $\bigwedge q. q \in \text{pmdl } A \implies q \in \text{dgrad-p-set } d \ m \implies q \neq 0 \implies \text{is-red } B \ q$*

and *$p \in \text{pmdl } A$ and $p \in \text{dgrad-p-set } d \ m$*

shows *$(\text{red } B)^{**} \ p \ 0$*

<proof>

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

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

assumes *$\text{pmdl } B \subseteq \text{pmdl } A$ and $\bigwedge q. q \in \text{pmdl } A \implies q \neq 0 \implies \text{is-red } B \ q$*

and *$p \in \text{pmdl } A$*

shows *$(\text{red } B)^{**} \ p \ 0$*

<proof>

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

fixes *$G::('t \Rightarrow_0 'b::\text{field}) \ \text{set}$*

assumes *dickson-grading* d **and** $G \subseteq \text{dgrad-p-set } d \ m$ **and** $B \subseteq \text{dgrad-p-set } d \ m$
and $\text{pmdl } G \subseteq \text{pmdl } B$
assumes $\bigwedge f. f \in \text{pmdl } B \implies f \in \text{dgrad-p-set } d \ m \implies f \neq 0 \implies (\exists g \in G. g \neq 0 \wedge \text{lt } g \text{ adds}_t \text{ lt } f)$
shows $\text{pmdl } G = \text{pmdl } B$
 $\langle \text{proof} \rangle$

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

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 $\exists! h. (\text{red } G)^{**} f h \wedge \neg \text{is-red } G h$
 $\langle \text{proof} \rangle$

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

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

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
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

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

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* $G \longleftrightarrow (\forall f \in \text{pmdl } G. f \neq 0 \longrightarrow \text{is-red } G \ f)$

<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* $G \longleftrightarrow (\forall f \in \text{pmdl } G. f \neq 0 \longrightarrow (\exists g \in G. g \neq 0 \wedge \text{lt } g \ \text{adds}_t \ \text{lt } f))$

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

<proof>

lemma *GB-insert:*

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

shows *is-Groebner-basis* $(\text{insert } f \ G)$

<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 dick-*

son-grading-dgrad-dummy dgrad-p-set-exhaust-expl
lemmas *GB-alt-2-finite = GB-alt-2-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *GB-alt-3-finite = GB-alt-3-dgrad-p-set*[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]
lemmas *pmdl-eqI-adds-lt-finite = pmdl-eqI-adds-lt-dgrad-p-set*'[*OF dickson-grading-dgrad-dummy dgrad-p-set-exhaust-expl*]

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

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* ⇔ (∃ *q*. *spoly g1 g2* = (∑ *g* ∈ *G*. *q g* ⊙ *g*) ∧
(∀ *g*. *q g* ∈ *punit.dgrad-p-set d m* ∧
(*q g* ⊙ *g* ≠ 0 ⇒ *lt (q g* ⊙ *g)* <_{*t*} *term-of-pair (lcs (lp g1) (lp g2),*
component-of-term (lt g2)))))

lemma *spoly-repI*:

spoly g1 g2 = (∑ *g* ∈ *G*. *q g* ⊙ *g*) ⇒ (∧ *g*. *q g* ∈ *punit.dgrad-p-set d m*) ⇒
(∧ *g*. *q g* ⊙ *g* ≠ 0 ⇒ *lt (q g* ⊙ *g)* <_{*t*} *term-of-pair (lcs (lp g1) (lp g2),*
component-of-term (lt g2))) ⇒
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* = (∑ *g* ∈ *G*. *q g* ⊙ *g*) **and** ∧ *g*. *q g* ∈ *punit.dgrad-p-set d m*
and ∧ *g*. *q g* ⊙ *g* ≠ 0 ⇒ *lt (q g* ⊙ *g)* <_{*t*} *term-of-pair (lcs (lp g1) (lp g2),*
component-of-term (lt g2))
⟨*proof*⟩

corollary *isGB-D-spoly-rep*:

assumes *dickson-grading d* **and** *is-Groebner-basis G* **and** *G* ⊆ *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* ⊆ *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
 $\langle \text{proof} \rangle$

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 $\text{insert } q\ (G - \{p\}) \subseteq \text{dgrad-p-set } d\ n$
 $\langle \text{proof} \rangle$

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** $lt\ q$
 $\text{adds}_t\ lt\ p$
shows *is-Groebner-basis* $(\text{insert } q\ (G - \{p\}))$ (**is** *is-Groebner-basis* $?G'$)
 $\langle \text{proof} \rangle$

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** $lt\ q\ \text{adds}_t$
 $lt\ p$
shows $\text{pmdl } (\text{insert } q\ (G - \{p\})) = \text{pmdl } G$ (**is** $\text{pmdl } ?G' = \text{pmdl } G$)
 $\langle \text{proof} \rangle$

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* $?G'$)
 $\langle \text{proof} \rangle$

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** $ptoq: \text{red } (G - \{p\})\ p\ q$
shows $\text{pmdl } (\text{insert } q\ (G - \{p\})) = \text{pmdl } G$ (**is** $\text{pmdl } ?G' = -$)
 $\langle \text{proof} \rangle$

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** $ptoq: (\text{red } (G - \{p\}))^{**}\ p\ q$
shows *is-Groebner-basis* $(\text{insert } q\ (G - \{p\}))$
 $\langle \text{proof} \rangle$

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** $ptoq: (\text{red } (G - \{p\}))^{**}\ p\ q$
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 $\text{pmdl (some-GB F)} = \text{pmdl F}$

<proof>

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

assumes *finite F*

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

<proof>

lemma *finite-some-GB-finite*: $\text{finite F} \implies \text{finite (some-GB F)}$

<proof>

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

<proof>

lemma *some-GB-pmdl-finite*: $\text{finite F} \implies \text{pmdl (some-GB F)} = \text{pmdl F}$

<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*:: $(t \Rightarrow_0 'b::\text{field}) \text{ set} \Rightarrow (t \Rightarrow_0 'b) \text{ set} \Rightarrow \text{bool}$ (**infixl** $\sqsupset p$ 50)

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

lemma *red-supsetE*:

assumes $A \sqsupset p B$

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

<proof>

lemma *red-supsetD*:

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

shows $\text{is-red } A p$

<proof>

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

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

shows $A \sqsupset p B$

<proof>

lemma *red-supset-insertI*:
assumes $x \neq 0$ **and** \neg *is-red* $A\ x$
shows $(\text{insert } x\ A) \sqsupseteq_p A$
 \langle *proof* \rangle

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

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

end

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

5.9 Context *od-term*

context *od-term*
begin

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

end

end

6 A General Algorithm Schema for Computing Gröbner Bases

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

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

even his F5 algorithm.

6.1 processed

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

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

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

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

abbreviation *not-in-pair* (**infix** \notin_p 50) **where** $\text{not-in-pair } p \ A \equiv \neg p \in_p A$

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 × 'a) ⇒ 'a list ⇒ ('a × 'a) list ⇒ bool
where *processed* p xs ps ↔ p ∈ set xs × set xs ∧ p ∉_p set ps

lemma *processed-alt*:

processed (a, b) xs ps ↔ ((a ∈ set xs) ∧ (b ∈ set xs) ∧ (a, b) ∉_p set ps)
⟨proof⟩

lemma *processedI*:

assumes a ∈ set xs **and** b ∈ set xs **and** (a, b) ∉_p set ps
shows *processed* (a, b) xs ps
⟨proof⟩

lemma *processedD1*:

assumes *processed* (a, b) xs ps
shows a ∈ set xs
⟨proof⟩

lemma *processedD2*:

assumes *processed* (a, b) xs ps
shows b ∈ set xs
⟨proof⟩

lemma *processedD3*:

assumes *processed* (a, b) xs ps
shows (a, b) ∉_p set ps
⟨proof⟩

lemma *processed-Nil*: *processed* (a, b) xs [] ↔ (a ∈ set xs ∧ b ∈ set xs)
⟨proof⟩

lemma *processed-Cons*:

assumes *processed* (a, b) xs ps
and a1: a = p ⇒ b = q ⇒ thesis
and a2: a = q ⇒ b = p ⇒ thesis
and a3: *processed* (a, b) xs ((p, q) # ps) ⇒ thesis
shows thesis
⟨proof⟩

lemma *processed-minus*:

assumes *processed* (a, b) xs (ps -- qs)
and a1: (a, b) ∈_p set qs ⇒ thesis
and a2: *processed* (a, b) xs ps ⇒ thesis
shows thesis
⟨proof⟩

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
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* \Rightarrow *bool*
where *sel-spec sel* \longleftrightarrow
 $(\forall \text{gs bs ps data. ps} \neq [] \longrightarrow (\text{sel gs bs ps data} \neq [] \wedge \text{set} (\text{sel gs bs ps data}) \subseteq \text{set ps}))$

lemma *sel-specI*:
assumes $\bigwedge \text{gs bs ps data. ps} \neq [] \Longrightarrow (\text{sel gs bs ps data} \neq [] \wedge \text{set} (\text{sel gs bs ps data}) \subseteq \text{set ps})$
shows *sel-spec sel*
 $\langle \text{proof} \rangle$

lemma *sel-specD1*:
assumes *sel-spec sel* **and** $\text{ps} \neq []$
shows $\text{sel gs bs ps data} \neq []$
 $\langle \text{proof} \rangle$

lemma *sel-specD2*:
assumes *sel-spec sel* **and** $\text{ps} \neq []$
shows $\text{set} (\text{sel gs bs ps data}) \subseteq \text{set ps}$
 $\langle \text{proof} \rangle$

6.2.4 Specification of the *add-basis* parameter

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

lemma *ab-specI*:
assumes $\bigwedge \text{gs bs ns data. ns} \neq [] \Longrightarrow \text{set} (\text{ab gs bs ns data}) = \text{set bs} \cup \text{set ns}$
and $\bigwedge \text{gs bs data. ab gs bs [] data} = \text{bs}$
shows *ab-spec ab*
 $\langle \text{proof} \rangle$

lemma *ab-specD1*:
assumes *ab-spec ab*
shows $\text{set} (\text{ab gs bs ns data}) = \text{set bs} \cup \text{set ns}$
 $\langle \text{proof} \rangle$

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** $fst\ (snd\ f) = fst\ (snd\ g)$
 $(snd\ g)$
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 $(h, g) \in_p \text{set } \text{ps} \text{ } \neg_p \text{set } (\text{ap } \text{gs } \text{bs } \text{ps } \text{hs } \text{data})$
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 $\text{set } (\text{ap } \text{gs } \text{bs } \text{ps } \square \ \text{data}) \subseteq \text{set } \text{ps}$
 $\langle \text{proof} \rangle$

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

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

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

<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*)

<proof>

lemma *Keys-full-gb:*

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

lemma *pps-full-gb: pp-of-term ' Keys (fst ' set (full-gb bs)) \subseteq {0}*
<proof>

lemma *components-full-gb:*

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

lemma *full-gb-is-full-pmdl: is-full-pmdl (fst ' set (full-gb bs))*

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

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

lemma *full-gb-isGB: is-Groebner-basis (fst ' set (full-gb bs))*
<proof>

6.2.8 Specification of the *completion* parameter

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

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

lemma *compl-structI:*

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

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-struct compl)

lemma *struct-specI:*

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

lemma *struct-specD1:*

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

lemma *struct-specD2:*

assumes *struct-spec sel ap ab compl*
shows *ap-spec ap*
 \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*
 $\text{sps} \subseteq \text{set ps} \implies$
 $\text{unique-idx (gs @ bs) data} \implies$
 $\text{fst ' (set (fst (compl gs bs (ps -- sps) sps data)))} \subseteq \text{pmdl (args-to-set}$
 $(gs, bs, ps))$
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*
 $d m \longrightarrow$
 $\text{is-Groebner-basis (fst ' set gs) \longrightarrow fst ' set bs} \subseteq \text{dgrad-p-set } d m \longrightarrow$
 $\text{set ps} \subseteq \text{set bs} \times (\text{set gs} \cup \text{set bs}) \longrightarrow \text{sps} \neq [] \longrightarrow \text{set sps} \subseteq \text{set ps} \longrightarrow$
 $\text{unique-idx (gs @ bs) data} \longrightarrow (p, q) \in \text{set sps} \longrightarrow \text{fst } p \neq 0 \longrightarrow \text{fst } q$
 $\neq 0 \longrightarrow$
 $\text{crit-pair-cbelow-on } d m (\text{fst ' (set gs} \cup \text{set bs)} \cup \text{fst ' set (fst (compl gs$

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

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 $\bigwedge d m gs bs ps sps p q data. dickson\text{-}grading\ d \implies fst \text{ ' } set\ gs \subseteq dgrad\text{-}p\text{-}set\ d\ m \implies$
 $dgrad\text{-}p\text{-}set\ d\ m \implies$
 $is\text{-}Groebner\text{-}basis\ (fst \text{ ' } set\ gs) \implies fst \text{ ' } set\ bs \subseteq dgrad\text{-}p\text{-}set\ d\ m \implies$
 $set\ ps \subseteq set\ bs \times (set\ gs \cup set\ bs) \implies sps \neq [] \implies set\ sps \subseteq set\ ps \implies$
 $unique\text{-}idx\ (gs @ bs) data \implies (p, q) \in set\ sps \implies fst\ p \neq 0 \implies fst\ q \neq$
 $0 \implies$
 $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst \text{ ' } (set\ gs \cup set\ bs) \cup fst \text{ ' } set\ (fst\ (compl\ gs\ bs\ (ps \text{ --- } sps)\ sps\ data)))) (fst\ p)\ (fst\ q)$
shows *compl-conn compl*
 $\langle proof \rangle$

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 union set bs) union fst ' set (fst (compl gs bs (ps --- sps) sps data)))) (fst p) (fst q)*
 $\langle proof \rangle$

6.2.9 Function *gb-schema-dummy*

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

where $[code\ del]:\ add\text{-}indices\ ns\ data =$
 $(map\text{-}idx\ (\lambda h\ i.\ (fst\ h,\ i,\ snd\ h))\ (fst\ ns)\ (fst\ data),\ fst\ data + length\ (fst\ ns),\ snd\ ns)$

lemma $(in \text{ ---})\ add\text{-}indices\text{-}code\ [code]:$

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

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

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

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* ::

$(('a \Rightarrow \text{nat}) \Rightarrow ('t, 'b::\text{field}, 'c) \text{pdata list} \Rightarrow (((('t, 'b, 'c) \text{pdata list} \times ('t, 'b, 'c)$
 $\text{pdata-pair list}) \times$
 $((('t, 'b, 'c) \text{pdata list} \times ('t, 'b, 'c) \text{pdata-pair list})) \text{set}$
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 $\text{dgrad-p-set-le } d (\text{args-to-set } (\text{gs}, \text{ab } \text{gs } \text{bs } \text{hs } \text{data}', \text{ap } \text{gs } \text{bs } (\text{ps} \text{ --- } \text{sps})$
 $\text{hs } \text{data}')) (\text{args-to-set } (\text{gs}, \text{bs}, \text{ps}))$
 $(\text{is } \text{dgrad-p-set-le } - ?l ?r)$
 $\langle \text{proof} \rangle$

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-idx* $(gs\ @\ hs)\ data'$
 $\langle proof \rangle$

corollary *unique-idx-ab:*

assumes *ab-spec* ab **and** *unique-idx* $(gs\ @\ bs)\ data$ **and** $(hs,\ data') = add_indices$

aux data

shows *unique-idx* (*gs @ ab gs bs hs data'*) *data'*
{*proof*}

lemma *rem-comps-spec-struct*:

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

lemma *pmdl-struct*:

