A Rule-Based Component-Free Vector Algebra Package

(Thesis format: Monograph)

by

Songxin Liang

Graduate Program in Applied Mathematics

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario, Canada
April 2006

THE UNIVERSITY OF WESTERN ONTARIO

FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

The thesis by

Songxin Liang

entitled

A Rule-Based Component-Free Vector Algebra Package

is accepted in partial fulfilment of the

requirements for the degree of

Master of Science

Date April 17, 2006

Dr. Christopher Essex

Chair of the Thesis Examining Board

Abstract

This thesis presents a rule-based component-free vector algebra package which is developed using computer programming language Aldor. Besides the general vector algebra operations such as addition, scalar multiplication, scalar product and vector product, this package provides simplification functionality which successfully overcomes the difficulties encountered by existing packages. In chapter 2, we describe the algebraic formulas needed for the package. In chapter 3, we discuss the implementation details of this package in Aldor. They include data representation, term order, implementations of transformation rules, implementations of vector algebra operations, simplifications of vector expressions and proofs of vector identities. In chapter 4, we give some examples including the one used by Stoutemyer to test our package. In the appendix A, we include the Aldor source code of the package for reference.

Keywords: Vector algebra package, component-free, transformation rules, simplification

Acknowledgement

I would like to express my sincere thanks to Dr. David Jeffrey and Dr. Stephen Watt. Under their supervision, there are no pressures, there are only motivations. I have learned a lot from them. They always know more than I can learn.

I appreciate the helpful suggestions coming from Dr. Marc Moreno Maza. His suggestions greatly saved my time in my research.

Many thanks go to the staff in the Department of Applied Mathematics at UWO. Special thanks should go to Pat Malone who always gives me helpful and instant information about the graduate program.

Last but not least, I would like to take this opportunity to express my gratitude to my wife Yanfei Wu. Without her understanding and support, this thesis would not have been possible.

Table of Contents

Chapter 1

Introduction

‘The numerical description of a vector requires three numbers, but nothing prevents us from using a single letter for its symbolical designation. An algebra or analytical method in which a single letter or other expression is used to specify a vector may be called a vector algebra or vector analysis.’
J. Willard Gibbs [9]

This sentence introduced Gibbs’s notation for vectors in a privately printed pamphlet that was dated 1881. It is interesting to compare the introduction of vectors in this publication with the notation in a paper published 2 years earlier by Gibbs [8], in which the equations of dynamics were presented entirely in component form. Gibbs’s notation was not immediately accepted by all.

‘Even Prof. Willard Gibbs must be ranked as one of the retarders of quaternion progress, in virtue of his pamphlet on *Vector Analysis*, a sort of hermaphrodite monster, compounded of the notations of Hamilton and Grassmann”.
Peter Guthrie Tait [19]

In spite of Tait's resistance, vector algebra and vector calculus have found many applications throughout engineering and science. Vector analysis often simplifies the derivation of mathematical theorems and the statements of physical laws, while vector notation can often clearly convey geometric or physical interpretations that greatly facilitate understanding.

In section 1.1, some basic concepts about vector spaces and their operations are introduced [14, 5, 15]. In section 1.2, some background on vector analysis packages is presented and the reason why we want to develop the current package is discussed.

1.1 Basic Concepts and Notation

In this section, we define a **vector-product space**, which is an inner-product vector space with an operation called vector product defined on it. We begin by defining vector space and inner-product space.

A **vector space** V over a **field** F consists of a nonempty set V of objects called **vectors** which can be added, multiplied by an element in F, and for which the following axioms hold. If $\mathbf{v}$ and $\mathbf{w}$ are two vectors in V, their sum is expressed as $\mathbf{v} + \mathbf{w}$, and the scalar product of $\mathbf{v}$ by $a \in F$ is denoted as $a * \mathbf{v}$. These operations are called **addition** and **scalar multiplication** respectively. Sometimes we simply denote the vector space by $(V, +, *)$. The axioms that $(V, +, *)$ should satisfy are:

A1. If $\mathbf{u}, \mathbf{v} \in V$, then $\mathbf{u} + \mathbf{v} \in V$.

A2. $\forall\, \mathbf{u}, \mathbf{v} \in V$, $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$.

A3. $\forall\, \mathbf{u}, \mathbf{v}, \mathbf{w} \in V$, $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$.

A4. $\exists\, \mathbf{0} \in V$ such that $\forall\, \mathbf{v} \in V, \mathbf{v} + \mathbf{0} = \mathbf{v}$.

A5. $\forall\, \mathbf{v} \in V, \exists\, \text{-}\mathbf{v} \in V$ such that $\mathbf{v} + (\text{-}\mathbf{v}) = \mathbf{0}$.

A6. $\forall\, \mathbf{v} \in V$ and $\forall\, a \in F$, $a * \mathbf{v} \in V$.

A7. $\forall\, a \in F$ and $\forall\, \mathbf{v}, \mathbf{w} \in V$, $a * (\mathbf{v} + \mathbf{w}) = a * \mathbf{v} + a * \mathbf{w}$.

A8. $\forall\, a, b \in F$ and $\forall\, \mathbf{v} \in V$, $(a + b) * \mathbf{v} = a * \mathbf{v} + b * \mathbf{v}$.

A9. $\forall\, a, b \in F$ and $\forall\, \mathbf{v} \in V$, $a * (b * \mathbf{v}) = (ab) * \mathbf{v}$.

A10. $\forall\, \mathbf{v} \in V, 1 * \mathbf{v} = \mathbf{v}$.

There are many examples of vector spaces. For example, $\mathbb{R}^n$ with matrix addition and scalar multiplication forms a vector space over $\mathbb{R}$. Let $V = \{(x, y) |\, x, y \in \mathbb{C}\}$. We define the addition $+$ and scalar multiplication $*$ as follows. For $(x_1, y_1), (x_2, y_2) \in V$, $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$. For $(x, y) \in V$ and $a \in \mathbb{C}$, $a * (x, y) = (ay, ax)$. It is easy to check that $(V, +, *)$ is a vector space over $\mathbb{C}$.

If V is a vector space over $\mathbb{R}$, then an **inner product** on V is a function that assigns a number $\langle \mathbf{v}, \mathbf{w} \rangle$ to every pair $\mathbf{v}, \mathbf{w}$ of vectors in V in such a way that the following axioms are satisfied.

P1. $\forall\, \mathbf{v}, \mathbf{w} \in V, \langle \mathbf{v}, \mathbf{w} \rangle \in \mathbb{R}$.

P2. $\forall\, \mathbf{v}, \mathbf{w} \in V, \langle \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{w}, \mathbf{v} \rangle$.

P3. $\forall\, \mathbf{u}, \mathbf{v}, \mathbf{w} \in V, \langle \mathbf{v} + \mathbf{w}, \mathbf{u} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle + \langle \mathbf{w}, \mathbf{u} \rangle$.

P4. $\forall\, \mathbf{v}, \mathbf{w} \in V$ and $\forall\, r \in \mathbb{R}$, $\langle r * \mathbf{v}, \mathbf{w} \rangle = r \langle \mathbf{v}, \mathbf{w} \rangle$.

P5. For all $\mathbf{v} \neq \mathbf{0}$ in V, $\langle \mathbf{v}, \mathbf{v} \rangle \ > 0$.

For the vector space $(\mathbb{R}^n, +, *)$ over $\mathbb{R}$, the most common inner product is defined using components. If $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ with the standard basis of $\mathbb{R}^n$, then the inner product of $\mathbf{v}$ and $\mathbf{w}$ is

$$\langle \mathbf{v}, \mathbf{w} \rangle = \sum_{i=1}^{n} v_i w_i \ .$$

For vector space $(\mathbb{R}^3, +, *)$ over $\mathbb{R}$, we can define a special inner product for it: scalar product. This is usually denoted as a dot product. Let $\mathbf{a}, \mathbf{b}$ be two non-zero vectors in $\mathbb{R}^3$ and a, b and θ respectively be the magnitude of $\mathbf{a}$, the magnitude of $\mathbf{b}$, and the angle between $\mathbf{a}$ and $\mathbf{b}$, where $0 \leq \theta \leq \pi$. The **scalar product** of $\mathbf{a}$ and $\mathbf{b}$ is defined by

$$\mathbf{a} \cdot \mathbf{b} = a\, b\, \cos\theta.$$

If either $\mathbf{a}$ or $\mathbf{b}$ is zero, we define $\mathbf{a} \cdot \mathbf{b}$ to be zero. It is easy to check that the scalar product is an inner product for $(\mathbb{R}^3, +, *)$, and it is well known that the definition using angle is equivalent to that using components. It should be noted that the inner product is defined by axioms P1 to P5, and the component-based definitions are particular possible realizations.