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

lemma *discarded-subset*:

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

lemma *compl-struct-disjoint*:

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

context

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

begin

function (*domintros*) *gb-schema-dummy* :: *nat* \times *nat* \times 'd \Rightarrow ('t, 'b, 'c) *pdata-pair*
set \Rightarrow

$(ap\ gs\ bs\ ps0\ hs\ data')$
 $(ab\ gs\ bs\ hs\ data')\ (ap\ gs\ bs\ ps0\ hs\ data')$
 $)$
 $)$
 $\langle proof \rangle$

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

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

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* ($fst\ 'set\ (fst\ (gb\ schema\ dummy\ data\ D\ bs\ ps))$) (*args-to-set*
 $(gs,\ bs,\ ps)$)
 $\langle proof \rangle$

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

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

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

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

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** $res = gb\text{-}schema\text{-}dummy$
data D bs ps
shows $snd\ res \subseteq set\ (fst\ res) \times set\ (fst\ res) \vee (\exists xs. fst\ (res) = full\text{-}gb\ xs)$
<proof>

lemma *gb-schema-dummy-connectible1*:

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

lemma *gb-schema-dummy-connectible2*:

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

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

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* $(gs @ bs) (\text{snd data})$
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 $f \in \text{set } (\text{fst } (\text{gb-schema-dummy data } D \text{ bs ps}))$
and $g \in \text{set } (\text{fst } (\text{gb-schema-dummy data } D \text{ bs ps}))$
and $\text{fst } f \neq 0$ **and** $\text{fst } g \neq 0$
shows $\text{crit-pair-cbelow-on } d \text{ m} (\text{fst } ' \text{ set } (\text{fst } (\text{gb-schema-dummy data } D \text{ bs ps})))$
 $(\text{fst } f) (\text{fst } g)$
 $\langle \text{proof} \rangle$

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

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

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 $\text{fst } ' \text{ set } (\text{fst } (\text{gb-schema-dummy } (rc, \text{data}) D (\text{ab } gs \square bs \text{ data}) (\text{ap } gs \square \square bs \text{ data})))) \subseteq \text{dgrad-p-set } d \text{ m}$
 $\langle \text{proof} \rangle$

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

fixes $bs \text{ data}$
defines $bs0 \equiv \text{ab } gs \square bs \text{ data}$
defines $ps0 \equiv \text{ap } gs \square \square bs \text{ data}$

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

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

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

6.2.10 Function *gb-schema-aux*

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

<proof>

The *data* parameter of *gb-schema-aux* is a triple (c, i, d) , where c is the number of components *cmp* of the input list for which the current basis $gs @ bs$ does *not* yet contain an element whose leading power-product is 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 $\text{fst } (gb\text{-schema-dummy data } D \text{ bs ps}) = gb\text{-schema-aux data bs ps}$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

assumes *struct-spec sel ap ab compl*
shows $\text{component-of-term ' Keys (fst ' set (gb-schema-aux (rc, data) (ab gs [] bs data) (ap gs [] [] bs data)))} =$
 $\text{component-of-term ' Keys (fst ' set (gs @ bs))}$
 $\langle \text{proof} \rangle$

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 $pmdl (fst\ 'set\ (gb\ schema\ aux\ (rc, data)\ (ab\ gs\ []\ bs\ data)\ (ap\ gs\ []\ []\ bs\ data))) =$
 $pmdl (fst\ '(set\ (gs\ @\ bs)))$
 $\langle proof \rangle$

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* $(fst\ 'set\ (gb\ schema\ aux\ (rc, data)\ (ab\ gs\ []\ bs\ data)\ (ap\ gs\ []\ []\ bs\ data)))$

$\langle proof \rangle$

end

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

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

$(t, 'b, 'c, 'd)\ complT \Rightarrow (t, 'b, 'c)\ pdata' list \Rightarrow 'd \Rightarrow$
 $(t, 'b::field, 'c::default)\ pdata' list$

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* $:: (t, 'b, 'c, 'd)\ selT \Rightarrow (t, 'b, 'c, 'd)\ apT \Rightarrow (t, 'b, 'c, 'd)\ abT \Rightarrow$

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

$((t, 'b, 'c)\ pdata\ list \Rightarrow (t, 'b, 'c)\ pdata \Rightarrow 'd \Rightarrow 'd) \Rightarrow$

$(t, 'b, 'c)\ pdata' list \Rightarrow 'd \Rightarrow (t, 'b::field, 'c::default)$

$pdata' list$

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*:
 $\text{fst } \langle \lambda(f, -, d). (f, d) \rangle \text{ } \langle A = \text{fst } \langle A \text{ for } A::('x \times 'y \times 'z) \text{ set} \rangle \text{ set} \rangle$
 $\langle \text{proof} \rangle$

lemma *fst-gb-schema-direct*:
 $\text{fst } \langle \text{set } (\text{gb-schema-direct sel ap ab compl bs0 data0}) =$
 $(\text{let data} = (\text{length bs0}, \text{data0}); \text{bs1} = \text{fst } (\text{add-indices } (\text{bs0}, \text{data0}) (0, \text{data0}));$
 $\text{bs} = \text{ab } \square \square \text{ bs1 data in}$
 $\text{fst } \langle \text{set } (\text{gb-schema-aux sel ap ab compl } \square (\text{count-rem-components bs}, \text{data})$
 $\text{bs } (\text{ap } \square \square \text{ bs1 data}))$
 \rangle
 $\langle \text{proof} \rangle$

lemma *gb-schema-direct-dgrad-p-set*:
assumes *dickson-grading d and struct-spec sel ap ab compl and fst ' set bs \subseteq dgrad-p-set d m*
shows $\text{fst } \langle \text{set } (\text{gb-schema-direct sel ap ab compl bs data}) \subseteq \text{dgrad-p-set d m}$
 $\langle \text{proof} \rangle$

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))*
 $\langle \text{proof} \rangle$

theorem *gb-schema-direct-pmdl*:
assumes *struct-spec sel ap ab compl and compl-pmdl compl*
shows $\text{pmdl } (\text{fst } \langle \text{set } (\text{gb-schema-direct sel ap ab compl bs data}) \rangle) = \text{pmdl } (\text{fst } \langle \text{set bs} \rangle)$
 $\langle \text{proof} \rangle$

lemma *fst-gb-schema-incr*:
 $\text{fst } \langle \text{set } (\text{gb-schema-incr sel ap ab compl upd } (b0 \# \text{bs}) \text{ data}) =$
 $(\text{let } (gs, n, \text{data}') = \text{add-indices } (\text{gb-schema-incr sel ap ab compl upd bs data},$
 $\text{data}) (0, \text{data});$
 $b = (\text{fst } b0, n, \text{snd } b0); \text{data}'' = \text{upd } gs \ b \ \text{data}' \ \text{in}$
 $\text{fst } \langle \text{set } (\text{gb-schema-aux sel ap ab compl } gs \ (\text{count-rem-components } (b \# \text{gs}),$
 $\text{Suc } n, \text{data}'')$
 $(\text{ab } gs \ \square \ [b] \ (\text{Suc } n, \text{data}'')) \ (\text{ap } gs \ \square \ \square \ [b] \ (\text{Suc } n, \text{data}''))$
 \rangle
 $\langle \text{proof} \rangle$

lemma *gb-schema-incr-dgrad-p-set*:
assumes *dickson-grading d and struct-spec sel ap ab compl*
and *fst ' set bs \subseteq dgrad-p-set d m*
shows $\text{fst } \langle \text{set } (\text{gb-schema-incr sel ap ab compl upd bs data}) \subseteq \text{dgrad-p-set d m}$
 $\langle \text{proof} \rangle$

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

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

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

definition *icrit-spec* :: ($'t, 'b::\text{field}, 'c, 'd$) *icritT* \Rightarrow *bool*

where *icrit-spec crit* \longleftrightarrow
 $(\forall d m \text{ data } gs \ bs \ hs \ p \ q. \text{dickson-grading } d \longrightarrow$
 $\text{fst ' (set } gs \cup \text{ set } bs \cup \text{ set } hs) \subseteq \text{dgrad-p-set } d \ m \longrightarrow \text{unique-idx } (gs \ @$
 $bs \ @ \ hs) \ \text{data} \longrightarrow$
 $\text{is-Groebner-basis } (fst \ ' \ \text{set } gs) \longrightarrow p \in \text{set } hs \longrightarrow q \in \text{set } gs \cup \text{set } bs \cup$
 $\text{set } hs \longrightarrow$
 $\text{fst } p \neq 0 \longrightarrow \text{fst } q \neq 0 \longrightarrow \text{crit } \text{data } gs \ bs \ hs \ p \ q \longrightarrow$
 $\text{crit-pair-cbelow-on } d \ m \ (fst \ ' \ (\text{set } gs \cup \text{set } bs \cup \text{set } hs)) \ (fst \ p) \ (fst \ q))$

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

definition *ncrit-spec* :: ($'t, 'b::\text{field}, 'c, 'd$) *ncritT* \Rightarrow *bool*

where *ncrit-spec crit* \longleftrightarrow
 $(\forall d m \text{ data } gs \ bs \ hs \ ps \ B \ q\text{-in-bs } p \ q. \text{dickson-grading } d \longrightarrow \text{set } gs \cup \text{set}$

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

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

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

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 $\bigwedge d\ m\ data\ gs\ bs\ hs\ p\ q.$
 $dickson\text{-}grading\ d \Longrightarrow fst\ ' (set\ gs \cup set\ bs \cup set\ hs) \subseteq dgrad\text{-}p\text{-}set\ d\ m$
 \Longrightarrow
 $unique\text{-}idx\ (gs\ @\ bs\ @\ hs)\ data \Longrightarrow is\text{-}Groebner\text{-}basis\ (fst\ ' set\ gs) \Longrightarrow$
 $p \in set\ hs \Longrightarrow q \in set\ gs \cup set\ bs \cup set\ hs \Longrightarrow fst\ p \neq 0 \Longrightarrow fst\ q \neq 0$
 \Longrightarrow
 $crit\ data\ gs\ bs\ hs\ p\ q \Longrightarrow$
 $crit\text{-}pair\text{-}cbelow\text{-}on\ d\ m\ (fst\ ' (set\ gs \cup set\ bs \cup set\ hs))\ (fst\ p)\ (fst\ q)$
shows *icrit-spec crit*
 $\langle proof \rangle$

lemma *icrit-specD*:

assumes *icrit-spec crit* **and** *dickson-grading d*
and $fst\ ' (set\ gs \cup set\ bs \cup set\ hs) \subseteq dgrad\text{-}p\text{-}set\ d\ m$ **and** $unique\text{-}idx\ (gs\ @\ bs$
 $@\ hs)\ data$
and *is-Groebner-basis (fst ' set gs)* **and** $p \in set\ hs$ **and** $q \in set\ gs \cup set\ bs \cup$
 $set\ hs$
and $fst\ p \neq 0$ **and** $fst\ q \neq 0$ **and** $crit\ data\ gs\ bs\ hs\ p\ q$

shows *crit-pair-cbelow-on* $d m$ (*fst* ' (*set* $gs \cup set bs \cup set hs$)) (*fst* p) (*fst* q)
 ⟨*proof*⟩

lemma *ncrit-specI*:

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

lemma *ncrit-specD*:

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

lemma *ocrit-specI*:

assumes $\bigwedge d m$ *data* $hs ps B p q$.
dickson-grading $d \implies set hs \subseteq B \implies fst$ ' $B \subseteq dgrad-p-set d m \implies$
unique-idx ($p \# q \# hs @ (map (fst \circ snd) ps) @ (map (snd \circ snd)$
 $ps)$) *data* \implies
 ($\bigwedge p' q'. (p', q') \in_p snd$ ' $set ps \implies fst p' \neq 0 \implies fst q' \neq 0 \implies$
crit-pair-cbelow-on $d m$ (*fst* ' B) (*fst* p') (*fst* q')) \implies
 $p \in B \implies q \in B \implies fst p \neq 0 \implies fst q \neq 0 \implies$
crit data $hs ps p q \implies crit-pair-cbelow-on d m$ (*fst* ' B) (*fst* p) (*fst* q)
shows *ocrit-spec crit*
 ⟨*proof*⟩

lemma *ocrit-specD*:

assumes *ocrit-spec crit* **and** *dickson-grading d* **and** *set hs* $\subseteq B$ **and** *fst ' B* \subseteq
dgrad-p-set d m
and *unique-idx* (*p # q # hs* @ (*map* (*fst* \circ *snd*) *ps*) @ (*map* (*snd* \circ *snd*) *ps*))
data
and $\bigwedge p' q'. (p', q') \in_p \textit{snd ' set ps} \implies \textit{fst p'} \neq 0 \implies \textit{fst q'} \neq 0 \implies$
 $\textit{crit-pair-cbelow-on d m (fst ' B) (fst p') (fst q')}$
and $p \in B$ **and** $q \in B$ **and** $\textit{fst p} \neq 0$ **and** $\textit{fst q} \neq 0$
and *crit data hs ps p q*
shows *crit-pair-cbelow-on d m (fst ' B) (fst p) (fst q)*
 $\langle \textit{proof} \rangle$

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)
 $\langle \textit{proof} \rangle$

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*)
 $\langle \textit{proof} \rangle$

component-crit and *product-crit* ignore the *data* parameter.

fun (**in** $-$) *pair-in-list* :: ($\textit{bool} \times ('a, 'b, 'c)$ *pdata-pair*) *list* \Rightarrow $\textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{bool}$
where

pair-in-list [] - - = *False*
 $\textit{pair-in-list} ((-, (-, i', -), (-, j', -)) \# \textit{ps}) i j =$
 $((i = i' \wedge j = j') \vee (i = j' \wedge j = i') \vee \textit{pair-in-list ps i j})$

lemma (**in** $-$) *pair-in-listE*:

assumes *pair-in-list ps i j*
obtains $p q a b$ **where** $((p, i, a), (q, j, b)) \in_p \textit{snd ' set ps}$
 $\langle \textit{proof} \rangle$

definition *chain-ncrit* :: (*t, 'b::zero, 'c, 'd*) *ncritT*

where *chain-ncrit data gs bs hs q-in-bs ps p q* \longleftrightarrow
 $(\textit{let } v = \textit{lt} (\textit{fst } p); l = \textit{term-of-pair} (\textit{lcs} (\textit{pp-of-term } v) (\textit{lp} (\textit{fst } q))),$
component-of-term v);
 $i = \textit{fst} (\textit{snd } p); j = \textit{fst} (\textit{snd } q)$ *in*

$$\begin{aligned}
& (\exists r \in \text{set } gs. \text{ let } k = \text{fst } (\text{snd } r) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt } (\text{fst } r) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge (q\text{-in-bs} \vee \\
& \text{pair-in-list } ps \ j \ k) \wedge \text{fst } r \neq 0) \vee \\
& (\exists r \in \text{set } bs. \text{ let } k = \text{fst } (\text{snd } r) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt } (\text{fst } r) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge (q\text{-in-bs} \vee \\
& \text{pair-in-list } ps \ j \ k) \wedge \text{fst } r \neq 0) \vee \\
& (\exists h \in \text{set } hs. \text{ let } k = \text{fst } (\text{snd } h) \text{ in} \\
& \quad k \neq i \wedge k \neq j \wedge \text{lt } (\text{fst } h) \text{ adds}_t l \wedge \text{pair-in-list } ps \ i \ k \wedge \text{pair-in-list} \\
& \text{ps } j \ k \wedge \text{fst } h \neq 0)
\end{aligned}$$

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** $\text{snd } ' \text{ set } ps \subseteq \text{set } hs \times (\text{set } gs \cup \text{set } bs \cup \text{set } hs)$
and *unique-idx (gs @ bs @ hs) data* **and** $p \in \text{set } hs$ **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 $(p, r) \in_p \text{snd } ' \text{ set } ps$ **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 $(p, h) \in_p \text{snd } ' \text{ set } ps$ **and** $(q, h) \in_p \text{snd } ' \text{ set } ps$
 $\langle \text{proof} \rangle$

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* ⇒ (*'t, 'b, 'c*) *pdata list* ⇒ (*'t, 'b, 'c*) *pdata list* ⇒
 (*'t, 'b, 'c*) *pdata* ⇒ (*bool* × (*'t, 'b, 'c*) *pdata-pair*) *list* ⇒
 (*bool* × (*'t, 'b, 'c*) *pdata-pair*) *list*

type-synonym (**in** -) (*'t, 'b, 'c, 'd*) *npT* = (*'t, 'b, 'c*) *pdata list* ⇒ (*'t, 'b, 'c*) *pdata list* ⇒
 (*'t, 'b, 'c*) *pdata list* ⇒ *nat* × *'d* ⇒
 (*bool* × (*'t, 'b, 'c*) *pdata-pair*) *list*

definition *np-spec* :: (*'t, 'b, 'c, 'd*) *npT* ⇒ *bool*
where *np-spec np* ↔ (∀ *gs bs hs data*.
 snd ' set (np gs bs hs data) ⊆ set hs × (set gs ∪ set bs ∪ set
hs) ∧
 set hs × (set gs ∪ set bs) ⊆ snd ' set (np gs bs hs data) ∧
 (∀ *a b. a ∈ set hs → b ∈ set hs → a ≠ b → (a, b) ∈_p*
snd ' set (np gs bs hs data)) ∧
 (∀ *p q. (True, p, q) ∈ set (np gs bs hs data) → q ∈ set gs ∪*
set bs))

lemma *np-specI*:
assumes ∧*gs bs hs data*.
 snd ' set (np gs bs hs data) ⊆ set hs × (set gs ∪ set bs ∪ set hs) ∧
 set hs × (set gs ∪ set bs) ⊆ snd ' set (np gs bs hs data) ∧
 (∀ *a b. a ∈ set hs → b ∈ set hs → a ≠ b → (a, b) ∈_p snd ' set (np*
gs bs hs data)) ∧
 (∀ *p q. (True, p, q) ∈ set (np gs bs hs data) → q ∈ set gs ∪ set bs*)
shows *np-spec np*
 ⟨*proof*⟩

lemma *np-specD1*:
assumes *np-spec np*
shows *snd ' set (np gs bs hs data) ⊆ set hs × (set gs ∪ set bs ∪ 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* = $ps \text{ @ } (map\ (\lambda g. (flag, h, g))\ gs) \text{ @ } (map\ (\lambda b. (flag, h, b))\ bs)$

lemma *set-add-pairs-single-naive*:

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

fun *add-pairs-single-sorted* :: $((bool \times ('t, 'b, 'c) pdata\text{-pair}) \Rightarrow (bool \times ('t, 'b, 'c) pdata\text{-pair})) \Rightarrow bool \Rightarrow$

$(('t, 'b::zero, 'c) apsT \text{ where}$
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 $\square \square bs h (insort\text{-wrt rel } (flag, h, b) ps) |$
add-pairs-single-sorted rel flag $(g \# gs) bs h ps =$
add-pairs-single-sorted rel flag gs bs h (insort\text{-wrt rel } (flag, h, g) ps)

lemma *set-add-pairs-single-sorted*:

$set\ (add\text{-pairs-single-sorted rel flag gs bs h ps) = set\ ps \cup Pair\ flag \text{ ' } (\{h\} \times (set\ gs \cup set\ bs))$
 ⟨proof⟩

primrec (**in** $-$) *pairs* :: $('t, 'b, 'c) apsT \Rightarrow bool \Rightarrow ('t, 'b, 'c) pdata\ list \Rightarrow (bool \times ('t, 'b, 'c) pdata\text{-pair}) list$

where
pairs - - $\square = \square |$
pairs aps flag $(x \# xs) = aps\ flag \square xs\ x (pairs\ aps\ flag\ xs)$

lemma *pairs-subset*:

assumes $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ '(\{h\} \times (set\ gs \cup set\ bs))$
shows $set\ (pairs\ aps\ flag\ xs) \subseteq Pair\ flag\ ' (set\ xs \times set\ xs)$
 $\langle proof \rangle$

lemma *in-pairsI*:

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

corollary *in-pairsI'*:

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

definition *new-pairs-naive* $:: (t, 'b::zero, 'c, 'd)\ npT$

where *new-pairs-naive* $gs\ bs\ hs\ data =$
 $fold\ (add-pairs-single-naive\ data\ True\ gs\ bs)\ hs\ (pairs\ (add-pairs-single-naive\ data)\ False\ hs)$

definition *new-pairs-sorted* $:: (nat \times 'd \Rightarrow (bool \times (t, 'b, 'c)\ pdata-pair) \Rightarrow (bool \times (t, 'b, 'c)\ pdata-pair) \Rightarrow bool) \Rightarrow$
 $(t, 'b::zero, 'c, 'd)\ npT$

where *new-pairs-sorted* $rel\ gs\ bs\ hs\ data =$
 $fold\ (add-pairs-single-sorted\ (rel\ data)\ True\ gs\ bs)\ hs\ (pairs\ (add-pairs-single-sorted\ (rel\ data))\ False\ hs)$

lemma *set-fold-aps*:

assumes $\bigwedge gs\ bs\ h\ ps.\ set\ (aps\ flag\ gs\ bs\ h\ ps) = set\ ps \cup Pair\ flag\ '(\{h\} \times (set\ gs \cup set\ bs))$
shows $set\ (fold\ (aps\ flag\ gs\ bs)\ hs\ ps) = Pair\ flag\ ' (set\ hs \times (set\ gs \cup set\ bs)) \cup set\ ps$
 $\langle proof \rangle$

lemma *set-new-pairs-naive*:

$set\ (new-pairs-naive\ gs\ bs\ hs\ data) =$
 $Pair\ True\ ' (set\ hs \times (set\ gs \cup set\ bs)) \cup set\ (pairs\ (add-pairs-single-naive\ data)\ False\ hs)$
 $\langle proof \rangle$

lemma *set-new-pairs-sorted*:

$set\ (new-pairs-sorted\ rel\ gs\ bs\ hs\ data) =$
 $Pair\ True\ ' (set\ hs \times (set\ gs \cup set\ bs)) \cup set\ (pairs\ (add-pairs-single-sorted\ (rel\ data))\ False\ hs)$

<proof>

lemma (*in -*) *fst-snd-Pair* [*simp*]:

shows $\text{fst} \circ \text{Pair } x = (\lambda-. x)$ **and** $\text{snd} \circ \text{Pair } x = \text{id}$

<proof>

lemma *np-spec-new-pairs-naive*: *np-spec new-pairs-naive*

<proof>

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

<proof>

new-pairs-naive gs bs hs data and *new-pairs-sorted rel gs bs hs data* return lists of triples $(q\text{-in-}bs, p, q)$, 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* $\Rightarrow \text{bool}$

where *canon-pair-order-aux p q* \longleftrightarrow
 $(\text{lcs } (\text{lp } (\text{fst } (\text{fst } p))) (\text{lp } (\text{fst } (\text{snd } p)))) \preceq \text{lcs } (\text{lp } (\text{fst } (\text{fst } q))) (\text{lp } (\text{fst } (\text{snd } q))))$

abbreviation *canon-pair-order data p q* $\equiv \text{canon-pair-order-aux } (\text{snd } p) (\text{snd } q)$

abbreviation *canon-pair-comb* $\equiv \text{merge-wrt canon-pair-order-aux}$

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* = $(\text{let } c = \text{crit data gs bs hs in map } (\lambda(q\text{-in-}bs, p, q). (c p q, q\text{-in-}bs, p, q))) ps$

lemma *fst-apply-icrit*:

assumes *icrit-spec crit* **and** *dickson-grading d*

and $\text{fst } \text{' } (\text{set } gs \cup \text{set } bs \cup \text{set } hs) \subseteq \text{dgrad-p-set } d m$ **and** *unique-idx (gs @ bs @ hs) data*

and *is-Groebner-basis (fst ' 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** $(\text{True}, q\text{-in-}bs, p, q) \in \text{set } (\text{apply-icrit crit data gs bs hs ps})$

shows *crit-pair-cbelow-on d m (fst ' (set gs @ set bs @ set hs)) (fst p) (fst q)*

<proof>

lemma *snd-apply-icrit* [*simp*]: $\text{map snd } (\text{apply-icrit crit data gs bs hs ps}) = ps$

<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* =
 (let $c = \text{crit data gs bs hs in}$
 $\text{rev (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs ps' p q then ps'}$
 $\text{else } (ic, p, q) \# ps') ps \text{ []])}$)

lemma *apply-ncrit-append*:

apply-ncrit crit data gs bs hs (xs @ ys) =
 $\text{rev (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge \text{crit data gs bs hs q-in-bs ps' p q}$
 $\text{then ps' else } (ic, p, q) \# ps') ys$
 $(\text{rev (apply-ncrit crit data gs bs hs xs))})$
 ⟨proof⟩

lemma *fold-superset*:

$\text{set acc} \subseteq$
 $\text{set (fold } (\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs ps' p q then ps' else } (ic,$
 $p, q) \# ps') ps \text{ acc})$
 ⟨proof⟩

lemma *apply-ncrit-superset*:

$\text{set (apply-ncrit crit data gs bs hs ps)} \subseteq \text{set (apply-ncrit crit data gs bs hs (ps @$
 $qs))}$ (is $?l \subseteq ?r$)
 ⟨proof⟩

lemma *apply-ncrit-subset-aux*:

assumes $(ic, p, q) \in \text{set (fold}$
 $(\lambda(ic, q\text{-in-bs}, p, q). \lambda ps'. \text{if } \neg ic \wedge c \text{ q-in-bs ps' p q then ps' else } (ic, p,$
 $q) \# ps') ps \text{ acc})$
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 $(\text{in } -) \ \text{set}E$:

assumes $x \in \text{set } xs$
obtains $ys \ zs$ **where** $xs = ys \ @ \ (x \ # \ zs)$
 $\langle \text{proof} \rangle$

lemma $\text{apply-ncrit-connectible}$:

assumes ncrit-spec crit **and** $\text{dickson-grading } d$
and $\text{set } gs \cup \text{set } bs \cup \text{set } hs \subseteq B$ **and** $\text{fst } 'B \subseteq \text{dgrad-}p\text{-set } d \ m$
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** $\text{unique-idx } (gs \ @ \ bs \ @ \ hs) \ \text{data}$
and $\text{is-Groebner-basis } (\text{fst } ' \text{set } gs)$
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 $\text{apply-ocrit} :: ('t, 'b, 'c, 'd) \ \text{ocrit}T \Rightarrow (\text{nat} \times 'd) \Rightarrow ('t, 'b, 'c) \ \text{pdata list} \Rightarrow$
 $\Rightarrow (\text{bool} \times ('t, 'b, 'c) \ \text{pdata-pair}) \ \text{list} \Rightarrow ('t, 'b, 'c) \ \text{pdata-pair list} \Rightarrow$
 $(('t, 'b, 'c) \ \text{pdata-pair list})$
where $\text{apply-ocrit crit data } hs \ ps' \ ps = (\text{let } c = \text{crit data } hs \ ps' \ \text{in } [(p, q) \leftarrow ps \ . \ \neg \ c \ p \ q])$

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 $\text{set-apply-ocrit-iff}$:

$(p, q) \in \text{set } (\text{apply-ocrit crit data } hs \ ps' \ ps) \longleftrightarrow ((p, q) \in \text{set } ps \wedge \neg \ \text{crit data } hs \ ps' \ p \ q)$
 $\langle \text{proof} \rangle$

lemma $\text{apply-ocrit-connectible}$:

assumes ocrit-spec crit **and** $\text{dickson-grading } d$ **and** $\text{set } hs \subseteq B$ **and** $\text{fst } 'B \subseteq$

dgrad-p-set $d\ m$
and *unique-idx* $(p \# q \# hs \ @ \ (map \ (fst \circ \ snd) \ ps') \ @ \ (map \ (snd \circ \ snd) \ ps'))$
data
and $\bigwedge p' \ q'. \ (p', \ q') \in \ snd \ ' \ set \ ps' \implies \ fst \ p' \neq 0 \implies \ fst \ q' \neq 0 \implies$
 $\quad \textit{crit-pair-cbelow-on} \ d \ m \ (fst \ ' \ B) \ (fst \ p') \ (fst \ q')$
assumes $p \in B$ **and** $q \in B$ **and** $\textit{fst} \ p \neq 0$ **and** $\textit{fst} \ q \neq 0$
and $(p, \ q) \in \ set \ ps$ **and** $(p, \ q) \notin \ set \ (apply-ocrit \ crit \ data \ hs \ ps' \ ps)$
shows $\textit{crit-pair-cbelow-on} \ d \ m \ (fst \ ' \ B) \ (fst \ p) \ (fst \ q)$
 $\langle proof \rangle$

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 $comb::(t, 'b, 'c) \ pdata-pair \ list \Rightarrow (t, 'b, 'c) \ pdata-pair \ list \Rightarrow (t, 'b, 'c)$
 $\textit{pdata-pair} \ list$
begin

definition *add-pairs* $:: (t, 'b, 'c, 'd) \ apT$

where $\textit{add-pairs} \ gs \ bs \ ps \ hs \ data =$
 $(let \ ps1 = \ apply-ncrit \ ncrit \ data \ gs \ bs \ hs \ (apply-icrit \ icrit \ data \ gs \ bs \ hs \ (np$
 $\textit{gs} \ bs \ hs \ data));$
 $\quad \textit{ps2} = \ apply-ocrit \ ocrit \ data \ hs \ ps1 \ ps \ in \ comb \ (map \ snd \ [x \leftarrow \ ps1 \ . \ \neg$
 $\textit{fst} \ x]) \ ps2)$

lemma *set-add-pairs*:

assumes $\bigwedge xs \ ys. \ set \ (comb \ xs \ ys) = set \ xs \cup \ set \ ys$
assumes $ps1 = \ apply-ncrit \ ncrit \ data \ gs \ bs \ hs \ (apply-icrit \ icrit \ data \ gs \ bs \ hs \ (np$
 $\textit{gs} \ bs \ hs \ data))$
shows $set \ (\textit{add-pairs} \ gs \ bs \ ps \ hs \ data) =$
 $\{(p, \ q) \mid p \ q. \ (False, \ p, \ q) \in \ set \ ps1 \vee ((p, \ q) \in \ set \ ps \wedge \neg \textit{ocrit} \ data \ hs$
 $\textit{ps1} \ p \ q)\}$
 $\langle proof \rangle$

lemma *set-add-pairs-iff*:

assumes $\bigwedge xs \ ys. \ set \ (comb \ xs \ ys) = set \ xs \cup \ set \ ys$
assumes $ps1 = \ apply-ncrit \ ncrit \ data \ gs \ bs \ hs \ (apply-icrit \ icrit \ data \ gs \ bs \ hs \ (np$
 $\textit{gs} \ bs \ hs \ data))$
shows $((p, \ q) \in \ set \ (\textit{add-pairs} \ gs \ bs \ ps \ hs \ data)) \longleftrightarrow$
 $((False, \ p, \ q) \in \ set \ ps1 \vee ((p, \ q) \in \ set \ ps \wedge \neg \textit{ocrit} \ data \ hs \ \textit{ps1} \ p \ q))$
 $\langle proof \rangle$

lemma *ap-spec-add-pairs*:

assumes $np\text{-spec} \ np$ **and** $icrit\text{-spec} \ icrit$ **and** $ncrit\text{-spec} \ ncrit$ **and** $ocrit\text{-spec} \ ocrit$
and $\bigwedge xs \ ys. \ set \ (comb \ xs \ ys) = set \ xs \cup \ set \ ys$
shows $ap\text{-spec} \ \textit{add-pairs}$