Although Gibbs originally called the product direct product, and although the term scalar product is available, many books, and also MAPLE, call it dot product after one particular notation.

We can now define vector product. We first give the common definition used in physics and engineering that applies in the vector space $(\mathbb{R}^3, +, *)$. Let $\mathbf{a}, \mathbf{b}$ be non-parallel vectors in $\mathbb{R}^3$ and again let a, b and θ respectively be the magnitude of $\mathbf{a}$, the magnitude of $\mathbf{b}$, and the angle between $\mathbf{a}$ and $\mathbf{b}$. Let $\mathbf{n}$ be the vector such that

- the magnitude of $\mathbf{n}$ is unity,
- $\mathbf{n}$ is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$,

- the system $\mathbf{a}, \mathbf{b}, \mathbf{n}$ is right-handed.

Then, $\mathbf{n}$ is uniquely determined by $\mathbf{a}$ and $\mathbf{b}$. The **vector product** $\mathbf{a} \wedge \mathbf{b}$ of the vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by

$$\mathbf{a} \wedge \mathbf{b} = (a\, b\, \sin\theta) * \mathbf{n}.$$

If $\mathbf{a}$ and $\mathbf{b}$ are parallel vectors, we define the vector product $\mathbf{a} \wedge \mathbf{b}$ to be the zero vector.

An equivalent definition in terms of components is that $\mathbf{c} = \mathbf{a} \wedge \mathbf{b}$ with

$$(c_1, c_2, c_3) = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1) \ ,$$

where the standard basis of $\mathbb{R}^3$ is used.

It should be noted that this definition is different from the definitions given for an inner-product vector space, in that for inner-product spaces, the component definitions constituted a realization of the definitions, whereas here they are the common primary definition. Abstract definitions of vector product have been attempted in connection with the question of whether a vector product can be defined in dimensions higher than 3, see for example [16, 17], but are still not standard. We note that a vector product can be defined in 7 dimensions, but one of the properties must be weakened. Therefore we confine ourselves here to the 3 dimensional case. Since the object of this work is a component-free description of a vector-product space, we deduce some abstract properties from the primary definitions. Note that if a different component realization of the inner-product space were defined, then the component-based definition of vector product would also have to be changed to retain the abstract properties.

As with the scalar product, the vector product is often called cross product (again Maple does this), however, in this thesis we follow an older notation due to Chapman [4, 13], and use the wedge.

The main properties of vector product are as follows, where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$, and $r \in \mathbb{R}$.

- $\mathbf{a} \wedge \mathbf{b}$ is a vector perpendicular to both $\mathbf{a}$ and $\mathbf{b}$.
- $\mathbf{a} \wedge \mathbf{b} = \mathbf{0}$ if and only if $\mathbf{a}$ and $\mathbf{b}$ are parallel.
- $\mathbf{a} \wedge \mathbf{b} = -(\mathbf{b} \wedge \mathbf{a})$.
- $(r * \mathbf{a}) \wedge \mathbf{b} = r * (\mathbf{a} \wedge \mathbf{b}) = \mathbf{a} \wedge (r * \mathbf{b})$.
- $\mathbf{a} \wedge (\mathbf{b} + \mathbf{c}) = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \wedge \mathbf{c}$.
- In general, $(\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c} \neq \mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c})$.

Our package deals with the vector space $(\mathbb{R}^3, +, *)$ together with the scalar product $\cdot$ and vector product $\wedge$. In other words, we deal with the algebraic structure $(\mathbb{R}^3, +, *, \cdot, \wedge)$.

Before ending the section, we would like to introduce the last, but not least concept: scalar triple product. For $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$, the **scalar triple product** $(\mathbf{abc})$ or $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ of the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ is defined by

$$(\mathbf{abc}) = (\mathbf{a} \wedge \mathbf{b}) \cdot \mathbf{c}.$$

There are some important properties for scalar triple product:

- $(\mathbf{abc}) \in \mathbb{R}$.
- $(\mathbf{b} \wedge \mathbf{c}, \mathbf{c} \wedge \mathbf{a}, \mathbf{a} \wedge \mathbf{b}) = (\mathbf{a}, \mathbf{b}, \mathbf{c})^2$.
- $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are independent if and only if $(\mathbf{abc}) \neq 0$.

1.2 Background and Motivation

Almost all well known computer algebra systems provide general vector operations. Some of them even have built-in vector analysis packages. For example, the *VectorAnalysis* package for Mathematica [22], the *Vector33* package for Reduce [11] and the *VectorCalculus* package for Maple. There are also many vector analysis packages using the well known computer algebra systems. For example, the *VecCalc* package uses Maple [3], the *Vect* package uses Macsyma [10] and the *OrthoVec* package uses Reduce [6]. All these packages can provide all or most of the general vector algebra operations such as addition, scalar multiplication, scalar product, vector product, and vector calculus operations such as derivative, gradient, divergence and curl. The following are some performances of the *VectorCalculus* package for Maple.

```
> with(VectorCalculus)
> CrossProduct(A, B)
     Error, (in VectorCalculus:-CrossProduct) the first argument
     must either be the differential operator Del, or a three
     dimensional VectorField, got A
> DotProduct(A, B)
     A . B
```

It is curious that MAPLE checks the arguments of `CrossProduct` and requires an explicit set of components, but `DotProduct` does not. In any event, it is clear that the package does not expect abstract vectors. Once explicit components are given, the package is able to perform common tasks.

```
> A := <a, b, c>
     A := a e_x  + b e_y  + c e_z
```

```
> B := <d, e, f>

      B := d e_x  + e e_y  + f e_z

> CrossProduct(A, B)

      (b f - c e) e_x  + (c d - a f) e_y  + (a e - b d) e_z

> DotProduct(A, B)

      a d + b e + c f
```

However, almost all these packages can only perform component dependent operations; they cannot perform component-free operations. This means that before one can do any vector operations, one should set the components for all vectors involved. That would be a bad situation when one wants to deal with problems involving many vectors and only wants to know the relationship among them, because the expansion of vector expressions into specific components is usually tedious and full of opportunities for mistakes. If one has a component-free system, one will be free from the distracting details of individual components and can concentrate on the meaningful results of the problems.

Compared with component dependent systems which have a systematic method for deriving mathematical statements, component-free systems are more difficult and challenging because vector algebra has a strange and intriguing structure (see [18]):

- Vectors are not closed under the scalar product operation. If $\mathbf{p}$, $\mathbf{q}$ are vectors, then $\mathbf{p} \cdot \mathbf{q}$ is a scalar.
- The scalar product is commutative while the vector product is anticommutative. If $\mathbf{p}$, $\mathbf{q}$ are vectors, then $\mathbf{p} \cdot \mathbf{q} = \mathbf{q} \cdot \mathbf{p}$ while $\mathbf{p} \wedge \mathbf{q} = -\mathbf{q} \wedge \mathbf{p}$.
- Neither is associative. If $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ are vectors, then $\mathbf{p} \wedge (\mathbf{q} \wedge \mathbf{r}) \neq (\mathbf{p} \wedge \mathbf{q}) \wedge \mathbf{r}$, whereas $\mathbf{p} \cdot (\mathbf{q} \cdot \mathbf{r})$ and $(\mathbf{p} \cdot \mathbf{q}) \cdot \mathbf{r}$ are invalid.

- Neither has a multiplicative unit. There does not exist a fixed vector $\mathbf{u}$ such that for any vector $\mathbf{p}$, $\mathbf{u} \wedge \mathbf{p} = \mathbf{p}$ or $\mathbf{p} \wedge \mathbf{u} = \mathbf{p}$ or $\mathbf{u} \cdot \mathbf{p} = \mathbf{p}$ or $\mathbf{p} \cdot \mathbf{u} = \mathbf{p}$.

- Both admit zero divisors. For any vector $\mathbf{p}$, $\mathbf{p} \wedge \mathbf{p} = \mathbf{0}$; if $\mathbf{q}$ is a vector perpendicular to $\mathbf{p}$, then $\mathbf{p} \cdot \mathbf{q} = 0$.