<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

$(\forall gs\ bs\ ps\ sps\ data.$

$0 \notin \text{fst } ' \text{ set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data)) \wedge$

$(\forall h\ b. h \in \text{set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data)) \longrightarrow b \in \text{set } gs \cup \text{set } bs \longrightarrow$

$\text{fst } b \neq 0 \longrightarrow$

$\neg \text{lt } (\text{fst } b) \text{ adds}_t \text{ lt } (\text{fst } h) \wedge$

$(\forall d. \text{dickson-grading } d \longrightarrow$

$dgrad\text{-}p\text{-set-le } d\ (\text{fst } ' \text{ set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data)))\ (\text{args-to-set}$

$(gs, bs, sps))) \wedge$

$\text{component-of-term } ' \text{ Keys } (\text{fst } ' (\text{set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data)))) \subseteq$

$\text{component-of-term } ' \text{ Keys } (\text{args-to-set } (gs, bs, sps)) \wedge$

$(\text{is-Groebner-basis } (\text{fst } ' \text{ set } gs) \longrightarrow \text{unique-idx } (gs @ bs) \text{ data} \longrightarrow$

$(\text{fst } ' \text{ set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data))) \subseteq \text{pmdl } (\text{args-to-set } (gs, bs, sps))$

\wedge

$(\forall (p, q) \in \text{set } sps. \text{set } sps \subseteq \text{set } bs \times (\text{set } gs \cup \text{set } bs) \longrightarrow$

$(\text{red } (\text{fst } ' (\text{set } gs \cup \text{set } bs)) \cup \text{fst } ' \text{ set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data))))^{**}$

$(\text{spoly } (\text{fst } p) (\text{fst } q) 0))))$

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 $\text{set } bs \cup \text{set } gs$, whose leading terms are not divisible by the leading term of any non-zero $b \in \text{set } bs$, 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 $\bigwedge gs\ bs\ ps\ sps\ data. 0 \notin \text{fst } ' \text{ set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data))$

assumes $\bigwedge gs\ bs\ ps\ sps\ h\ b\ data. h \in \text{set } (\text{fst } (rcp\ gs\ bs\ ps\ sps\ data)) \Longrightarrow b \in \text{set } gs \cup \text{set } bs \Longrightarrow \text{fst } b \neq 0 \Longrightarrow$

$\neg \text{lt } (\text{fst } b) \text{ adds}_t \text{ lt } (\text{fst } h)$

assumes $\bigwedge gs\ bs\ ps\ sps\ d\ data. \text{dickson-grading } d \Longrightarrow$

$dgrad-p-set-le\ d\ (fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))\ (args-to-set\ (gs,\ bs,\ sps))$
assumes $\bigwedge gs\ bs\ ps\ sps\ data.\ component-of-term\ 'Keys\ (fst\ '(set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))) \subseteq$
 $component-of-term\ 'Keys\ (args-to-set\ (gs,\ bs,\ sps))$
assumes $\bigwedge gs\ bs\ ps\ sps\ data.\ is-Groebner-basis\ (fst\ 'set\ gs) \implies unique-idx\ (gs\ @\ bs)\ data \implies$
 $(fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))) \subseteq pmdl\ (args-to-set\ (gs,\ bs,\ sps))$
 \wedge
 $(\forall (p, q) \in set\ sps.\ set\ sps \subseteq set\ bs \times (set\ gs \cup set\ bs) \longrightarrow$
 $(red\ (fst\ '(set\ gs \cup set\ bs) \cup fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))))^{**}$
 $(spoly\ (fst\ p)\ (fst\ q))\ 0))$
shows $rcp-spec\ rcp$
 $\langle proof \rangle$

lemma $rcp-specD1$:

assumes $rcp-spec\ rcp$
shows $0 \notin fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data))$
 $\langle proof \rangle$

lemma $rcp-specD2$:

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

lemma $rcp-specD3$:

assumes $rcp-spec\ rcp$ **and** $dickson-grading\ d$
shows $dgrad-p-set-le\ d\ (fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))\ (args-to-set\ (gs,\ bs,\ sps))$
 $\langle proof \rangle$

lemma $rcp-specD4$:

assumes $rcp-spec\ rcp$
shows $component-of-term\ 'Keys\ (fst\ '(set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)))) \subseteq$
 $component-of-term\ 'Keys\ (args-to-set\ (gs,\ bs,\ sps))$
 $\langle proof \rangle$

lemma $rcp-specD5$:

assumes $rcp-spec\ rcp$ **and** $is-Groebner-basis\ (fst\ 'set\ gs)$ **and** $unique-idx\ (gs\ @\ bs)\ data$
shows $fst\ 'set\ (fst\ (rcp\ gs\ bs\ ps\ sps\ data)) \subseteq pmdl\ (args-to-set\ (gs,\ bs,\ sps))$
 $\langle proof \rangle$

lemma $rcp-specD6$:

assumes $rcp-spec\ rcp$ **and** $is-Groebner-basis\ (fst\ 'set\ gs)$ **and** $unique-idx\ (gs\ @\ bs)\ data$
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 ⇒₀ 'b) list ⇒ ('t, 'b, 'c) pdata-pair ⇒ ('t ⇒₀ 'b::field)*
where *trdsp bs p ≡ trd bs (spoly (fst (fst p)) (fst (snd p)))*

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

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

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

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

definition *gb-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 $\text{gb-red-aux } bs \ ps =$
 $(\text{let } bs' = \text{map } \text{fst } bs \text{ in}$
 $\text{filter } (\lambda h. h \neq 0) (\text{map } (\text{trdsp } bs') \ ps)$
)

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

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

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

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

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

lemma *gb-red-aux-irreducible*:
assumes $h \in \text{set } (\text{gb-red-aux } bs \ ps)$ **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 *gb-red-aux-dgrad-p-set-le*:
assumes *dickson-grading* d

shows $dgrad\text{-}p\text{-set}\text{-}le\ d\ (set\ (gb\text{-}red\text{-}aux\ bs\ ps))\ (args\text{-}to\text{-}set\ ([],\ bs,\ ps))$
 ⟨proof⟩

lemma *components-gb-red-aux-subset*:

$component\text{-}of\text{-}term\ 'Keys\ (set\ (gb\text{-}red\text{-}aux\ bs\ ps))\ \subseteq\ component\text{-}of\text{-}term\ 'Keys$
 $(args\text{-}to\text{-}set\ ([],\ bs,\ ps))$
 ⟨proof⟩

lemma *pmdl-gb-red-aux*: $set\ (gb\text{-}red\text{-}aux\ bs\ ps)\ \subseteq\ pmdl\ (args\text{-}to\text{-}set\ ([],\ bs,\ ps))$
 ⟨proof⟩

lemma *gb-red-aux-spoly-reducible*:

assumes $(p,\ q)\ \in\ set\ ps$
shows $(red\ (fst\ 'set\ bs\ \cup\ set\ (gb\text{-}red\text{-}aux\ bs\ ps)))^{**}\ (spoly\ (fst\ p)\ (fst\ q))\ 0$
 ⟨proof⟩

definition *gb-red* :: $(t,\ 'b::field,\ 'c::default,\ 'd)\ complT$

where $gb\text{-}red\ gs\ bs\ ps\ sps\ data = (map\ (\lambda h.\ (h,\ default))\ (gb\text{-}red\text{-}aux\ (gs\ @\ bs)$
 $sps),\ snd\ data)$

lemma *fst-set-fst-gb-red*: $fst\ 'set\ (fst\ (gb\text{-}red\ gs\ bs\ ps\ sps\ data)) = set\ (gb\text{-}red\text{-}aux$
 $(gs\ @\ bs)\ sps)$
 ⟨proof⟩

lemma *rcp-spec-gb-red*: $rcp\text{-}spec\ gb\text{-}red$
 ⟨proof⟩

lemmas $compl\text{-}struct\text{-}gb\text{-}red = compl\text{-}struct\text{-}rcp[OF\ rcp\text{-}spec\text{-}gb\text{-}red]$

lemmas $compl\text{-}pmdl\text{-}gb\text{-}red = compl\text{-}pmdl\text{-}rcp[OF\ rcp\text{-}spec\text{-}gb\text{-}red]$

lemmas $compl\text{-}conn\text{-}gb\text{-}red = compl\text{-}conn\text{-}rcp[OF\ rcp\text{-}spec\text{-}gb\text{-}red]$

7.2 Pair Selection

primrec *gb-sel* :: $(t,\ 'b::zero,\ 'c,\ 'd)\ selT$ **where**

$gb\text{-}sel\ gs\ bs\ []\ data = []$
 $gb\text{-}sel\ gs\ bs\ (p\ \#\ ps)\ data = [p]$

lemma *sel-spec-gb-sel*: $sel\text{-}spec\ gb\text{-}sel$
 ⟨proof⟩

7.3 Buchberger's Algorithm

lemma *struct-spec-gb*: $struct\text{-}spec\ gb\text{-}sel\ add\text{-}pairs\text{-}canon\ add\text{-}basis\text{-}canon\ gb\text{-}red$
 ⟨proof⟩

definition *gb-aux* :: $(t,\ 'b,\ 'c)\ pdata\ list\ \Rightarrow\ nat\ \times\ nat\ \times\ 'd\ \Rightarrow\ (t,\ 'b,\ 'c)\ pdata\ list$
 \Rightarrow

$(t,\ 'b,\ 'c)\ pdata\text{-}pair\ list\ \Rightarrow\ (t,\ 'b::field,\ 'c::default)\ pdata\ list$

where $gb\text{-}aux = gb\text{-}schema\text{-}aux\ gb\text{-}sel\ add\text{-}pairs\text{-}canon\ add\text{-}basis\text{-}canon\ gb\text{-}red$

lemmas *gb-aux-simps* [code] = *gb-schema-aux-simps*[OF *struct-spec-gb*, *folded gb-aux-def*]

definition *gb* :: ('t, 'b, 'c) *pdata*' list \Rightarrow 'd \Rightarrow ('t, 'b::field, 'c::default) *pdata*' list
where *gb* = *gb-schema-direct gb-sel add-pairs-canon add-basis-canon gb-red*

lemmas *gb-simps* [code] = *gb-schema-direct-def*[of *gb-sel add-pairs-canon add-basis-canon gb-red*, *folded gb-def gb-aux-def*]

lemmas *gb-isGB* = *gb-schema-direct-isGB*[OF *struct-spec-gb compl-conn-gb-red*, *folded gb-def*]

lemmas *gb-pmdl* = *gb-schema-direct-pmdl*[OF *struct-spec-gb compl-pmdl-gb-red*, *folded gb-def*]

7.3.1 Special Case: *punit*

lemma (in *gd-term*) *struct-spec-gb-punit*: *punit.struct-spec punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*
 ⟨*proof*⟩

definition *gb-aux-punit* :: ('a, 'b, 'c) *pdata* list \Rightarrow nat \times nat \times 'd \Rightarrow ('a, 'b, 'c) *pdata* list \Rightarrow

('a, 'b, 'c) *pdata-pair* list \Rightarrow ('a, 'b::field, 'c::default) *pdata* list
where *gb-aux-punit* = *punit.gb-schema-aux punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*

lemmas *gb-aux-punit-simps* [code] = *punit.gb-schema-aux-simps*[OF *struct-spec-gb-punit*, *folded gb-aux-punit-def*]

definition *gb-punit* :: ('a, 'b, 'c) *pdata*' list \Rightarrow 'd \Rightarrow ('a, 'b::field, 'c::default) *pdata*' list

where *gb-punit* = *punit.gb-schema-direct punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*

lemmas *gb-punit-simps* [code] = *punit.gb-schema-direct-def*[of *punit.gb-sel add-pairs-punit-canon punit.add-basis-canon punit.gb-red*, *folded gb-punit-def gb-aux-punit-def*]

lemmas *gb-punit-isGB* = *punit.gb-schema-direct-isGB*[OF *struct-spec-gb-punit punit.compl-conn-gb-red*, *folded gb-punit-def*]

lemmas *gb-punit-pmdl* = *punit.gb-schema-direct-pmdl*[OF *struct-spec-gb-punit punit.compl-pmdl-gb-red*, *folded gb-punit-def*]

end

end

8 Benchmark Problems for Computing Gröbner Bases

```

theory Benchmarks
  imports Polynomials.MPoly-Type-Class-OAlist
begin

```

This theory defines various well-known benchmark problems for computing Gröbner bases. The actual tests of the different algorithms on these problems are contained in the theories whose names end with *-Examples*.

8.1 Cyclic

```

definition cycl-pp :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  (nat, nat) pp
  where cycl-pp n d i = sparse0 (map ( $\lambda k$ . (modulo (k + i) n, 1)) [0..d])

```

```

definition cyclic :: (nat, nat) pp nat-term-order  $\Rightarrow$  nat  $\Rightarrow$  ((nat, nat) pp  $\Rightarrow_0$ 
'a::{zero,one,uminus}) list
  where cyclic to n =
    (let xs = [0..n] in
      (map ( $\lambda d$ . distr0 to (map ( $\lambda i$ . (cycl-pp n d i, 1)) xs)) [1..n]) @
      [distr0 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  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  ((nat,
nat) pp  $\Rightarrow_0$  'a::comm-ring-1)
  where katsura-poly to n i =
    change-ord to (( $\sum j::int=-int\ n..<n + 1$ . if abs (i - j)  $\leq n$  then V0
(nat (abs j)) * V0 (nat (abs (i - j))) else 0) - V0 i)

```

```

definition katsura :: (nat, nat) pp nat-term-order  $\Rightarrow$  nat  $\Rightarrow$  ((nat, nat) pp  $\Rightarrow_0$ 
'a::comm-ring-1) list
  where katsura to n =
    (let xs = [0..n] in
      (distr0 to ((sparse0 [(0, 1), 1] # (map ( $\lambda i$ . (sparse0 [(Suc i, 1)], 2)) xs)
@ [(0, -1)])) #
      (map (katsura-poly to n) xs)
    )

```

For $(1::'a) \leq n$, *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

(*SML*) *Rat.rat* |

constant *plus* :: *rat* \Rightarrow - \Rightarrow - \rightarrow

(*SML*) *Rat.add* |

constant *minus* :: *rat* \Rightarrow - \Rightarrow - \rightarrow

(*SML*) *Rat.add* ((-)) (*Rat.neg* ((-))) |

constant *times* :: *rat* \Rightarrow - \Rightarrow - \rightarrow

(*SML*) *Rat.mult* |

constant *inverse* :: *rat* \Rightarrow - \rightarrow

(*SML*) *Rat.inv* |

constant *divide* :: *rat* \Rightarrow - \Rightarrow - \rightarrow

(*SML*) *Rat.mult* ((-)) (*Rat.inv* ((-))) |

constant *rat-of-integer* :: *integer* \Rightarrow *rat* \rightarrow

(*SML*) *Rat.of'-int* |

constant *abs* :: *rat* \Rightarrow - \rightarrow

(*SML*) *Rat.abs* |

constant *0* :: *rat* \rightarrow

(*SML*) !(*Rat.make* (0, 1)) |

constant *1* :: *rat* \rightarrow

(*SML*) !(*Rat.make* (1, 1)) |

constant *uminus* :: *rat* \Rightarrow *rat* \rightarrow

(*SML*) *Rat.neg* |

```

constant HOL.equal :: rat  $\Rightarrow$  -  $\rightarrow$ 
  (SML) !((- : Rat.rat) = -) |
constant quotient-of'  $\rightarrow$ 
  (SML) Rat.dest

```

end

10 Sample Computations with Buchberger's Algorithm

```

theory Buchberger-Examples
  imports Buchberger Algorithm-Schema-Impl Code-Target-Rat
begin

```

```

lemma (in gd-term) compute-trd-aux [code]:
  trd-aux fs p r =
    (if is-zero p then
      r
    else
      case find-adds fs (lt p) of
        None  $\Rightarrow$  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)

```

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 ($X^2 * Z^3 + 3 * X^2 * Y$) = *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

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

lemma

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

lemma

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

lemma

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

lemma

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

lemma

$$\text{gb-punit DRLEX} \\ [\\ (X^3 - X * Y * Z^2, ()), \\ (Y^2 * Z - 1, ()) \\]$$

```

] () =
[
  (- (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*) ($Vec_0\ 1\ (X^2 * Z) + Vec_0\ 0\ (2 * Y \wedge 3 * Z^2)$) ($Vec_0\ 1\ (X^2 * Z^2 + 2 * Y \wedge 3 * Z^2)$)
 ⟨proof⟩

lemma

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

lemma

lt-pprod (*POT DRLEX*) ($Vec_0\ 1\ (X^2 * Z) + Vec_0\ 0\ (2 * Y \wedge 3 * Z^2)$) = (*sparse₀* [(0, 2), (2, 1)], 1)
 ⟨proof⟩

lemma

keys ($Vec_0\ 0\ (X^2 * Z \wedge 3) + Vec_0\ 1\ (2 * Y \wedge 3 * Z^2)$) =
 {(*sparse₀* [(0, 2), (2, 3)], 0), (*sparse₀* [(1, 3), (2, 2)], 1)}
 ⟨proof⟩

lemma

keys ($Vec_0\ 0\ (X^2 * Z \wedge 3) + Vec_0\ 2\ (2 * Y \wedge 3 * Z^2)$) =
 {(*sparse₀* [(0, 2), (2, 3)], 0), (*sparse₀* [(1, 3), (2, 2)], 2)}
 ⟨proof⟩

lemma

$Vec_0\ 1\ (X^2 * Z \wedge 7 + 2 * Y \wedge 3 * Z^2) + Vec_0\ 3\ (X^2 * Z \wedge 4) + Vec_0\ 1\ (-2 * Y \wedge 3 * Z^2) =$
 $Vec_0\ 1\ (X^2 * Z \wedge 7) + Vec_0\ 3\ (X^2 * Z \wedge 4)$
 ⟨proof⟩

lemma

lookup (*Vec*₀ 0 ($X^2 * Z^{\wedge} \gamma$) + *Vec*₀ 1 ($2 * Y^{\wedge} 3 * Z^2 + 2$)) (*sparse*₀ [(0, 2), (2, γ)], 0) = 1
 ⟨*proof*⟩

lemma

lookup (*Vec*₀ 0 ($X^2 * Z^{\wedge} \gamma$) + *Vec*₀ 1 ($2 * Y^{\wedge} 3 * Z^2 + 2$)) (*sparse*₀ [(0, 2), (2, γ)], 1) = 0
 ⟨*proof*⟩

lemma

*Vec*₀ 0 (0 * $X^{\wedge} 2 * Z^{\wedge} \gamma$) + *Vec*₀ 1 (0 * $Y^{\wedge} 3 * Z^2$) = 0
 ⟨*proof*⟩

lemma

monom-mult-pprod 3 (*sparse*₀ [(1, 2::nat)]) (*Vec*₀ 0 ($X^2 * Z$) + *Vec*₀ 1 ($2 * Y^{\wedge} 3 * Z^2$)) =
*Vec*₀ 0 ($3 * Y^2 * Z * X^2$) + *Vec*₀ 1 ($6 * Y^{\wedge} 5 * Z^2$)
 ⟨*proof*⟩

lemma

trd-pprod DRLEX [*Vec*₀ 0 ($Y^2 * Z + 2 * Y * Z^{\wedge} 3$)] (*Vec*₀ 0 ($X^2 * Z^{\wedge} 4 - 2 * Y^{\wedge} 3 * Z^{\wedge} 3$)) =
*Vec*₀ 0 ($X^2 * Z^{\wedge} 4 + Y^{\wedge} 4 * Z$)
 ⟨*proof*⟩

lemma

length (*gb-pprod* (*POT DRLEX*)
 [
 (*Vec*₀ 0 ($X^2 * Z^{\wedge} 4 - 2 * Y^{\wedge} 3 * Z^2$), ()),
 (*Vec*₀ 0 ($Y^2 * Z + 2 * Z^{\wedge} 3$), ())
] ()) = 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 *monomial 1 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* :: $(t, 'b::\text{zero})$ *poly-mapping set* \Rightarrow *t set* **where**

$\text{lt-set } F = \text{lt } \text{' } (F - \{0\})$

definition *lc-set* :: $(t, 'b::\text{zero})$ *poly-mapping set* \Rightarrow *'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 $\text{lc } f = c$
 $\langle \text{proof} \rangle$

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

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

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 $\text{lt } (\text{sum } p \ I) = \text{ord-term-lin.Max } (\text{lt-set } (p \ 'I))$
 $\langle \text{proof} \rangle$

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 $\text{lt } (\text{sum } p \ I) \in \text{lt-set } (p \ 'I)$
 $\langle \text{proof} \rangle$

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*: $\text{lookup } (\text{monic } p) \ v = (\text{lookup } p \ v) / \text{lc } p$
 $\langle \text{proof} \rangle$

lemma *lookup-monic-lt*:
assumes $p \neq 0$
shows $\text{lookup } (\text{monic } p) \ (\text{lt } p) = 1$
 $\langle \text{proof} \rangle$

lemma *monic-0 [simp]*: $\text{monic } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *monic-0-iff*: $(\text{monic } p = 0) \iff (p = 0)$
 $\langle \text{proof} \rangle$

lemma *keys-monic [simp]*: $\text{keys } (\text{monic } p) = \text{keys } p$

<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**
 $is-minimal-basis\ B \longleftrightarrow (0 \notin B \wedge (\forall p\ q. p \in B \longrightarrow q \in B \longrightarrow p \neq q \longrightarrow \neg lt\ p\ q))$
 $adds_t\ lt\ q))$

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 $\bigwedge p. p \in B \implies p \neq 0$ **and** $\bigwedge p\ q. p \in B \implies q \in B \implies p \neq q \implies \neg$
 $lt\ p\ q\ adds_t\ lt\ q$
shows *is-minimal-basis* B
 ⟨proof⟩

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

lemma *is-minimal-basisD2*:
assumes *is-minimal-basis* B **and** $p \in B$ **and** $q \in B$ **and** $p \neq q$
shows $\neg lt\ p\ adds_t\ lt\ q$
 ⟨proof⟩

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

lemma *is-minimal-basis-subset*:

assumes *is-minimal-basis* B **and** $A \subseteq B$

shows *is-minimal-basis* A

<proof>

lemma *nadds-red*:

assumes *nadds*: $\bigwedge q. q \in B \implies \neg \text{lt } q \text{ adds}_t \text{ lt } p$ **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) []

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 $p \in \text{set } xs$ and $p \neq 0$
obtains q where $q \in \text{set } (\text{comp-red-basis } xs)$ and $lt \ q \ \text{adds}_t \ lt \ p$
(proof)

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

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

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

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

12.5 Auto-Reduction and Monicity

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

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

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

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

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

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

<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 ($'t \Rightarrow_0 'b::\text{field}$) *set*
where *reduced-GB* $B = (\text{THE } G. \text{finite } G \wedge \text{is-reduced-GB } G \wedge \text{pmdl } G = \text{pmdl } B)$

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 *pmdl (reduced-GB F) = pmdl F*
<proof>

lemma *reduced-GB-unique-dgrad-p-set:*

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

lemma *reduced-GB-dgrad-p-set:*

assumes *dickson-grading d and finite (component-of-term ‘ Keys F) and $F \subseteq$*
dgrad-p-set d m
shows *reduced-GB F \subseteq dgrad-p-set d m*
<proof>

lemma *reduced-GB-unique:*

assumes *finite G and is-reduced-GB G and pmdl G = pmdl F*
shows *reduced-GB F = G*
<proof>

lemma *is-reduced-GB-empty: is-reduced-GB {}*

<proof>

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

<proof>

lemma *reduced-GB-empty: reduced-GB {} = {}*

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *reduced-GB-is-reduced-GB-finite: finite F \implies is-reduced-GB (reduced-GB*
F)

<proof>

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

<proof>

lemma *reduced-GB-is-auto-reduced-finite*: $\text{finite } F \implies \text{is-auto-reduced } (\text{reduced-GB } F)$

<proof>

lemma *reduced-GB-is-monic-set-finite*: $\text{finite } F \implies \text{is-monic-set } (\text{reduced-GB } F)$

<proof>

lemma *reduced-GB-nonzero-finite*: $\text{finite } F \implies 0 \notin \text{reduced-GB } F$

<proof>

lemma *reduced-GB-pmdl-finite*: $\text{finite } F \implies \text{pmdl } (\text{reduced-GB } F) = \text{pmdl } F$

<proof>

lemma *reduced-GB-unique-finite*: $\text{finite } F \implies \text{is-reduced-GB } G \implies \text{pmdl } G = \text{pmdl } F \implies \text{reduced-GB } F = G$

<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** $F \subseteq \text{punit.dgrad-p-set } d$

shows $\text{ideal } F = \text{UNIV} \iff \text{punit.reduced-GB } F = \{1\}$

<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 character-

istic 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** $_v^*$ 70)
where $v\ _v^*\ A \equiv vec\ (dim\text{-col } A)\ (\lambda j. v \cdot col\ A\ j)$

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*:
 $v\ _v^*\ A = (resize\text{-vec } (dim\text{-row } A)\ v)\ _v^*\ A$
<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 $\text{row-space } A = (\lambda v. \text{mult-vec-mat } v A) \text{ ` } (\text{carrier-vec } (\dim\text{-row } A))$

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 *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 *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 *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 *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 *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 $\text{keys } p \subseteq \text{set } ts$

shows *row-to-poly* *ts* (*poly-to-row* *ts* *p*) = (*p*::*t* \Rightarrow_0 *b*::*semiring-1*)
<proof>

lemma *lookup-list-to-poly*: *lookup* (*list-to-poly* *ts* *cs*) = *list-to-fun* *ts* *cs*
<proof>

lemma *list-to-fun-Nil* [*simp*]: *list-to-fun* [] *cs* = 0
<proof>

lemma *list-to-poly-Nil* [*simp*]: *list-to-poly* [] *cs* = 0
<proof>

lemma *row-to-poly-Nil* [*simp*]: *row-to-poly* [] *r* = 0
<proof>

lemma *lookup-row-to-poly*:
 assumes *distinct* *ts* **and** *dim-vec* *r* = *length* *ts* **and** *i* < *length* *ts*
 shows *lookup* (*row-to-poly* *ts* *r*) (*ts* ! *i*) = *r* \$ *i*
<proof>

lemma *keys-row-to-poly*: *keys* (*row-to-poly* *ts* *r*) \subseteq *set* *ts*
<proof>

lemma *lookup-row-to-poly-not-zeroE*:
 assumes *lookup* (*row-to-poly* *ts* *r*) *t* \neq 0
 obtains *i* **where** *i* < *length* *ts* **and** *t* = *ts* ! *i*
<proof>

lemma *row-to-poly-zero* [*simp*]: *row-to-poly* *ts* (*0_v* (*length* *ts*)) = (*0*::*t* \Rightarrow_0 *b*::*zero*)
<proof>

lemma *row-to-poly-zeroD*:
 assumes *distinct* *ts* **and** *dim-vec* *r* = *length* *ts* **and** *row-to-poly* *ts* *r* = 0
 shows *r* = *0_v* (*length* *ts*)
<proof>

lemma *row-to-poly-inj*:
 assumes *distinct* *ts* **and** *dim-vec* *r1* = *length* *ts* **and** *dim-vec* *r2* = *length* *ts*
 and *row-to-poly* *ts* *r1* = *row-to-poly* *ts* *r2*
 shows *r1* = *r2*
<proof>

lemma *row-to-poly-vec-plus*:
 assumes *distinct* *ts* **and** *length* *ts* = *n*
 shows *row-to-poly* *ts* (*vec* *n* (*f1* + *f2*)) = *row-to-poly* *ts* (*vec* *n* *f1*) + *row-to-poly*
ts (*vec* *n* *f2*)
<proof>

lemma *row-to-poly-vec-sum*:

assumes *distinct ts and length ts = n*
shows $\text{row-to-poly } ts \ (\text{vec } n \ (\lambda j. \sum_{i \in I}. f \ i \ j)) = ((\sum_{i \in I}. \text{row-to-poly } ts \ (\text{vec } n \ (f \ i)))::'t \Rightarrow_0 \ 'b::\text{comm-monoid-add})$
 $\langle \text{proof} \rangle$

lemma *row-to-poly-smult*:
assumes *distinct ts and dim-vec r = length ts*
shows $\text{row-to-poly } ts \ (c \cdot_v \ r) = c \cdot (\text{row-to-poly } ts \ r)$
 $\langle \text{proof} \rangle$

lemma *poly-to-row-Nil [simp]*: $\text{poly-to-row } [] \ p = \text{vec } 0 \ f$
 $\langle \text{proof} \rangle$

lemma *polys-to-mat-Nil [simp]*: $\text{polys-to-mat } ts \ [] = \text{mat } 0 \ (\text{length } ts) \ f$
 $\langle \text{proof} \rangle$

lemma *dim-row-polys-to-mat[simp]*: $\text{dim-row } (\text{polys-to-mat } ts \ ps) = \text{length } ps$
 $\langle \text{proof} \rangle$

lemma *dim-col-polys-to-mat[simp]*: $\text{dim-col } (\text{polys-to-mat } ts \ ps) = \text{length } ts$
 $\langle \text{proof} \rangle$

lemma *polys-to-mat-index*:
assumes $i < \text{length } ps$ **and** $j < \text{length } ts$
shows $(\text{polys-to-mat } ts \ ps) \ \$\$ \ (i, j) = \text{lookup } (ps \ ! \ i) \ (ts \ ! \ j)$
 $\langle \text{proof} \rangle$

lemma *row-polys-to-mat*:
assumes $i < \text{length } ps$
shows $\text{row } (\text{polys-to-mat } ts \ ps) \ i = \text{poly-to-row } ts \ (ps \ ! \ i)$
 $\langle \text{proof} \rangle$

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

lemma *length-mat-to-polys[simp]*: $\text{length } (\text{mat-to-polys } ts \ A) = \text{dim-row } A$
 $\langle \text{proof} \rangle$

lemma *mat-to-polys-nth*:
assumes $i < \text{dim-row } A$
shows $(\text{mat-to-polys } ts \ A) \ ! \ i = \text{row-to-poly } ts \ (\text{row } A \ i)$
 $\langle \text{proof} \rangle$

lemma *Keys-mat-to-polys*: $\text{Keys } (\text{set } (\text{mat-to-polys } ts \ A)) \subseteq \text{set } ts$
 $\langle \text{proof} \rangle$

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 $phull (set (mat-to-polys ts (row-echelon (polys-to-mat ts ps)))) = phull (set ps)$
 ⟨proof⟩

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** $?l = ?r$)
 ⟨proof⟩

end

context *ordered-term*

begin

lemma *lt-row-to-poly-pivot-fun*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** *pivot-fun A f (dim-col A)*
and $i < dim-row A$ **and** $f i < dim-col A$
shows $lt ((mat-to-polys (pps-to-list S) A) ! i) = (pps-to-list S) ! (f i)$
 ⟨proof⟩

lemma *lc-row-to-poly-pivot-fun*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** *pivot-fun A f (dim-col A)*
and $i < dim-row A$ **and** $f i < dim-col A$
shows $lc ((mat-to-polys (pps-to-list S) A) ! i) = 1$
 ⟨proof⟩

lemma *lt-row-to-poly-pivot-fun-less*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** *pivot-fun A f (dim-col A)*
and $i1 < i2$ **and** $i2 < dim-row A$ **and** $f i1 < dim-col A$ **and** $f i2 < dim-col A$
shows $(pps-to-list S) ! (f i2) \prec_t (pps-to-list S) ! (f i1)$
 ⟨proof⟩

lemma *lt-row-to-poly-pivot-fun-eqD*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** *pivot-fun A f (dim-col A)*
and $i1 < dim-row A$ **and** $i2 < dim-row A$ **and** $f i1 < dim-col A$ **and** $f i2 < dim-col A$
and $(pps-to-list S) ! (f i1) = (pps-to-list S) ! (f i2)$
shows $i1 = i2$
 ⟨proof⟩

lemma *lt-row-to-poly-pivot-in-keysD*:

assumes $card S = dim-col (A::'b::semiring-1 mat)$ **and** *pivot-fun A f (dim-col A)*

and $i1 < \dim\text{-row } A$ **and** $i2 < \dim\text{-row } A$ **and** $f\ i1 < \dim\text{-col } A$
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 $\text{card } S = \dim\text{-col } (A :: 'b :: \{\text{comm-semiring-0, semiring-1-no-zero-divisors}\})$
 mat

and $\text{pivot-fun } A\ f\ (\dim\text{-col } A)$ **and** $p \in \text{row-to-poly } (\text{pps-to-list } S) \text{ 'row-space } A$
and $p \neq 0$

shows $\text{lt } p \in \text{lt-set } (\text{set } (\text{mat-to-polys } (\text{pps-to-list } S) A))$
 $\langle \text{proof} \rangle$

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