- Both are connected via ordinary scalar multiplication $*$ by the strange side relation $\mathbf{p} \wedge (\mathbf{q} \wedge \mathbf{r}) = (\mathbf{p} \cdot \mathbf{r}) * \mathbf{q} - (\mathbf{p} \cdot \mathbf{q}) * \mathbf{r}$

There exist a lot of component dependent vector analysis packages. On the contrary, few component-free vector analysis packages are found due to the above reasons. To our knowledge, only the packages by Fiedler [7], Qin et al. [20] and Stoutemyer [18] involve component-free vector operations. However, the emphasis of these packages is still on component dependent operations, and only the package by Stoutemyer provides non-trivial simplification examples. But even in Stoutemyer's package, some simplification problems still remain unsolved. For example, when he tried to simplify the vector expression

$$(\mathbf{a} \wedge \mathbf{b}) \wedge (\mathbf{b} \wedge \mathbf{c}) \cdot (\mathbf{c} \wedge \mathbf{a}) - (\mathbf{a} \cdot (\mathbf{b} \wedge \mathbf{c}))^2$$

which should be simplified to zero, he only got

$$-\mathbf{a} \cdot \mathbf{c} \wedge (\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c} * \mathbf{b} - \mathbf{a} \cdot \mathbf{b} \wedge (\mathbf{b} \wedge \mathbf{c})) - (\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c})^2.$$

When he tried to simplify the resulting expression again, instead of getting the desired result 0, he could only get

$$\mathbf{a} \cdot (\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c} * \mathbf{b}) \wedge \mathbf{c} - (\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c})^2.$$

He explained that the scalar factor $\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c}$ could be factored out, clearly revealing that the expression is zero, but the built-in scalar-factoring-out mechanism does not recognize that $\mathbf{a} \cdot \mathbf{b} \wedge \mathbf{c}$ is a scalar despite its vector components.

Therefore, it is very necessary and meaningful to develop a component-free vector analysis package. In order to achieve this goal, it is important to choose a proper programming language, a good methodology and some suitable data representations.

1.3 Outline of Thesis

In computer algebra, there are at least two ways to compute with symbolic expressions. One is zero recognition. Another is rewriting expressions in ways that the user desires. We can illustrate this using polynomials, whose treatment in computer systems is more advanced than the treatment of abstract vector analysis. The expression

$$(x^2 - 49) - (x - 7)(x + 7) \ ,$$

is actually zero. It is obviously important for a system to be able to discover this, and this is an example of the first type of computation. However, another type of computation is to rewrite the polynomial

$$x^3 + 7\,x^2 + 2\,x - 40$$

in factored form. The expression is not zero, but the user wants us to rewrite it. This is one example of another type of computation. In this thesis, we shall be concerned with the first type of computation.

In this thesis, we present a rule-based component-free vector algebra package [12] which is developed using computer programming language Aldor. Besides the general vector algebra operations, this package provides simplification functionality which successfully overcomes the difficulties encountered in Stoutemyer's package. In Chapter 2, we describe the basic algebraic formulas on which the transformation rules of our package are based. Chapter 3 is the main part of the thesis. We discuss the implementation details of this package in Aldor. They include data representation, term order, implementations of transformation rules, implementations of vector algebra operations, simplifications of vector expressions and proofs of vector identities. In Chapter 4, we use some examples including the one used by Stoutemyer to test our package. Some of these examples are quite complicated and big. In the Appendix A, we include the Aldor source code of the package for reference.

Chapter 2

Transformation Rules

In some degree, our package is a production system in the terminology of artificial intelligence [21]. It is important to choose a suitable set of transformation rules (production rules) carefully. A suitable set of transformation rules is similar to a complete set of axioms in geometry. But they are not the same. In a complete set of axioms, any axiom is independent of the others, but in a suitable set of transformation rules, a transformation rule may be dependent on the others. The reason is that human beings use axioms while computers use transformation rules to think, and human beings are smarter than computers!

2.1 Algebraic Formulas

The transformation rules for our package are based on a set of algebraic formulas. In the following, we first choose a set of algebraic formulas which is needed for our package. Then, we will prove those which are not always seen in the literature or not obvious. The reasonableness of our choice will be given in chapter 3.

The algebraic formulas are as follows. If $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, $\mathbf{g}$ and $\mathbf{h}$ are vectors, then

$$\mathbf{a}\wedge\mathbf{a} = \mathbf{0}\ . \tag{2.1}$$

$$\mathbf{a}\wedge\mathbf{b} = -\mathbf{b}\wedge\mathbf{a}\ ; \qquad \mathbf{a}\cdot\mathbf{b} = \mathbf{b}\cdot\mathbf{a}\ . \tag{2.2}$$

$$(\mathbf{abc}) = (\mathbf{bca}) = (\mathbf{cab}) = -(\mathbf{acb}) = -(\mathbf{cba}) = -(\mathbf{bac})\ . \tag{2.3}$$

$$(\mathbf{aab}) = (\mathbf{abb}) = (\mathbf{aba}) = 0\ . \tag{2.4}$$

$$\mathbf{a}\wedge(\mathbf{b}\wedge\mathbf{c}) = (\mathbf{a}\cdot\mathbf{c})*\mathbf{b} - (\mathbf{a}\cdot\mathbf{b})*\mathbf{c}\ . \tag{2.5}$$

$$(\mathbf{a}\wedge\mathbf{b})\wedge(\mathbf{c}\wedge\mathbf{d}) = (\mathbf{abd})*\mathbf{c} - (\mathbf{abc})*\mathbf{d}\ . \tag{2.6}$$

$$(\mathbf{a}\wedge\mathbf{b})\cdot(\mathbf{c}\wedge\mathbf{d}) = (\mathbf{a}\cdot\mathbf{c})(\mathbf{b}\cdot\mathbf{d}) - (\mathbf{a}\cdot\mathbf{d})(\mathbf{b}\cdot\mathbf{c})\ . \tag{2.7}$$

$$\begin{aligned}(\mathbf{abc})*\mathbf{d} &= (\mathbf{d}\cdot\mathbf{a})*(\mathbf{b}\wedge\mathbf{c}) + (\mathbf{d}\cdot\mathbf{b})*(\mathbf{c}\wedge\mathbf{a})\\ &\quad +(\mathbf{d}\cdot\mathbf{c})*(\mathbf{a}\wedge\mathbf{b})\ . \end{aligned}\tag{2.8}$$

$$\begin{aligned}(\mathbf{abc})*(\mathbf{d}\wedge\mathbf{h}) &= [(\mathbf{b}\wedge\mathbf{c})\cdot(\mathbf{d}\wedge\mathbf{h})]*\mathbf{a} + [(\mathbf{c}\wedge\mathbf{a})\cdot(\mathbf{d}\wedge\mathbf{h})]*\mathbf{b}\\ &\quad +[(\mathbf{a}\wedge\mathbf{b})\cdot(\mathbf{d}\wedge\mathbf{h})]*\mathbf{c}\ . \end{aligned}\tag{2.9}$$

$$(\mathbf{d}\cdot\mathbf{h})(\mathbf{abc}) = (\mathbf{d}\cdot\mathbf{a})(\mathbf{hbc}) + (\mathbf{d}\cdot\mathbf{b})(\mathbf{ahc}) + (\mathbf{d}\cdot\mathbf{c})(\mathbf{abh})\ . \tag{2.10}$$

$$\begin{aligned}(\mathbf{abc})(\mathbf{dgh}) &= [(\mathbf{b}\wedge\mathbf{c})\cdot(\mathbf{d}\wedge\mathbf{g})](\mathbf{a}.\mathbf{h}) + [(\mathbf{c}\wedge\mathbf{a})\cdot(\mathbf{d}\wedge\mathbf{g})](\mathbf{b}\cdot\mathbf{h})\\ &\quad +[(\mathbf{a}\wedge\mathbf{b})\cdot(\mathbf{d}\wedge\mathbf{g})](\mathbf{c}\cdot\mathbf{h})\ . \end{aligned}\tag{2.11}$$

Proof. (2.1), (2.2), (2.3), (2.4) are obvious by the definitions of vector operations.

(2.5) can be seen in most of the literature. Note that in 7 dimensions, this transformation is no longer valid [17].

For (2.6), replacing $\mathbf{a}$ with $\mathbf{a}\wedge\mathbf{b}$, $\mathbf{b}$ with $\mathbf{c}$ and $\mathbf{c}$ with $\mathbf{d}$ in (2.5), we get the result.

For (2.7), by(2.2), (2.3) and (2.5),

$$\begin{aligned}(\mathbf{a}\wedge\mathbf{b})\cdot(\mathbf{c}\wedge\mathbf{d}) &= (\mathbf{a},\mathbf{b},\mathbf{c}\wedge\mathbf{d}) = (\mathbf{b},\mathbf{c}\wedge\mathbf{d},\mathbf{a}) \\ &= (\mathbf{b}\wedge(\mathbf{c}\wedge\mathbf{d}))\cdot\mathbf{a} = \mathbf{a}\cdot(\mathbf{b}\wedge(\mathbf{c}\wedge\mathbf{d})) \\ &= \mathbf{a}\cdot[(\mathbf{b}\cdot\mathbf{d})*\mathbf{c}-(\mathbf{b}\cdot\mathbf{c})*\mathbf{d}] = (\mathbf{a}\cdot\mathbf{c})(\mathbf{b}\cdot\mathbf{d})-(\mathbf{a}\cdot\mathbf{d})(\mathbf{b}\cdot\mathbf{c}) \\ &= (\mathbf{a}\cdot\mathbf{c})(\mathbf{b}\cdot\mathbf{d})-(\mathbf{a}\cdot\mathbf{d})(\mathbf{b}\cdot\mathbf{c}).\end{aligned}$$

Now let's prove (2.8). There are two cases. If $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are non-coplanar, then $\mathbf{a}\wedge\mathbf{b}$, $\mathbf{b}\wedge\mathbf{c}$, $\mathbf{c}\wedge\mathbf{a}$ are also non-coplanar and we have $(\mathbf{b}\wedge\mathbf{c},\mathbf{c}\wedge\mathbf{a},\mathbf{a}\wedge\mathbf{b}) = (\mathbf{a},\mathbf{b},\mathbf{c})^2 \neq 0$. So, we can express $\mathbf{d}$ as

$$\mathbf{d} = x*(\mathbf{b}\wedge\mathbf{c}) + y*(\mathbf{c}\wedge\mathbf{a}) + z*(\mathbf{a}\wedge\mathbf{b}). \tag{2.12}$$

Taking scalar product of both sides of (2.12) with $\mathbf{a}$, by (2.4) we get $\mathbf{d}\cdot\mathbf{a} = (\mathbf{abc})x$, and so, $x = (\mathbf{d}\cdot\mathbf{a})/(\mathbf{abc})$. Similarly, we can get $y = (\mathbf{d}\cdot\mathbf{b})/(\mathbf{abc})$ and $z = (\mathbf{d}\cdot\mathbf{c})/(\mathbf{abc})$. Therefore, by (2.12) we get the desired result.

If $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are coplanar. Then $(\mathbf{abc}) = 0$. Without loss of generality, we can express $\mathbf{c}$ as $\mathbf{c} = x*\mathbf{a} + y*\mathbf{b}$. Then, by (2.1) and (2.2), the right side of (2.8) $=$ $x(\mathbf{d}\cdot\mathbf{a})*(\mathbf{b}\wedge\mathbf{a}) + y(\mathbf{d}\cdot\mathbf{b})*(\mathbf{b}\wedge\mathbf{a}) - [x(\mathbf{d}\cdot\mathbf{a}) + y(\mathbf{d}\cdot\mathbf{b})]*(\mathbf{b}\wedge\mathbf{a}) = \mathbf{0} =$ the left side of (2.8).

For (2.9), we only need to prove

$$(\mathbf{abc})*\mathbf{g} = (\mathbf{gbc})*\mathbf{a} + (\mathbf{agc})*\mathbf{b} + (\mathbf{abg})*\mathbf{c}$$

if we replace $\mathbf{d}\wedge\mathbf{h}$ with $\mathbf{g}$. We can use a similar method as in the proof of (2.8). So we omit it.

For (2.10), taking scalar product of both sides of (2.8) with $\mathbf{h}$, we get the result.

For (2.11), exchanging $\mathbf{d}$ and $\mathbf{h}$ in (2.10) we get

$$(\mathbf{h}\cdot\mathbf{d})(\mathbf{abc}) = (\mathbf{h}\cdot\mathbf{a})(\mathbf{dbc}) + (\mathbf{h}\cdot\mathbf{b})(\mathbf{adc}) + (\mathbf{h}\cdot\mathbf{c})(\mathbf{abd}).$$

That is

$$(\mathbf{d}\cdot\mathbf{h})(\mathbf{abc}) = [(\mathbf{b}\wedge\mathbf{c})\cdot\mathbf{d}](\mathbf{a}\cdot\mathbf{h}) + [(\mathbf{c}\wedge\mathbf{a})\cdot\mathbf{d}](\mathbf{b}\cdot\mathbf{h}) + [(\mathbf{a}\wedge\mathbf{b})\cdot\mathbf{d}](\mathbf{c}\cdot\mathbf{h}).$$

Then replacing $\mathbf{d}$ with $\mathbf{d}\wedge\mathbf{g}$ we get the desired result.

$\square$

2.2 Normal Form

A common strategy for simplifying expressions in computer algebra is to define a normal form. Taking again a polynomial as an example, to simplify $p = (x^2 - 49) - (x + 7)(x - 7)$, we can rewrite it in a monomial basis by expanding all terms and collecting like powers. Then we find $p = 0x^2 + 0x + 0$, which we immediately see is zero. Computer algebra systems use normal forms frequently, but not always. In Maple, normal forms are often obtained using commands containing `normal`. For example, the radnormal function performs normalization of expressions containing algebraic numbers in radical notation, for example, $\sqrt{2}$ and $\sqrt{7+5\sqrt{2}}$, and radnormal simplifies such a number to 0 if and only if it is mathematically equal to 0.

Frequently in Maple an expression is not reduced to a normal form automatically, but only after a specific request. Furthermore, a normal form may not be the simplest form of the expression. For example, the Maple commands `expand` and `combine` can be used to convert a trigonometric expression to different normal forms, but in the example below, neither is a single term, the way the original is. Which form is better will depend upon the application.

$$\begin{aligned}\sin^3(3x) &= 64\sin^3 x\cos^6 x - 48\sin^3 x\cos^4 x + 12\sin^3 x\cos^2 x - \sin^3 x\\ &= -1/4\,\sin(9\,x) + 3/4\,\sin(3\,x)\end{aligned}$$

In line with these considerations, we define a normal form for vector expressions as follows.

The **normal form** of a vector expression is a sum of terms. Each term consists of an underlying coefficient, a scalar part and a vector part. The vector part consists of a empty list, or a single letter or a vector product. The components of a scalar part are empty list, single letters, scalar products and scalar triple products. The order of terms is determined by the **term order** defined in section 3.3.

In our vector algebra package, all input vector expressions will be transferred automatically into their normal forms. A vector expression can be presented in different normal forms. For example, according to formula (2.8), we can present same expression in two different normal forms: $(\mathbf{abc}) * \mathbf{d}$ and $(\mathbf{c} \cdot \mathbf{d}) * (\mathbf{a} \wedge \mathbf{b}) - (\mathbf{b} \cdot \mathbf{d}) * (\mathbf{a} \wedge \mathbf{c}) + (\mathbf{a} \cdot \mathbf{d}) * (\mathbf{b} \wedge \mathbf{c})$. However, by using advanced simplification (see section 3.5), we can get the shortest normal form for a given vector expression.

Chapter 3

Implementations in Aldor

Aldor is a type-complete, strongly-typed, imperative programming language with a two-level object model of categories and domains (similar to the concepts of interfaces and classes in Java). Types and functions are first class entities allowing them to be constructed and manipulated within Aldor programs just like any other values. Aldor is an ideal tool for symbolic mathematical computations. For detailed information, please see *Aldor Compiler User Guide* [1] and *Algebra User Guide and Reference Manual* [2].

3.1 Structure of the Package

In our vector algebra package, we first define a vector space category *VectorSpcCategory* as follows.

```
define VectorSpcCategory(R:Join(ArithmeticType, ExpressionType),
n:MI==3): Category==with
{
  *: (R,%)->%;
  *: (%,R)->%;
  +: (%,%)->%;
  -: (%,%)->%;
  -:     %->%;
  =: (%,%)->Boolean;
  default
  {
    import from R;
    (x:%)-(y:%):%==x+(-1)*y;
    (x:%)*(r:R):%==r*x;
    -(x:%):%==(-1)*x;
  }
}.
```

Here *ArithmeticType* and *ExpressionType* are two categories defined in the algebra library of Aldor and n is the dimension of the space (the default dimension is 3). $R : \mathit{Join}(\mathit{ArithmeticType}, \mathit{ExpressionType})$ means that the coefficient domain R is an ArithmeticType and an ExpressionType. Many algebraic structures, such as *ring* and *field*, are examples of R.

Then, based on VectorSpcCategory, we define a vector algebra category *VectorAlgCategory* as follows.

```
define VectorAlgCategory(R:Join(ArithmeticType,
ExpressionType)):Category == VectorSpcCategory(R) with
{
  vector:      Symbol->%;
  scalarZero:  ()->%;
  vectorZero:  ()->%;
  realVector?: %->Boolean;
  simplify:    (%, advanced:Boolean==false)->%;
  identity?:   (%,%)->Boolean;
  s3p:   (%,%,%)->%;
  *:     (%,%)->%;
  apply: (%,%)->%;
  /\:     (%,%)->%;
  <<:    (TextWriter,%)->TextWriter;
  default
  {
    s3p(x:%,y:%,z:%):%==apply(x^y,z);
  }
}.
```

Here *with* means that besides the exports of VectorSpcCategory,VectorAlgCategory has additional exports (within the braces) which normally are related to component-free vector operations. For example, *apply* is the scalar product, *s3p* is the scalar triple product, and $\wedge$ is the vector product.