definition *Macaulay-mat* :: $('t \Rightarrow_0 'b) \text{ list} \Rightarrow 'b :: \text{field } \text{mat}$

where $\text{Macaulay-mat } ps = \text{polys-to-mat } (\text{Keys-to-list } ps) ps$

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

where $\text{Macaulay-list } ps =$
 $\text{filter } (\lambda p. p \neq 0) (\text{mat-to-polys } (\text{Keys-to-list } ps) (\text{row-echelon } (\text{Macaulay-mat } ps)))$

lemma *dim-Macaulay-mat[simp]*:

$\dim\text{-row } (\text{Macaulay-mat } ps) = \text{length } ps$

$\dim\text{-col } (\text{Macaulay-mat } ps) = \text{card } (\text{Keys } (\text{set } ps))$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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 $\text{pivot-fun } (\text{row-echelon } (\text{Macaulay-mat } ps))\ f\ (\dim\text{-col } (\text{row-echelon } (\text{Macaulay-mat } ps)))$

obtains i **where** $i < \dim\text{-row } (\text{row-echelon } (\text{Macaulay-mat } ps))$

and $p = (\text{mat-to-polys } (\text{Keys-to-list } ps) (\text{row-echelon } (\text{Macaulay-mat } ps))) ! i$

and $f\ i < \dim\text{-col } (\text{row-echelon } (\text{Macaulay-mat } ps))$

$\langle \text{proof} \rangle$

lemma *phull-Macaulay-list*: $\text{phull} (\text{set} (\text{Macaulay-list } ps)) = \text{phull} (\text{set } ps)$
 <proof>

lemma *pmdl-Macaulay-list*: $\text{pmdl} (\text{set} (\text{Macaulay-list } ps)) = \text{pmdl} (\text{set } ps)$
 <proof>

lemma *Macaulay-list-is-monic-set*: $\text{is-monic-set} (\text{set} (\text{Macaulay-list } ps))$
 <proof>

lemma *Macaulay-list-not-zero*: $0 \notin \text{set} (\text{Macaulay-list } ps)$
 <proof>

lemma *Macaulay-list-distinct-lt*:
 assumes $x \in \text{set} (\text{Macaulay-list } ps)$ and $y \in \text{set} (\text{Macaulay-list } ps)$
 and $x \neq y$
 shows $\text{lt } x \neq \text{lt } y$
 <proof>

lemma *Macaulay-list-lt*:
 assumes $p \in \text{phull} (\text{set } ps)$ and $p \neq 0$
 obtains g where $g \in \text{set} (\text{Macaulay-list } ps)$ and $g \neq 0$ and $\text{lt } p = \text{lt } g$
 <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* :: $('a \Rightarrow \text{nat}) \Rightarrow (((('t \Rightarrow_0 'b) \text{ list} \times 't \text{ list} \times 't \text{ list} \times ('t \Rightarrow_0 'b) \text{ list}) \times ((('t \Rightarrow_0 'b) \text{ list} \times 't \text{ list} \times 't \text{ list} \times ('t \Rightarrow_0 'b) \text{ list})) \text{ set}$

where $\text{sym-preproc-aux-term1 } d =$
 $\{(gs1, ks1, ts1, fs1), (gs2::('t \Rightarrow_0 'b) \text{ list}, ks2, ts2, fs2)\}. \exists t2 \in \text{set } ts2.$
 $\forall t1 \in \text{set } ts1. t1 \prec_t t2\}$

definition *sym-preproc-aux-term2* :: ('a \Rightarrow nat) \Rightarrow (((('t \Rightarrow_0 'b)::zero) 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-term2* d =
 {((gs1, ks1, ts1, fs1), (gs2::('t \Rightarrow_0 'b) list, ks2, ts2, fs2)). gs1 = gs2 \wedge dgrad-set-le d (pp-of-term ' set ts1) (pp-of-term ' (Keys (set gs2) \cup set ts2))}

definition *sym-preproc-aux-term*

where *sym-preproc-aux-term* d = *sym-preproc-aux-term1* d \cap *sym-preproc-aux-term2* d

lemma *wfp-on-ord-term-strict*:

assumes *dickson-grading* d

shows *wfp-on* (\prec_t) (pp-of-term - ' dgrad-set d m)

<proof>

lemma *sym-preproc-aux-term1-wf-on*:

assumes *dickson-grading* d

shows *wfp-on* ($\lambda x y. (x, y) \in \text{sym-preproc-aux-term1 } d$) {x. set (fst (snd (snd x))) \subseteq 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 \Rightarrow_0 'b)::semiring-1) list \Rightarrow 't list \Rightarrow ('t \Rightarrow_0 'b) list \Rightarrow 't \Rightarrow

('t list \times ('t \Rightarrow_0 'b) list) **where**
sym-preproc-addnew [] vs fs - = (vs, fs)|
sym-preproc-addnew (g # gs) vs fs v =
 (if lt g adds_t v then
 (let f = monom-mult 1 (pp-of-term v - lp g) g in
 sym-preproc-addnew gs (merge-wrt (\succ_t) vs (keys-to-list (tail f))) (insert-list f fs) v
)
 else
 sym-preproc-addnew gs vs fs v
)

lemma *fst-sym-preproc-addnew-less*:

assumes $\bigwedge u. u \in \text{set } vs \implies u \prec_t v$

and $u \in \text{set } (\text{fst } (\text{sym-preproc-addnew } gs \text{ vs } fs \ v))$

shows $u \prec_t v$

<proof>

lemma *fst-sym-preproc-addnew-dgrad-set-le*:

assumes *dickson-grading* d

shows $dgrad\text{-}set\text{-}le\ d\ (pp\text{-}of\text{-}term\ 'set\ (fst\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v)))$
 $(pp\text{-}of\text{-}term\ '(Keys\ (set\ gs)\ \cup\ insert\ v\ (set\ vs)))$
 $\langle proof \rangle$

lemma *components-fst-sym-preproc-addnew-subset*:

component-of-term $'set\ (fst\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v)) \subseteq component\text{-}of\text{-}term$
 $'(Keys\ (set\ gs)\ \cup\ insert\ v\ (set\ vs))$
 $\langle proof \rangle$

lemma *fst-sym-preproc-addnew-superset*: $set\ vs \subseteq set\ (fst\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$
 $\langle proof \rangle$

lemma *snd-sym-preproc-addnew-superset*: $set\ fs \subseteq set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$
 $\langle proof \rangle$

lemma *in-snd-sym-preproc-addnewE*:

assumes $p \in set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$

assumes 1: $p \in set\ fs \implies thesis$

assumes 2: $\bigwedge g\ s.\ g \in set\ gs \implies p = monom\text{-}mult\ 1\ s\ g \implies thesis$

shows *thesis*

$\langle proof \rangle$

lemma *sym-preproc-addnew-pmdl*:

$pmdl\ (set\ gs\ \cup\ set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))) = pmdl\ (set\ gs\ \cup\ set\ fs)$

(**is** $pmdl\ (set\ gs\ \cup\ ?l) = ?r$)

$\langle proof \rangle$

lemma *Keys-snd-sym-preproc-addnew*:

$Keys\ (set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))) \cup\ insert\ v\ (set\ vs) =$

$Keys\ (set\ fs) \cup\ insert\ v\ (set\ (fst\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ (fs::('t \Rightarrow_0 'b)::semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors)$
 $list)\ v)))$

$\langle proof \rangle$

lemma *sym-preproc-addnew-complete*:

assumes $g \in set\ gs$ **and** $lt\ g\ adds_t\ v$

shows $monom\text{-}mult\ 1\ (pp\text{-}of\text{-}term\ v - lp\ g)\ g \in set\ (snd\ (sym\text{-}preproc\text{-}addnew\ gs\ vs\ fs\ v))$

$\langle proof \rangle$

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

$('t\ list \times ('t \Rightarrow_0 'b)\ list)$ **where**

$sym\text{-}preproc\text{-}aux\ gs\ ks\ (vs,\ fs) =$

(*if* $vs = []$ *then*

(ks, fs)
else
 let $v = \text{ord-term-lin.max-list } vs; vs' = \text{removeAll } v \text{ in}$
 $\text{sym-preproc-aux } gs \ (ks \ @ \ [v]) \ (\text{sym-preproc-addnew } gs \ vs' \ fs \ v)$
 $)$
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$

lemma *sym-preproc-aux-Nil*: $\text{sym-preproc-aux } gs \ ks \ ([], fs) = (ks, fs)$
 $\langle \text{proof} \rangle$

lemma *sym-preproc-aux-sorted*:
assumes *sorted-wrt* $(\succ_t) \ (v \# vs)$
shows $\text{sym-preproc-aux } gs \ ks \ (v \# vs, fs) = \text{sym-preproc-aux } gs \ (ks \ @ \ [v])$
 $(\text{sym-preproc-addnew } gs \ vs \ fs \ v)$
 $\langle \text{proof} \rangle$

lemma *sym-preproc-aux-induct* [*consumes 0, case-names base rec*]:
assumes *base*: $\bigwedge ks \ fs. P \ ks \ [] \ fs \ (ks, fs)$
and *rec*: $\bigwedge ks \ vs \ fs \ v \ vs'. vs \neq [] \implies v = \text{ord-term-lin.Max} \ (\text{set } vs) \implies vs' =$
 $\text{removeAll } v \ vs \implies$
 $P \ (ks \ @ \ [v]) \ (\text{fst} \ (\text{sym-preproc-addnew } gs \ vs' \ fs \ v)) \ (\text{snd}$
 $(\text{sym-preproc-addnew } gs \ vs' \ fs \ v))$
 $(\text{sym-preproc-aux } gs \ (ks \ @ \ [v]) \ (\text{sym-preproc-addnew } gs \ vs' \ fs \ v))$
 \implies
 $P \ ks \ vs \ fs \ (\text{sym-preproc-aux } gs \ (ks \ @ \ [v]) \ (\text{sym-preproc-addnew } gs \ vs'$
 $fs \ v))$
shows $P \ ks \ vs \ fs \ (\text{sym-preproc-aux } gs \ ks \ (vs, fs))$
 $\langle \text{proof} \rangle$

lemma *fst-sym-preproc-aux-sorted-wrt*:
assumes *sorted-wrt* $(\succ_t) \ ks$ **and** $\bigwedge k \ v. k \in \text{set } ks \implies v \in \text{set } vs \implies v \prec_t k$
shows *sorted-wrt* $(\succ_t) \ (\text{fst} \ (\text{sym-preproc-aux } gs \ ks \ (vs, fs)))$
 $\langle \text{proof} \rangle$

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

lemma *snd-sym-preproc-aux-superset*: $\text{set } fs \subseteq \text{set} \ (\text{snd} \ (\text{sym-preproc-aux } gs \ ks \ (vs,$
 $fs)))$
 $\langle \text{proof} \rangle$

lemma *in-snd-sym-preproc-auxE*:
assumes $p \in \text{set} \ (\text{snd} \ (\text{sym-preproc-aux } gs \ ks \ (vs, fs)))$
assumes *1*: $p \in \text{set } fs \implies \text{thesis}$

assumes $\exists g t. g \in \text{set } gs \implies p = \text{monom-mult } 1 t g \implies \text{thesis}$
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 $\text{dgrad-set-le } d (\text{pp-of-term } ' \text{Keys } (\text{set } (\text{snd } (\text{sym-preproc-aux } gs \text{ ks } (vs, fs)))))) (\text{pp-of-term } ' \text{Keys } (\text{set } gs \cup \text{set } fs))$
 ⟨*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 $\text{sym-preproc } gs \text{ fs} = \text{sym-preproc-aux } gs \ \square \ (\text{Keys-to-list } fs, fs)$

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

⟨*proof*⟩

lemma *fst-sym-preproc*:

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

lemma *snd-sym-preproc-superset*: $\text{set } fs \subseteq \text{set } (\text{snd } (\text{sym-preproc } gs \text{ fs}))$

⟨*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** *red-single* $p\ q\ f\ 0$
shows *lin-red* $F\ p\ q$
 \langle *proof* \rangle

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

lemma *lin-red-imp-red*:
assumes *lin-red* $F\ p\ q$
shows *red* $F\ p\ q$
 \langle *proof* \rangle

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

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

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

16.3 Reduction

definition *Macaulay-red* $:: 't\ list \Rightarrow ('t \Rightarrow_0 'b)\ list \Rightarrow ('t \Rightarrow_0 'b::field)\ list$
where *Macaulay-red vs fs* =
 $(let\ lts = map\ lt\ (filter\ (\lambda p.\ p \neq 0)\ fs)\ in$
 $filter\ (\lambda p.\ p \neq 0 \wedge lt\ p \notin set\ lts)\ (mat-to-polys\ vs\ (row-echelon\ (polys-to-mat$
 $vs\ fs)))$
 $)$

Macaulay-red vs fs auto-reduces (w. r. t. *lin-red*) the given list *fs* and returns those non-zero polynomials whose leading terms are not in *lt-set* (*set fs*). Argument *vs* is expected to be *Keys-to-list fs*; this list is passed as an argument to *Macaulay-red*, because it can be efficiently computed by symbolic preprocessing.

lemma *Macaulay-red-alt*:
 $Macaulay-red\ (Keys-to-list\ fs)\ fs = filter\ (\lambda p.\ lt\ p \notin lt-set\ (set\ fs))\ (Macaulay-list\ fs)$
 \langle *proof* \rangle

lemma *set-Macaulay-red*:
 $set\ (Macaulay-red\ (Keys-to-list\ fs)\ fs) = set\ (Macaulay-list\ fs) - \{p.\ lt\ p \in lt-set\ (set\ fs)\}$

<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 (set (f₄-red-aux bs ps)) \subseteq *component-of-term 'Keys*
(args-to-set ([], bs, ps))
 ⟨proof⟩

lemma *pmdl- f_4 -red-aux*: $\text{set } (f_4\text{-red-aux } bs \ ps) \subseteq \text{pmdl } (\text{args-to-set } ([], \ bs, \ ps))$
 ⟨proof⟩

lemma f_4 -red-aux-phull-reducible:
assumes $\text{set } ps \subseteq \text{set } bs \times \text{set } bs$
and $f \in \text{phull } (\text{set } (\text{pdata-pairs-to-list } ps))$
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::field, 'c::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 *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 *rcp-spec- f_4 -red*: *rcp-spec* f_4 -red
 ⟨proof⟩

lemmas *compl-struct- f_4 -red* = *compl-struct-rcp*[*OF rcp-spec- f_4 -red*]

lemmas *compl-pmdl- f_4 -red* = *compl-pmdl-rcp*[*OF rcp-spec- f_4 -red*]

lemmas *compl-conn- f_4 -red* = *compl-conn-rcp*[*OF 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 ($\lambda x. \text{fst } x \notin \text{set } vs$) xs) k
 <proof>

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

lemma (in *term-powerprod*) *compute-list-to-poly* [code]:
 list-to-poly ts cs = distr₀ DRLEX (remdups-wrt-rev fst (zip ts cs)) []
 <proof>

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

declare *conversep-iff* [code]

derive (eq) *ceq poly-mapping*
derive (no) *ccompare poly-mapping*
derive (dlist) *set-impl poly-mapping*
derive (no) *cenum poly-mapping*

derive (eq) *ceq rat*
derive (no) *ccompare rat*
derive (dlist) *set-impl rat*
derive (no) *cenum rat*

global-interpretation *punit'*: *gd-powerprod ord-pp-punit cmp-term ord-pp-strict-punit*
cmp-term

rewrites *punit.adds-term* = (adds)
and *punit.pp-of-term* = ($\lambda x. x$)
and *punit.component-of-term* = ($\lambda. ()$)
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
and punit'.punit.keys-to-list = keys-to-list-punit cmp-term
for cmp-term :: ('a::nat, 'b::{nat,add-wellorder}) pp nat-term-order

defines max-punit = punit'.ordered-powerprod-lin.max
and max-list-punit = punit'.ordered-powerprod-lin.max-list
and find-adds-punit = punit'.punit.find-adds
and trd-aux-punit = punit'.punit.trd-aux
and trd-punit = punit'.punit.trd
and spoly-punit = punit'.punit.spoly
and count-const-lt-components-punit = punit'.punit.count-const-lt-components
and count-rem-components-punit = punit'.punit.count-rem-components
and const-lt-component-punit = punit'.punit.const-lt-component
and full-gb-punit = punit'.punit.full-gb
and add-pairs-single-sorted-punit = punit'.punit.add-pairs-single-sorted
and add-pairs-punit = punit'.punit.add-pairs
and canon-pair-order-aux-punit = punit'.punit.canon-pair-order-aux
and canon-basis-order-punit = punit'.punit.canon-basis-order
and new-pairs-sorted-punit = punit'.punit.new-pairs-sorted
and product-crit-punit = punit'.punit.product-crit
and chain-ncrit-punit = punit'.punit.chain-ncrit
and chain-ocrit-punit = punit'.punit.chain-ocrit
and apply-icrit-punit = punit'.punit.apply-icrit
and apply-ncrit-punit = punit'.punit.apply-ncrit
and apply-ocrit-punit = punit'.punit.apply-ocrit
and Keys-to-list-punit = punit'.punit.Keys-to-list
and sym-preproc-addnew-punit = punit'.punit.sym-preproc-addnew
and sym-preproc-aux-punit = punit'.punit.sym-preproc-aux
and sym-preproc-punit = punit'.punit.sym-preproc
and Macaulay-mat-punit = punit'.punit.Macaulay-mat
and Macaulay-list-punit = punit'.punit.Macaulay-list
and pdata-pairs-to-list-punit = punit'.punit.pdata-pairs-to-list
and Macaulay-red-punit = punit'.punit.Macaulay-red
and f4-sel-aux-punit = punit'.punit.f4-sel-aux
and f4-sel-punit = punit'.punit.f4-sel
and f4-red-aux-punit = punit'.punit.f4-red-aux
and f4-red-punit = punit'.punit.f4-red
and f4-aux-punit = punit'.punit.f4-aux-punit
and f4-punit = punit'.punit.f4-punit
<proof>