Finally, we come to the point of implementing the vector algebra domain *VectorAlg* in details.

```
VectorAlg(R:Join(ArithmeticType,ExpressionType)):
Join(ExpressionType,VectorAlgCategory R)== add
{
  ...
}.
```

Here *Join(ExpressionType, VectorAlgCategory R)* means that the return type of VectorAlg is an ExpressionType and a VectorAlgCategory. The part within the braces after *add* is the implementation details of the domain which composes the rest of this chapter.

3.2 Data Representations

First of all, in order to express vector expressions in their normal forms, we have to choose some suitable data structure. In our vector algebra package, a vector expression is expressed internally as:

`Rep==List Term`, and

`Term==Record (coe:R, sca:List List String, vec:List String)`,

where `coe`, `sca` and `vec` are respectively the underlying coefficient, the scalar part and the vector part of a term. For example, the term $-2(\mathbf{a}\cdot\mathbf{b})(\mathbf{abc}) * (\mathbf{c}\wedge\mathbf{d})$ can be expressed as $[-2, [[``a", ``b"], [``a", ``b", ``c"]], [``c", ``d"]]$. Inspired by Stoutemyer's unsolved problem, we adopt such data representation because we want to separate the scalar part of a term from the vector part of the term.

In the scalar part `sca`, there are three different kinds of lists of strings. List of length 1 stands for a scalar of single symbol, list of length 2 stands for a scalar product, and list of length 3 stands for a scalar triple product. Similiarly, in the vector part `vec`, list of length 0 means that the term is not really a vector, list of length 1 stands for a vector of single symbol, and list of length 2 stands for a vector of two symbols such as $\mathbf{a}\wedge\mathbf{b}$.

In the implementations of transformation rules, we also need another minor data representation for a term. It is necessary when one wants to simplify a vector expression which contains terms with same vector parts.

`mixTerm == Record(scal:scaList, vec2:List String)`, where

`scaList == List scaTerm`, and

`scaTerm == Record(coe2:R, sca2:List List String)`.

Now it is time to discuss the reasonableness of our choice for algebraic formulas (2.1)-(2.11). Some problems about the data representations arise naturally. For example, how to represent the vector product of $\mathbf{a}$ and $\mathbf{b} \wedge \mathbf{c}$, or the vector product of $\mathbf{a} \wedge \mathbf{b}$ and $\mathbf{c} \wedge \mathbf{d}$ since there are only two kinds of vectors in our data representation? Well, formulas (2.5) and (2.6) solve this problem. Formula (2.7) fixes the problem as how to represent the scalar product of $\mathbf{a} \wedge \mathbf{b}$ and $\mathbf{c} \wedge \mathbf{d}$ since there are only three kinds of scalar factors in our data representation.

As we mentioned before, there are two kinds of vectors in our data representation such as $\mathbf{a}$ and $\mathbf{a} \wedge \mathbf{b}$. In order to simplify vector expressions or prove vector identities, it is necessary to establish the relationship between vectors of single symbol and vectors of two symbols. That is the reason why formula (2.8) and formula (2.9) are necessary. When taking scalar product for both sides of formula (2.8) with vector $\mathbf{h}$, we get formula (2.10). When taking scalar product for both sides of formula (2.9) with a vector of single symbol, we get formula (2.11). It is obvious that formulas (2.1) – (2.4) are necessary for simplifying vector expressions. Therefore, all algebraic formulas, formulas (2.1) – (2.11), are necessary for our package.

On the other hand, from the arguments above, we can see that formula (2.1) – (2.11) cover all basic patterns of vector expressions. So, they are sufficient for our package.

3.3 Term Order

As we mentioned above, in order to define a normal form for a vector expression, it is necessary to define a term order for the expression. Unlike polynomials which have a natural way to define term order, We have to choose a term order for our purpose. The term order is defined step by step.

Firstly, we use the natural order for strings.

Secondly, we define the order for lists of strings. Given two lists of strings L_1 and L_2, then

$L_1 < L_2 \Longleftrightarrow \#L_1 < \#L_2$, or $\#L_1 = \#L_2$ and there exists a natural number i such that $L_1(j) = L_2(j)$ for all $0 < j < i$ and $L_1(i) < L_2(i)$,
where $\#L$ denotes the length of list L.

Thirdly, we define the order for lists of lists of strings. Given two lists of lists of strings L_1 and L_2, then

$L_1 < L_2 \Longleftrightarrow \#L_1 < \#L_2$, or $\#L_1 = \#L_2$ and there exists a natural number i such that $L_1(j) = L_2(j)$ for all $0 < j < i$ and $L_1(i) < L_2(i)$.

Finally, we define the order of terms for vector expressions. Given two Terms $T_1 = [coe_1, sca_1, vec_1]$ and $T_2 = [coe_2, sca_2, vec_2]$ (where coe_1 and coe_2 belong to R, sca_1 and sca_2 are lists of lists of strings, vec_1 and vec_2 are lists of strings), then

$T_1 < T_2 \Longleftrightarrow vec_1 < vec_2$, or $vec_1 = vec_2$ and $sca_1 < sca_2$, or $vec_1 = vec_2$ and $sca_1 = sca_2$ and $coe_1 < coe_2$ (if R is an OrderedArithmeticType).

For example, given a vector expression $5(\mathbf{abc})(\mathbf{a}\cdot\mathbf{d}) * (\mathbf{b}\wedge\mathbf{c}) + 2(\mathbf{b}\cdot\mathbf{c}) * (\mathbf{a}\wedge\mathbf{c}) + (\mathbf{acd})(\mathbf{a}\cdot\mathbf{b}) * \mathbf{c}$, after being sorted by the order above, the expression will become $(\mathbf{a}\cdot\mathbf{b})(\mathbf{acd}) * \mathbf{c} + 2(\mathbf{b}\cdot\mathbf{c}) * (\mathbf{a}\wedge\mathbf{c}) + 5(\mathbf{a}\cdot\mathbf{d})(\mathbf{abc}) * (\mathbf{b}\wedge\mathbf{c})$.

3.4 Implementations of Transformation Rules

Now we turn to the implementations of transformation rules. The transformation rules in our package are based on the algebraic formulas (2.1) – (2.11) in chapter 2. The transformation rules can be divided into two types: expansion type and combination type. For example,

$$(\mathbf{abc}) * \mathbf{d} = (\mathbf{d} \cdot \mathbf{a}) * (\mathbf{b} \wedge \mathbf{c}) + (\mathbf{d} \cdot \mathbf{b}) * (\mathbf{c} \wedge \mathbf{a}) + (\mathbf{d} \cdot \mathbf{c}) * (\mathbf{a} \wedge \mathbf{b})$$

is an expansion type rule which expands the term $(\mathbf{abc})*\mathbf{d}$ into 3 terms $(\mathbf{d}\cdot\mathbf{a})*(\mathbf{b}\wedge\mathbf{c})$, $(\mathbf{d} \cdot \mathbf{b}) * (\mathbf{c} \wedge \mathbf{a})$ and $(\mathbf{d} \cdot \mathbf{c}) * (\mathbf{a} \wedge \mathbf{b})$, while

$$(\mathbf{d} \cdot \mathbf{a})(\mathbf{hbc}) + (\mathbf{d} \cdot \mathbf{b})(\mathbf{ahc}) + (\mathbf{d} \cdot \mathbf{c})(\mathbf{abh}) = (\mathbf{d} \cdot \mathbf{h})(\mathbf{abc})$$

is a combination type rule which combines the three terms $(\mathbf{d} \cdot \mathbf{a})(\mathbf{hbc})$, $(\mathbf{d} \cdot \mathbf{b})(\mathbf{ahc})$ and $(\mathbf{d} \cdot \mathbf{c})(\mathbf{abh})$ into a single term $(\mathbf{d} \cdot \mathbf{h})(\mathbf{abc})$. The main skill we use is pattern-matching.

The implementations of expansion type rules are not complicated. For example, the purpose of the following algorithm is to implement the transformation rule $(\mathbf{abc}) * \mathbf{d} = (\mathbf{d} \cdot \mathbf{a}) * (\mathbf{b} \wedge \mathbf{c}) + (\mathbf{d} \cdot \mathbf{b}) * (\mathbf{c} \wedge \mathbf{a}) + (\mathbf{d} \cdot \mathbf{c}) * (\mathbf{a} \wedge \mathbf{b})$. Where *tx.coe*, *tx.sca* and *tx.vec* respectively stand for the underlying coefficient, the scalar part and the vector part of the Term *tx*. The main idea of the algorithm is as follows. Let *tx* range over the Terms of *xx*. If there are lists in *tx* which match the pattern $(\mathbf{abc}) * \mathbf{d}$, then we replace them with the lists which represent $(\mathbf{d} \cdot \mathbf{a}) * (\mathbf{b} \wedge \mathbf{c}) + (\mathbf{d} \cdot \mathbf{b}) * (\mathbf{c} \wedge \mathbf{a}) + (\mathbf{d} \cdot \mathbf{c}) * (\mathbf{a} \wedge \mathbf{b})$.

rule 01

Input: a List Term *xx* (the internal representation of a vector expression).

Output: a List Term *yy*.

```
If $\#(xx) < 3$ or $xx$ isn't a vector then
```

```
    return $yy := xx$
else
    $xx :=$ sort($xx$,termOrder)
    $yy :=$ empty
    for $tx$ in $xx$ do
        if $\#(tx.vec) \neq 1$ or $\#(\text{first}(\text{reverse}(tx.sca))) < 3$ then
            $yy := \text{cons}(tx, yy)$
        else
            $n := \#(tx.sca)$
            $lt :=$ empty
            $rt :=$ empty
            for $i$ from 1 to $n$ do
                if $\#(tx.sca(i)) = 3$ then
                    $lt := tx.sca(i)$
                    $rt := tx.sca - lt$
                    break
                end if
            end do
            $d := \text{first}(tx.vec)$
            $yy := \text{cons}([tx.coe,\text{append}(rt, [[d, lt(1)]]), [lt(2), lt(3)]], yy)$
            $yy := \text{cons}([tx.coe,\text{append}(rt, [[d, lt(2)]]), [lt(3), lt(1)]], yy)$
            $yy := \text{cons}([tx.coe,\text{append}(rt, [[d, lt(3)]]), [lt(1), lt(2)]], yy)$
        end if
    end do
    return $yy$
end if
```

Compared with the implementations of expansion type rules, the implementations of combination type rules are much more complicated. For example, in order to implement the transformation rule $(\mathbf{d}\cdot\mathbf{a})(\mathbf{hbc})+(\mathbf{d}\cdot\mathbf{b})(\mathbf{ahc})+(\mathbf{d}\cdot\mathbf{c})(\mathbf{abh})=(\mathbf{d}\cdot\mathbf{h})(\mathbf{abc})$, first we have to implement a subalgorithm `match?` to check if any given three scaTerms match the pattern of this rule, and then we have to implement another subalgorithm `comb` which combines the matching scaTerms in a mixTerm using this rule. For the definitions of scaTerm, mixTerm and scaList, please see section 3.2 for details. In the following algorithms, p_1, p_2 and p_3 are supposed to be in general case which means that the symbols a, b, c, d and h in the rule are all different, and *distribute* stands for the function to transfer a mixTerm to a List Term.

match?

Input: three scaTerms p_1, p_2 and p_3.

Output: TRUE if they are maching, FALSE otherwise.

```
If not ($|p_1.coe2| = |p_2.coe2| = |p_3.coe2|$) then
    return FALSE
else
    $l_1 := p_1.sca2$
    $l_2 := p_2.sca2$
    $l_3 := p_3.sca2$
    $sg$ := the intersection of $l_1$, $l_2$ and $l_3$
    $s_1 := l_1 - sg$
    $s_2 := l_2 - sg$
    $s_3 := l_3 - sg$
    if not($\#s_1 = 2$ and $\#s_2 = 2$ and $\#s_3 = 2$ and $\#s_1(1) * \#s_1(2) = 6$
    and $\#s_2(1) * \#s_2(2) = 6$ and $\#s_3(1) * \#s_3(2) = 6$) then
    return FALSE
else
    $s_1$ := sort($s_1$,listOrder)
    $s_2$ := sort($s_2$,listOrder)
    $s_3$ := sort($s_3$,listOrder)
    $ut_2$ := the union of $s_1(1)$, $s_2(1)$ and $s_3(1)$
    $ut_3$ := the union of $s_1(2)$, $s_2(2)$ and $s_3(2)$
    $it_2$ := the intersection of $s_1(1)$, $s_2(1)$ and $s_3(1)$
    $it_3$ := the intersection of $s_1(2)$, $s_2(2)$ and $s_3(2)$
    if not ($\#ut_2 = 4$ and $\#ut_3 = 4$ and $\#it_2 = 1$ and $\#it_3 = 1$
    and $(ut_2 - it_2) = (ut_3 - it_3)$ and $\#(ut_2 - it_2) = 3$) then
        return FALSE
    else
        $d := it_2(1)$
        $h := it_3(1)$
        $a := (ut_2 - it_2)(1)$
        $b := (ut_2 - it_2)(2)$
        $c := (ut_2 - it_2)(3)$
        $coe := p_1.coe2$
        $tp_1$ := [$coe$,append($[[d, a], [h, b, c]], sg$), [ ]]
        $tp_2$ := [$coe$,append($[[d, b], [a, h, c]], sg$), [ ]]
        $tp_3$ := [$coe$,append($[[d, c], [a, b, h]], sg$), [ ]]
        if $[tp_1, tp_2, tp_3] = \pm[[p_1, p_2, p_3], [\,]]$ then
            return TRUE
        else
            return FALSE
            end if
         end if
      end if
   end if
```

comb

Input: a mixTerm p (the length of $p.scal$ is 3).

Output: a List Term

```
$l_1 := (p.scal(1)).sca2$
$l_2 := (p.scal(2)).sca2$
$l_3 := (p.scal(3)).sca2$
use the same method as in **match?** to find out $sg, d, h, a, b, c$
$coe := (p.scal(1)).coe2$
$tp_1 := [coe, \text{append}([[d,a],[h,b,c]], sg), p.vec2]$
$tp_2 := [coe, \text{append}([[d,b],[a,h,c]], sg), p.vec2]$
$tp_3 := [coe, \text{append}([[d,c],[a,b,h]], sg), p.vec2]$
if $[tp_1, tp_2, tp_3] = p$ then
   return $[[coe, \text{append}([[d,h],[a,b,c]], sg), p.vec2]]$
else
   return $[[coe, \text{append}([[d,h],[a,c,b]], sg), p.vec2]]$
end if
```

Now it is time to implement the transformation rule $(\mathbf{d} \cdot \mathbf{a})(\mathbf{hbc}) + (\mathbf{d} \cdot \mathbf{b})(\mathbf{ahc}) + (\mathbf{d} \cdot \mathbf{c})(\mathbf{abh}) = (\mathbf{d} \cdot \mathbf{h})(\mathbf{abc})$ based on the two subalgorithms above. The main idea of this algorithm is as follows. Firstly, we change the data representation xx from List Term to List mixTerm. Secondly, let tx range over xx, and let L be the scaList part of tx. If the length of L is less than 3, then leave it for output since we cannot apply the transformation rule to L. Otherwise, again let L denote the collection of scaTerms in L in which there are lists of length 2 and length 3 in their *sca2* part, and leave other scaTerms for output. Thirdly, in a while-loop, check if any given three scaTerms match the rule using the subalgorithm `match?`. If there is no such three scaTerms or the length of L is less than 3, then break the loop. Otherwise, combine the three scaTerms using the subalgorithm `comb` and update L. Finally, output the remaining L.

rule03

Input: a List Term xx.

Output: a List Term yy.

```
If $\#xx < 3$ then
  return $yy := xx$
else
  $xx :=$ sort($xx$,termOrder)
  $xx :=$ combine those Terms of $xx$ which have same vector parts
  $yy :=$ empty
  for $tx$ in $xx$ repeat
    $L := tx.scal$
    $n := \#L$
    if $n < 3$ then
      $yy :=$ append($yy$, distribute($tx$))
    else
      for $i$ from 1 to $n$ do
        if $L(i).sca2$ doesn't contain lists of length 2 and 3 then
          $yy :=$ append($yy$,distribute($[[L(i)], tx.vec2]$))
```

```
            $L := L - L(i)$
          end if
        end do
        $n := \#L$
        $ttp :=$ empty
        while $n > 2$ loop
          for $1 \le i < j < k \le n$ do
            if match?$(L(i), L(j), L(k))$ then
              $ttp := [L(i), L(j), L(k)]$
              $L := L - ttp$
              break do
            end if
          end do
          if empty?$(ttp)$ then
            break loop
          else
            $yy :=$ append$(yy,$comb$([ttp, tx.vec2]))$
            $ttp :=$ empty
            $n := \#L$
          end if
        end loop
        if not(empty?$(L)$) then
          $yy :=$ append$(yy,$distribute$([L, tx.vec2]))$
        end if
      end if
    end repeat
    return $yy$
end if
```

3.5 Simplifications and Proofs

First, we would like to dicuss simplifications of vector expressions. There are two levels of simplifications. One is basic level which performs basic simplifications using transformation rules based on formulas (2.1) – (2.4). Another is advanced level which performs advanced simplifications using transformation rules based on formulas (2.1) – (2.4) and formulas (2.8) – (2.11).