```

17.2 Computations

```

experiment begin interpretation 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

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* \langle *proof* \rangle

18.1 Syzygy Modules Generated by Sets

context *module*

begin

definition *rep* :: (*'b* \Rightarrow_0 *'a*) \Rightarrow *'b*

where *rep* *r* = $(\sum_{v \in \text{keys } r} \text{lookup } r \ v \ *s \ v)$

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* = $\{s. \text{local.represents } B \ s \ 0\}$

end

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

context *module*

begin

lemma *rep-monomial* [*simp*]: *rep* (*monomial* *c* *x*) = *c* * *s* *x*

\langle *proof* \rangle

lemma *rep-zero* [*simp*]: *rep* 0 = 0

\langle *proof* \rangle

lemma *rep-uminus* [*simp*]: *rep* (- *r*) = - *rep* *r*

\langle *proof* \rangle

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 *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**
 $u = t \oplus v$
<proof>

lemma *rep-mult-scalar*: $\text{pmdl.rep } (\text{punit.monom-mult } c \ 0 \ r) = c \odot \text{pmdl.rep } r$
<proof>

lemma *represents-mult-scalar*:

assumes $\text{pmdl.represents } B \ r \ x$
shows $\text{pmdl.represents } B \ (\text{punit.monom-mult } c \ 0 \ r) \ (c \odot x)$
<proof>

lemma *syzygy-module-closed-map-scale*: $s \in \text{pmdl.syzygy-module } B \implies c \cdot s \in \text{pmdl.syzygy-module } B$
<proof>

lemma *phull-syzygy-module*: $\text{phull } (\text{pmdl.syzygy-module } B) = \text{pmdl.syzygy-module } B$
<proof>

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 $x \in \text{set } xs$
<proof>

lemma *lookup-pm-of-idx-pm-distinct*:

assumes *distinct* xs **and** $i < \text{length } xs$
shows $\text{lookup } (\text{pm-of-idx-pm } xs \ f) \ (xs ! i) = \text{lookup } f \ i$
<proof>

lemma *keys-pm-of-idx-pm-subset*: $\text{keys } (\text{pm-of-idx-pm } xs \ f) \subseteq \text{set } xs$
(proof)

lemma *range-pm-of-idx-pm-subset*: $\text{Poly-Mapping.range } (\text{pm-of-idx-pm } xs \ f) \subseteq$
 $\text{lookup } f \ \{0..<\text{length } xs\} - \{0\}$
(proof)

corollary *range-pm-of-idx-pm-subset'*: $\text{Poly-Mapping.range } (\text{pm-of-idx-pm } xs \ f) \subseteq$
 $\text{Poly-Mapping.range } f$
(proof)

lemma *pm-of-idx-pm-zero [simp]*: $\text{pm-of-idx-pm } xs \ 0 = 0$
(proof)

lemma *pm-of-idx-pm-plus*: $\text{pm-of-idx-pm } xs \ (f + g) = \text{pm-of-idx-pm } xs \ f + \text{pm-of-idx-pm}$
 $xs \ g$
(proof)

lemma *pm-of-idx-pm-uminus*: $\text{pm-of-idx-pm } xs \ (- f) = - \text{pm-of-idx-pm } xs \ f$
(proof)

lemma *pm-of-idx-pm-minus*: $\text{pm-of-idx-pm } xs \ (f - g) = \text{pm-of-idx-pm } xs \ f -$
 $\text{pm-of-idx-pm } xs \ g$
(proof)

lemma *pm-of-idx-pm-monom-mult*: $\text{pm-of-idx-pm } xs \ (\text{punit.monom-mult } c \ 0 \ f) =$
 $\text{punit.monom-mult } c \ 0 \ (\text{pm-of-idx-pm } xs \ f)$
(proof)

lemma *pm-of-idx-pm-monomial*:
assumes *distinct* xs
shows $\text{pm-of-idx-pm } xs \ (\text{monomial } c \ i) = (\text{monomial } c \ (xs \ ! \ i) \text{ when } i < \text{length}$
 $xs)$
(proof)

lemma *pm-of-idx-pm-take*:
assumes $\text{keys } f \subseteq \{0..<j\}$
shows $\text{pm-of-idx-pm } (\text{take } j \ xs) \ f = \text{pm-of-idx-pm } xs \ f$
(proof)

lemma *lookup-idx-pm-of-pm*: $\text{lookup } (\text{idx-pm-of-pm } xs \ f) = (\lambda i. \text{lookup } f \ (xs \ ! \ i)$
 $\text{when } i < \text{length } xs)$
(proof)

lemma *keys-idx-pm-of-pm-subset*: $\text{keys } (\text{idx-pm-of-pm } xs \ f) \subseteq \{0..<\text{length } xs\}$
(proof)

lemma *idx-pm-of-pm-zero [simp]*: $\text{idx-pm-of-pm } xs \ 0 = 0$
(proof)

lemma *idx-pm-of-pm-plus*: $\text{idx-pm-of-pm } xs (f + g) = \text{idx-pm-of-pm } xs f + \text{idx-pm-of-pm } xs g$
 ⟨proof⟩

lemma *idx-pm-of-pm-minus*: $\text{idx-pm-of-pm } xs (f - g) = \text{idx-pm-of-pm } xs f - \text{idx-pm-of-pm } xs g$
 ⟨proof⟩

lemma *pm-of-idx-pm-of-pm*:
 assumes $\text{keys } f \subseteq \text{set } xs$
 shows $\text{pm-of-idx-pm } xs (\text{idx-pm-of-pm } xs f) = f$
 ⟨proof⟩

lemma *idx-pm-of-pm-of-idx-pm*:
 assumes $\text{distinct } xs$ and $\text{keys } f \subseteq \{0..<\text{length } xs\}$
 shows $\text{idx-pm-of-pm } xs (\text{pm-of-idx-pm } xs f) = f$
 ⟨proof⟩

18.3 POT Orders

context *ordered-term*
begin

definition *is-pot-ord* :: *bool*
 where $\text{is-pot-ord} \longleftrightarrow (\forall u v. \text{component-of-term } u < \text{component-of-term } v \longrightarrow u \prec_t v)$

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 $\text{component-of-term } u < \text{component-of-term } v$
 shows $u \prec_t v$
 ⟨proof⟩

lemma *is-pot-ordD2*:
 assumes *is-pot-ord* and $u \preceq_t v$
 shows $\text{component-of-term } u \leq \text{component-of-term } v$
 ⟨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 ⇒ 'k) ⇒ 't ⇒ 't
where *map-component* f v = *term-of-pair* (pp-of-term v, f (component-of-term v))

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

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

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

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

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

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

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

end

18.4 Gröbner Bases of Syzygy Modules

locale *gd-inf-term* =
gd-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
for *pair-of-term*::'t ⇒ ('a::graded-dickson-powerprod × nat)
and *term-of-pair*::('a × nat) ⇒ 't
and *ord*::'a ⇒ 'a ⇒ bool (**infixl** ≤ 50)
and *ord-strict* (**infixl** < 50)
and *ord-term*::'t ⇒ 't ⇒ bool (**infixl** ≤_t 50)
and *ord-term-strict*::'t ⇒ 't ⇒ bool (**infixl** <_t 50)
begin

In order to compute a Gröbner basis of the syzygy module of a list *bs* of polynomials, one first needs to “lift” *bs* to a new list *bs'* by adding further components, compute a Gröbner basis *gs* of *bs'*, and then filter out those

elements of gs whose only non-zero components are those that were newly added to bs . Function *init-syzygy-list* takes care of constructing bs' , and function *filter-syzygy-basis* does the filtering. Function *proj-orig-basis*, finally, projects the Gröbner basis gs of bs' to a Gröbner basis of the original list bs .

definition *lift-poly-syz* :: $\text{nat} \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow \text{nat} \Rightarrow ('t \Rightarrow_0 'b::\text{semiring-1})$
where *lift-poly-syz* $n\ b\ i = \text{Abs-poly-mapping}$
 $(\lambda x. \text{if pair-of-term } x = (0, i) \text{ then } 1$
 $\text{else if } n \leq \text{component-of-term } x \text{ then lookup } b \text{ (map-component } (\lambda k. k - n) x)$
 $\text{else } 0)$

definition *proj-poly-syz* :: $\text{nat} \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::\text{semiring-1})$
where *proj-poly-syz* $n\ b = \text{Poly-Mapping.map-key } (\lambda x. \text{map-component } (\lambda k. k + n) x) b$

definition *cofactor-list-syz* :: $\text{nat} \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b::\text{semiring-1}) \text{ list}$
where *cofactor-list-syz* $n\ b = \text{map } (\lambda i. \text{proj-poly } i\ b) [0..<n]$

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

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

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

definition *syzygy-module-list* :: $('t \Rightarrow_0 'b) \text{ list} \Rightarrow ('t \Rightarrow_0 'b::\text{comm-ring-1}) \text{ set}$
where *syzygy-module-list* $bs = \text{atomize-poly } ' \text{ idx-pm-of-pm } bs\ ' \text{ pmdl.syzygy-module}$
 $(\text{set } bs)$

18.4.1 lift-poly-syz

lemma *keys-lift-poly-syz-aux*:

$\{x. (\text{if pair-of-term } x = (0, i) \text{ then } 1$
 $\text{else if } n \leq \text{component-of-term } x \text{ then lookup } b \text{ (map-component } (\lambda k. k - n)$
 $x)$
 $\text{else } 0) \neq 0\} \subseteq \text{insert } (\text{term-of-pair } (0, i)) \text{ (map-component } (\lambda k. k + n) ' \text{ keys } b)$
 $(\text{is } ?l \subseteq ?r) \text{ for } b::'t \Rightarrow_0 'b::\text{semiring-1}$
 $\langle \text{proof} \rangle$

lemma *lookup-lift-poly-syz*:

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 *proj-poly-syz* n (*lift-poly-syz* n p i) = p
 ⟨*proof*⟩

lemma *proj-poly-syz-eq-zero-iff*: *proj-poly-syz* n p = 0 \longleftrightarrow (*component-of-term* ‘
keys $p \subseteq \{0..<n\}$)
 ⟨*proof*⟩

lemma *component-of-lt-ge*:
assumes *is-pot-ord* **and** *proj-poly-syz* n $p \neq 0$
shows $n \leq$ *component-of-term* (*lt* p)
 ⟨*proof*⟩

lemma *lt-proj-poly-syz*:
assumes *is-pot-ord* **and** *proj-poly-syz* n $p \neq 0$
shows *lt* (*proj-poly-syz* n p) = *map-component* ($\lambda k. k - n$) (*lt* p) (**is** - = ?*l*)
 ⟨*proof*⟩

lemma *proj-proj-poly-syz*: *proj-poly* k (*proj-poly-syz* n p) = *proj-poly* ($k + n$) p
 ⟨*proof*⟩

lemma *poly-mapping-eqI-proj-syz*:
assumes *proj-poly-syz* n p = *proj-poly-syz* n q
and $\bigwedge k. k < n \implies$ *proj-poly* k p = *proj-poly* k q
shows $p = q$
 ⟨*proof*⟩

18.4.3 cofactor-list-syz

lemma *length-cofactor-list-syz* [*simp*]: *length* (*cofactor-list-syz* n p) = n
 ⟨*proof*⟩

lemma *cofactor-list-syz-nth*:
assumes $i < n$
shows (*cofactor-list-syz* n p) ! i = *proj-poly* i p
 ⟨*proof*⟩

lemma *cofactor-list-syz-zero* [*simp*]: *cofactor-list-syz* n 0 = *replicate* n 0
 ⟨*proof*⟩

lemma *cofactor-list-syz-plus*:
cofactor-list-syz n ($p + q$) = *map2* (+) (*cofactor-list-syz* n p) (*cofactor-list-syz* n q)
 ⟨*proof*⟩

18.4.4 *init-syzygy-list*

lemma *length-init-syzygy-list* [simp]: $\text{length } (\text{init-syzygy-list } bs) = \text{length } bs$
(proof)

lemma *init-syzygy-list-nth*:
assumes $i < \text{length } bs$
shows $(\text{init-syzygy-list } bs) ! i = \text{lift-poly-syz } (\text{length } bs) (bs ! i)$
(proof)

lemma *Keys-init-syzygy-list*:
 $\text{Keys } (\text{set } (\text{init-syzygy-list } bs)) =$
 $\text{map-component } (\lambda k. k + \text{length } bs) \text{ ' } \text{Keys } (\text{set } bs) \cup (\lambda i. \text{term-of-pair } (0, i))$
 $\text{' } \{0..<\text{length } bs\}$
(proof)

lemma *pp-of-Keys-init-syzygy-list-subset*:
 $\text{pp-of-term ' } \text{Keys } (\text{set } (\text{init-syzygy-list } bs)) \subseteq \text{insert } 0 (\text{pp-of-term ' } \text{Keys } (\text{set } bs))$
(proof)

lemma *pp-of-Keys-init-syzygy-list-superset*:
 $\text{pp-of-term ' } \text{Keys } (\text{set } bs) \subseteq \text{pp-of-term ' } \text{Keys } (\text{set } (\text{init-syzygy-list } bs))$
(proof)

lemma *pp-of-Keys-init-syzygy-list*:
assumes $bs \neq []$
shows $\text{pp-of-term ' } \text{Keys } (\text{set } (\text{init-syzygy-list } bs)) = \text{insert } 0 (\text{pp-of-term ' } \text{Keys } (\text{set } bs))$
(proof)

lemma *component-of-Keys-init-syzygy-list*:
 $\text{component-of-term ' } \text{Keys } (\text{set } (\text{init-syzygy-list } bs)) =$
 $(+) (\text{length } bs) \text{ ' } \text{component-of-term ' } \text{Keys } (\text{set } bs) \cup \{0..<\text{length } bs\}$
(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 $\text{monom-mult } c \ t \ s \in \text{syzygy-module-list } bs$
 $\langle \text{proof} \rangle$

lemma *pmdl-syzygy-module-list [simp]*: $\text{pmdl } (\text{syzygy-module-list } bs) = \text{syzygy-module-list } bs$
 $\langle \text{proof} \rangle$

The following lemma also holds without the distinctness constraint on bs , but then the proof becomes more difficult.

lemma *syzygy-module-listI'*:
assumes $\text{distinct } bs$ **and** $\text{sum-list } (\text{map2 } \text{mult-scalar } (\text{cofactor-list-syz } (\text{length } bs) \ s) \ bs) = 0$
and $\text{component-of-term ' keys } s \subseteq \{0..<\text{length } bs\}$
shows $s \in \text{syzygy-module-list } bs$
 $\langle \text{proof} \rangle$

lemma *component-of-syzygy-module-list*:
assumes $s \in \text{syzygy-module-list } bs$
shows $\text{component-of-term ' keys } s \subseteq \{0..<\text{length } bs\}$
 $\langle \text{proof} \rangle$

lemma *map2-mult-scalar-proj-poly-syz*:
 $\text{map2 } \text{mult-scalar } \ xs \ (\text{map } (\text{proj-poly-syz } n) \ ys) =$
 $\text{map } (\text{proj-poly-syz } n \circ (\lambda(x, y). \text{mult-scalar } x \ y)) \ (\text{zip } \ xs \ ys)$
 $\langle \text{proof} \rangle$

lemma *map2-times-proj*:
 $\text{map2 } (*) \ \ xs \ (\text{map } (\text{proj-poly } k) \ ys) = \text{map } (\text{proj-poly } k \circ (\lambda(x, y). \ x \odot \ y)) \ (\text{zip } \ xs \ ys)$
 $\langle \text{proof} \rangle$

Probably the following lemma also holds without the distinctness constraint on bs .

lemma *syzygy-module-list-subset*:
assumes $\text{distinct } bs$
shows $\text{syzygy-module-list } bs \subseteq \text{pmdl } (\text{set } (\text{init-syzygy-list } bs))$
 $\langle \text{proof} \rangle$

18.4.8 Cofactors

lemma *map2-mult-scalar-plus*:
 $\text{map2 } (\odot) \ (\text{map2 } (+) \ \ xs \ ys) \ zs = \text{map2 } (+) \ (\text{map2 } (\odot) \ \ xs \ zs) \ (\text{map2 } (\odot) \ \ ys \ zs)$
 $\langle \text{proof} \rangle$

lemma *syz-cofactors*:
assumes $p \in \text{pmdl } (\text{set } (\text{init-syzygy-list } bs))$

shows *proj-poly-syz* (length bs) p = sum-list (map2 mult-scalar (cofactor-list-syz (length bs) p) bs)
 ⟨proof⟩

18.4.9 Modules

lemma *pmdl-proj-orig-basis*:
assumes *pmdl* (set gs) = *pmdl* (set (init-syzygy-list bs))
shows *pmdl* (set (proj-orig-basis (length bs) gs)) = *pmdl* (set bs)
 ⟨proof⟩

lemma *pmdl-filter-syzygy-basis-subset*:
assumes *distinct* bs **and** *pmdl* (set gs) = *pmdl* (set (init-syzygy-list bs))
shows *pmdl* (set (filter-syzygy-basis (length bs) gs)) \subseteq *pmdl* (syzygy-module-list bs)
 ⟨proof⟩

lemma *ex-filter-syzygy-basis-adds-lt*:
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* (set gs)
and *pmdl* (set gs) = *pmdl* (set (init-syzygy-list bs))
and $f \in \text{pmdl}(\text{syzygy-module-list } bs)$ **and** $f \neq 0$
shows $\exists g \in \text{set}(\text{filter-syzygy-basis } (length\ bs)\ gs). g \neq 0 \wedge lt\ g\ adds_t\ lt\ f$
 ⟨proof⟩

lemma *pmdl-filter-syzygy-basis*:
fixes $bs::('t \Rightarrow_0 'b::field)\ list$
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* (set gs) **and**
pmdl (set gs) = *pmdl* (set (init-syzygy-list bs))
shows *pmdl* (set (filter-syzygy-basis (length bs) gs)) = *syzygy-module-list* bs
 ⟨proof⟩

18.4.10 Gröbner Bases

lemma *proj-orig-basis-isGB*:
assumes *is-pot-ord* **and** *is-Groebner-basis* (set gs) **and** *pmdl* (set gs) = *pmdl*
 (set (init-syzygy-list bs))
shows *is-Groebner-basis* (set (proj-orig-basis (length bs) gs))
 ⟨proof⟩

lemma *filter-syzygy-basis-isGB*:
assumes *is-pot-ord* **and** *distinct* bs **and** *is-Groebner-basis* (set gs)
and *pmdl* (set gs) = *pmdl* (set (init-syzygy-list bs))
shows *is-Groebner-basis* (set (filter-syzygy-basis (length bs) gs))
 ⟨proof⟩

end

end

19 Sample Computations of Syzygies