The main idea for simplification is that we first use those transformation rules based on formulas (2.1) – (2.4) to perform basic simplifications, and make the vector expression in ascending order which is very convenient for combining like terms. If the number of terms ≥ 3 and the *advanced* flag is true which means advanced simplification then we will apply transformation rules based on formulas (2.8) – (2.11) to the vector expression one by one within a while-loop. After applying a transformation rule and performing basic simplification, if the resulting expression is longer than the original one which means the uselessness of the application of the rule, then discard the resulting expression. If at any time the number of terms < 3, return the expression.

In the following algorithm, if x is an element of the VectorAlg domain, then $rep(x)$ denotes its data representation. On the other hand, if xx is the data representation of an element x in the domain VectorAlg, then $per(xx)$ is x itself. *Rule(k)* stands for transformation rules based on formula(k), and *RuleSet* stands for all transformation rules based on formulas (2.8) – (2.11).

simplify

Input: an element x of the VectorAlg domain; a boolean value variable *advanced.*

Output: an element of the VectorAlg domain.

```
$xx := rep(x)$
if empty?$(xx)$ then
   return $x$
else
   $lo :=$ empty
   for $tx$ in $xx$ repeat
      if $tx.vec(1) = tx.vec(2)$ then
         $lo := lo$
      else
         make $tx.vec$ in ascending order
         if empty?$(tx.sca)$ then
            $lo :=$ cons$(tx, lo)$
         else if *rule(2.4)* fires in $tx.sca$ then
            $lo := lo$
         else
            make every factor in $tx.sca$ in ascending order
            make $tx.sca$ in ascending order
            $lo :=$ cons$(tx, lo)$
         end if
      end if
   end repeat
   if empty?$(lo)$ then
      return $per(lo)$
```

```
    else
        $lo$ := sort($lo$, termOrder)
        $lo$ := combine like terms of $lo$
        if not ($advanced$ and $\#lo > 2$) then
            return $per(lo)$
        else
            $stp$ := empty
            $L$ := empty
            while $per(lo) \neq per(stp)$ loop
                $stp := lo$
                for $rule$ in $RuleSet$ do
                    $L$ := simplify($per(rule(lo))$)
                    if $\#rep(L) < \#lo$ then
                        $lo := rep(L)$
                    end if
                    if $\#lo < 3$ then
                        break and return $per(lo)$
                    end if
                end do
            end loop
            return $per(lo)$
        end if
    end if
end if
```

We want to make two points clear. First, we ask the number of terms in a vector expression to be at least 3 because otherwise it is useless to use transformation rules based on formulas (2.8) – (2.11). Second, we choose $per(lo) = per(stp)$ as the terminating condition of the loop because after each loop, $\#lo < \#stp$ and $\#stp$ is finite, so the loop should stop in finite steps.

Then, we would like to discuss the algorithm for proving vector identities. The basic principal is similar to that of simplification. For the sake of simplicity, in practice, we usually use $x = y$ instead of $identity?(x, y)$ to check if two elements x, y in the VectorAlg domain are identical.

identity?

Input: two elements x and y in VectorAlg domain.

Output: a boolean value.

```
$z :=$ simplify$(x - y)$
if $z = 0$ then
    return TRUE
else if $\#rep(z) \leq 2$ then
    return FALSE
else
    $lo := rep(z)$
    $stp :=$ empty
    while $per(lo) \neq per(stp)$ loop
        $stp := lo$
        for $rule$ in $RuleSet$ do
            $L :=$ simplify$(per(rule(lo))$
            if $L = 0$ then
                break and return TRUE
            else if $\#rep(L) < \#lo$ then
                $lo := rep(L)$
            end if
        end do
    end loop
    return FALSE
end if
```

3.6 Vector Algebra Operations

Finally, we talk about general vector algebra operations. For addition of two elements x and y in VectorAlg domain, simply merge $rep(x)$ and $rep(y)$ together, then perform basic simplification. For scalar multiplication of r and a domain element x, replace coe with $r * coe$ for every Term in $rep(x)$, then perform basic simplification. For scalar product of two domain elements x and y, simply use distribution law and formula (2.7), then perform basic simplification. For vector product of two domain elements x and y, simply use distribution law and formulas (2.5) and (2.6), then perform basic simplification .

Chapter 4

Examples

We use Aldor Interpreter to test our package. After compiling our source file into a platform-independent object file *vectorAlg.ao*, we open the Aldor Interpreter using command: *aldor -gloop*. Then we use following commands to do initialization:

```
%1 >> #include "algebra"
%2 >> #include "aldorinterp"
%3 >> #library aaa "vectorAlg.ao"
%4 >> macro M==MachineInteger
%5 >> import from aaa, M, String, Symbol, VectorAlg M
```

Now we declare vectors **a**, **b**, **c**, **d**, **e**, **f**, **g** and **h** as follows.

```
%6 >> a:=vector(-"a")
%7 >> b:=vector(-"b")
%8 >> c:=vector(-"c")
%9 >> d:=vector(-"d")
%10 >> e:=vector(-"e")
%11 >> f:=vector(-"f")
```

```
%12 >> g:=vector(-"g")

%13 >> h:=vector(-"h")
```

At this point, we are ready to test examples using our package. Firstly, we would like to test Stoutemyer's unsolved problem in chapter 1.

```
%14 >> ((a^b)^(b^c)).(c^a)-(a.(b^c))*(a.(b^c))
0 @ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 40 msec
```

From the experiment above, we can see that our package is very efficient. It only takes 40 milliseconds to get the result. Now, we will test some more examples also from Stoutemyer [18]:

- $(\mathbf{a}-\mathbf{d})\wedge(\mathbf{b}-\mathbf{c})+(\mathbf{b}-\mathbf{d})\wedge(\mathbf{c}-\mathbf{a})+(\mathbf{c}-\mathbf{d})\wedge(\mathbf{a}-\mathbf{b})-2*(\mathbf{a}\wedge\mathbf{b}+\mathbf{b}\wedge\mathbf{c}+\mathbf{c}\wedge\mathbf{a}) = \mathbf{0}$.
- $(\mathbf{b}\wedge\mathbf{c})\wedge(\mathbf{a}\wedge\mathbf{d})+(\mathbf{c}\wedge\mathbf{a})\wedge(\mathbf{b}\wedge\mathbf{d})+(\mathbf{a}\wedge\mathbf{b})\wedge(\mathbf{c}\wedge\mathbf{d})+2*(\mathbf{abc})*\mathbf{d} = \mathbf{0}$.

```
%15 >> (a-d)^(b-c)+(b-d)^(c-a)+(c-d)^(a-b)-2*(a^b+b^c+c^a)
0 @ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 40 msec
%16 >> v1:=(b^c)^(a^d)+(c^a)^(b^d)+(a^b)^(c^d)+2*s3p(a,b,c)*d
(bcd)*a-(acd)*b+(abd)*c-(abc)*d @ VectorAlg(MachineInteger)
                                  Comp: 10 msec, Interp: 30 msec
%17 >> simplify(v1,true)
0 @ VectorAlg(MachineInteger)
                                  Comp: 10 msec, Interp: 30 msec
```

The following examples come from Cunningham [5]:

- $\mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c}) + \mathbf{b} \wedge (\mathbf{c} \wedge \mathbf{a}) + \mathbf{c} \wedge (\mathbf{a} \wedge \mathbf{b}) = \mathbf{0}$.
- $\mathbf{a}\wedge(\mathbf{b}\wedge(\mathbf{c}\wedge\mathbf{d}))+\mathbf{b}\wedge(\mathbf{c}\wedge(\mathbf{d}\wedge\mathbf{a}))+\mathbf{c}\wedge(\mathbf{d}\wedge(\mathbf{a}\wedge\mathbf{b}))+\mathbf{d}\wedge(\mathbf{a}\wedge(\mathbf{b}\wedge\mathbf{c})) = (\mathbf{a}\wedge\mathbf{c})\wedge(\mathbf{b}\wedge\mathbf{d})$.

```
%18 >> a^(b^c)+b^(c^a)+c^(a^b)
0 @ VectorAlg(MachineInteger)
                                          Comp: 0 msec, Interp: 20 msec
%19 >> a^(b^(c^d))+b^(c^(d^a))+c^(d^(a^b))+d^(a^(b^c))=(a^c)^(b^d)
T @ Boolean
                                          Comp: 0 msec, Interp: 60 msec
```