```

theory Syzygy-Examples
  imports Buchberger Algorithm-Schema-Impl Syzygy Code-Target-Rat
begin

```

19.1 Preparations

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

```

definition splus-pprod :: ('a::nat, 'b::nat) pp  $\Rightarrow$  -
  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* ($\lambda kv. (\text{map-component } ((+) i) (\text{fst } kv), \text{snd } kv)$) (*list-of-oalist-ntm* *xs*))

lemma *list-of-oalist-lift-keys*:

list-of-oalist-ntm (*lift-keys* *i xs*) = (*map-raw* ($\lambda kv. (\text{map-component } ((+) i) (\text{fst } kv), \text{snd } kv)$) (*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* $\lambda x::('a, 'b) \text{ pp} \times \text{nat}. x \lambda x. x$
cmp-term

rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.monom-mult* = *monom-mult-pprod*
and *pprod.mult-scalar* = *mult-scalar-pprod*
and *pprod.adds-term* = *adds-term-pprod*
for *cmp-term* :: $(('a::\text{nat}, 'b::\text{nat}) \text{ pp} \times \text{nat}) \text{ nat-term-order}$
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* ∧ *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*: $\text{pprod}'.\text{is-pot-ord} (\text{TYPE}('a::\text{nat})) (\text{TYPE}('b::\text{nat})) (\text{POT to})$

$\langle \text{proof} \rangle$

definition $\text{Vec}_0 :: \text{nat} \Rightarrow ((a, \text{nat}) \text{pp} \Rightarrow_0 'b) \Rightarrow ((a::\text{nat}, \text{nat}) \text{pp} \times \text{nat}) \Rightarrow_0 'b::\text{semiring-1}$ **where**

$\text{Vec}_0 i p = \text{mult-scalar-pprod } p (\text{Poly-Mapping.single } (0, i) 1)$

definition *syzygy-basis to bs =*

$\text{filter-syzygy-basis-pprod} (\text{length } bs) (\text{map fst } (\text{gb-pprod } (\text{POT to}) (\text{map } (\lambda p. (p, ())) (\text{init-syzygy-list-pprod } bs)) ()))$

thm $\text{pprod}'.\text{aux.filter-syzygy-basis-isGB}[\text{OF POT-is-pot-ord}]$

lemma *lift-poly-syz-MP-oalist* [code]:

$\text{lift-poly-syz-pprod } n (\text{MP-oalist } xs) i = \text{MP-oalist} (\text{Oalist-insert-ntm } ((0, i), 1) (\text{lift-keys-pprod } n xs))$

$\langle \text{proof} \rangle$

19.2 Computations

experiment begin interpretation *trivariate₀-rat* $\langle \text{proof} \rangle$

lemma

$\text{syzygy-basis DRLEX} [\text{Vec}_0 0 (X^2 * Z \wedge 3 + 3 * X^2 * Y), \text{Vec}_0 0 (X * Y * Z + 2 * Y^2)] =$

$[\text{Vec}_0 0 (C_0 (1 / 3) * X * Y * Z + C_0 (2 / 3) * Y^2) + \text{Vec}_0 1 (C_0 (-1 / 3) * X^2 * Z \wedge 3 - X^2 * Y)]$

$\langle \text{proof} \rangle$

value [code] $\text{syzygy-basis DRLEX} [\text{Vec}_0 0 (X^2 * Z \wedge 3 + 3 * X^2 * Y), \text{Vec}_0 0 (X * Y * Z + 2 * Y^2), \text{Vec}_0 0 (X - Y + 3 * Z)]$

lemma

$\text{map fst } (\text{gb-pprod } (\text{POT DRLEX}) (\text{map } (\lambda p. (p, ())) (\text{init-syzygy-list-pprod} [\text{Vec}_0 0 (X \wedge 4 + 3 * X^2 * Y), \text{Vec}_0 0 (Y \wedge 3 + 2 * X * Z), \text{Vec}_0 0 (Z^2 - X - Y)])) ())) =$

[
 $\text{Vec}_0 0 1 + \text{Vec}_0 3 (X \wedge 4 + 3 * X^2 * Y),$
 $\text{Vec}_0 1 1 + \text{Vec}_0 3 (Y \wedge 3 + 2 * X * Z),$
 $\text{Vec}_0 0 (Y \wedge 3 + 2 * X * Z) - \text{Vec}_0 1 (X \wedge 4 + 3 * X^2 * Y),$
 $\text{Vec}_0 2 1 + \text{Vec}_0 3 (Z^2 - X - Y),$
 $\text{Vec}_0 1 (Z^2 - X - Y) - \text{Vec}_0 2 (Y \wedge 3 + 2 * X * Z),$
 $\text{Vec}_0 0 (Z^2 - X - Y) - \text{Vec}_0 2 (X \wedge 4 + 3 * X^2 * Y),$
 $\text{Vec}_0 0 (- (Y \wedge 3 * Z^2) + Y \wedge 4 + X * Y \wedge 3 + 2 * X^2 * Z + 2 * X * Y * Z - 2 * X * Z \wedge 3) +$
 $\text{Vec}_0 1 (X \wedge 4 * Z^2 - X \wedge 5 - X \wedge 4 * Y - 3 * X \wedge 3 * Y - 3 * X^2 * Y^2 + 3 * X^2 * Y * Z^2)$
]

<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 adds t | t 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 $s \preceq t \iff s \text{ adds } t$

<proof>

lemma *adds-iff-deg-le-univariate*:

assumes $s \in .[X]$ **and** $t \in .[X]$

shows $s \text{ adds } t \iff \text{deg-pm } s \leq \text{deg-pm } t$

<proof>

corollary *ord-iff-deg-le-univariate*: $s \in .[X] \implies t \in .[X] \implies s \preceq t \iff \text{deg-pm } s \leq \text{deg-pm } t$

<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 $lpp\ (dehomogenize\ x\ p) = except\ (lpp\ p)\ \{x\}$

<proof>

lemma *isGB-dehomogenize*:

assumes *is-hom-ord x* **and** *finite X* **and** $G \subseteq P[X]$ **and** *punit.is-Groebner-basis G*

and $\bigwedge g. g \in G \implies homogeneous\ g$

shows *punit.is-Groebner-basis (dehomogenize x ' G)*

<proof>

end

context *extended-ord-pm-powerprod*

begin

lemma *extended-ord-lp*:

assumes $None \notin indets\ p$

shows $restrict-indets-pp\ (extended-ord.lpp\ p) = lpp\ (restrict-indets\ p)$

<proof>

lemma *restrict-indets-reduced-GB*:

assumes *finite X* **and** $F \subseteq P[X]$

shows *punit.is-Groebner-basis (restrict-indets ' extended-ord.punit.reduced-GB (homogenize None ' extend-indets ' F))*

(is ?thesis1)

and *ideal (restrict-indets ' extended-ord.punit.reduced-GB (homogenize None ' extend-indets ' F)) = ideal F*

(is ?thesis2)

and $restrict-indets\ ' extended-ord.punit.reduced-GB\ (homogenize\ None\ ' extend-indets\ ' F) \subseteq P[X]$

(is ?thesis3)

<proof>

end

end

References

- [1] W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Society, July 1994.
- [2] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical

- Institute, University of Innsbruck, Austria, 1965. English translation in *Journal of Symbolic Computation* 41(3–4):475–511, Special Issue on Logic, Mathematics, and Computer Science: Interactions.
- [3] B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems (An Algorithmic Criterion for the Solvability of an Algebraic System of Equations). *Aequationes Mathematicae*, pages 374–383, 1970. (English translation in *Gröbner Bases and Applications (Proceedings of the International Conference “33 Years of Gröbner Bases”, 1998)*, London Mathematical Society Lecture Note Series 251, Cambridge 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.