The following examples come from Patterson [15]

- $(\mathbf{b} \wedge \mathbf{c}) \cdot (\mathbf{a} \wedge \mathbf{d}) + (\mathbf{c} \wedge \mathbf{a}) \cdot (\mathbf{b} \wedge \mathbf{d}) + (\mathbf{a} \wedge \mathbf{b}) \cdot (\mathbf{c} \wedge \mathbf{d}) = 0$.
- $((\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c}) \wedge \mathbf{d} + ((\mathbf{b} \wedge \mathbf{a}) \wedge \mathbf{d}) \wedge \mathbf{c} + ((\mathbf{c} \wedge \mathbf{d}) \wedge \mathbf{a}) \wedge \mathbf{b} + ((\mathbf{d} \wedge \mathbf{c}) \wedge \mathbf{b}) \wedge \mathbf{a} = \mathbf{0}$.

```
%20 >> (b^c).(a^d)+(c^a).(b^d)+(a^b).(c^d)
0 @ VectorAlg(MachineInteger)
                                          Comp: 0 msec, Interp: 30 msec
%21 >> ((a^b)^c)^d+((b^a)^d)^c+((c^d)^a)^b+((d^c)^b)^a
0 @ VectorAlg(MachineInteger)
                                          Comp: 10 msec, Interp: 30 msec
```

Actually, it is difficult for us to find examples in existing literature complicated enough to challenge our system. So, we create some examples by ourselves.

```
%22 >> p:=s3p(a,b,c)*d
(abc)*d @ VectorAlg(MachineInteger)
                                          Comp: 0 msec, Interp: 10 msec
```

```
%23 >> q:=(d.a)*(b^c)+(d.b)*(c^a)+(d.c)*(a^b)

(c.d)*(a^b)-(b.d)*(a^c)+(a.d)*(b^c) @ VectorAlg(MachineInteger)

                                  Comp: 0 msec, Interp: 20 msec

%24 >> m:=e^(f^(g^h))+f^(g^(h^e))+g^(h^(e^f))+h^(e^(f^g))

(g.h)*(e^f)-(f.g)*(e^h)+(e.h)*(f^g)+(e.f)*(g^h) @

VectorAlg(MachineInteger)

                                  Comp: 10 msec, Interp: 30 msec

%25 >> n:=(e^g)^(f^h)

(egh)*f+(efg)*h @ VectorAlg(MachineInteger)

                                  Comp: 0 msec, Interp: 10 msec

%26 >> p=q

T @ Boolean

                                  Comp: 10 msec, Interp: 20 msec

%27 >> m=n

T @ Boolean

                                  Comp: 0 msec, Interp: 40 msec

%28 >> r1:=p^n

(abc)(egh)*(d^f)+(abc)(efg)*(d^h) @ VectorAlg(MachineInteger)

                                  Comp: 10 msec, Interp: 10 msec

%29 >> r2:=q^m

-(a.d)(f.g)(bch)*e+(a.d)(g.h)(bcf)*e+(b.d)(f.g)(ach)*e

-(b.d)(g.h)(acf)*e-(c.d)(f.g)(abh)*e+(c.d)(g.h)(abf)*e

+(a.d)(e.h)(bcg)*f-(a.d)(g.h)(bce)*f-(b.d)(e.h)(acg)*f

+(b.d)(g.h)(ace)*f+(c.d)(e.h)(abg)*f-(c.d)(g.h)(abe)*f

+(a.d)(e.f)(bch)*g-(a.d)(e.h)(bcf)*g-(b.d)(e.f)(ach)*g

+(b.d)(e.h)(acf)*g+(c.d)(e.f)(abh)*g-(c.d)(e.h)(abf)*g
```

```
-(a.d)(e.f)(bcg)*h+(a.d)(f.g)(bce)*h+(b.d)(e.f)(acg)*h
-(b.d)(f.g)(ace)*h-(c.d)(e.f)(abg)*h+(c.d)(f.g)(abe)*h
@ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 80 msec
%30 >> r1=r2
T @ Boolean
                                  Comp: 0 msec, Interp: 2730 msec
%31 >> r:=r1+a^(b^c)-r2
(a.c)*b-(a.b)*c+(a.d)(f.g)(bch)*e-(a.d)(g.h)(bcf)*e
-(b.d)(f.g)(ach)*e+(b.d)(g.h)(acf)*e+(c.d)(f.g)(abh)*e
-(c.d)(g.h)(abf)*e-(a.d)(e.h)(bcg)*f+(a.d)(g.h)(bce)*f
+(b.d)(e.h)(acg)*f-(b.d)(g.h)(ace)*f-(c.d)(e.h)(abg)*f
+(c.d)(g.h)(abe)*f-(a.d)(e.f)(bch)*g+(a.d)(e.h)(bcf)*g
+(b.d)(e.f)(ach)*g-(b.d)(e.h)(acf)*g-(c.d)(e.f)(abh)*g
+(c.d)(e.h)(abf)*g+(a.d)(e.f)(bcg)*h-(a.d)(f.g)(bce)*h
-(b.d)(e.f)(acg)*h+(b.d)(f.g)(ace)*h+(c.d)(e.f)(abg)*h
-(c.d)(f.g)(abe)*h+(abc)(egh)*(d^f)+(abc)(efg)*(d^h) @
VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 110 msec
%32 >> simplify(r,true)
(a.c)*b-(a.b)*c @ VectorAlg(MachineInteger)
                                  Comp: 10 msec, Interp: 2670 msec
%33 >> r3:=m^q
-(b.d)(e.f)(cgh)*a-(b.d)(e.h)(cfg)*a+(b.d)(f.g)(ceh)*a
-(b.d)(g.h)(cef)*a+(c.d)(e.f)(bgh)*a+(c.d)(e.h)(bfg)*a
-(c.d)(f.g)(beh)*a+(c.d)(g.h)(bef)*a+(a.d)(e.f)(cgh)*b
```

```
+(a.d)(e.h)(cfg)*b-(a.d)(f.g)(ceh)*b+(a.d)(g.h)(cef)*b
-(c.d)(e.f)(agh)*b-(c.d)(e.h)(afg)*b+(c.d)(f.g)(aeh)*b
-(c.d)(g.h)(aef)*b-(a.d)(e.f)(bgh)*c-(a.d)(e.h)(bfg)*c
+(a.d)(f.g)(beh)*c-(a.d)(g.h)(bef)*c+(b.d)(e.f)(agh)*c
+(b.d)(e.h)(afg)*c-(b.d)(f.g)(aeh)*c+(b.d)(g.h)(aef)*c
@ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 90 msec
%34 >> r4:=n^q
-(b.d)(c.f)(egh)*a-(b.d)(c.h)(efg)*a+(b.f)(c.d)(egh)*a
+(b.h)(c.d)(efg)*a+(a.d)(c.f)(egh)*b+(a.d)(c.h)(efg)*b
-(a.f)(c.d)(egh)*b-(a.h)(c.d)(efg)*b-(a.d)(b.f)(egh)*c
-(a.d)(b.h)(efg)*c+(a.f)(b.d)(egh)*c+(a.h)(b.d)(efg)*c
@ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 40 msec
%35 >> r3=r4
T @ Boolean
                                  Comp: 0 msec, Interp: 4280 msec
%36 >> r34:=r3+b^a-r4
(b.d)(c.f)(egh)*a+(b.d)(c.h)(efg)*a-(b.d)(e.f)(cgh)*a
-(b.d)(e.h)(cfg)*a+(b.d)(f.g)(ceh)*a-(b.d)(g.h)(cef)*a
-(b.f)(c.d)(egh)*a-(b.h)(c.d)(efg)*a+(c.d)(e.f)(bgh)*a
+(c.d)(e.h)(bfg)*a-(c.d)(f.g)(beh)*a+(c.d)(g.h)(bef)*a
-(a.d)(c.f)(egh)*b-(a.d)(c.h)(efg)*b+(a.d)(e.f)(cgh)*b
+(a.d)(e.h)(cfg)*b-(a.d)(f.g)(ceh)*b+(a.d)(g.h)(cef)*b
+(a.f)(c.d)(egh)*b+(a.h)(c.d)(efg)*b-(c.d)(e.f)(agh)*b
-(c.d)(e.h)(afg)*b+(c.d)(f.g)(aeh)*b-(c.d)(g.h)(aef)*b
```

```
+(a.d)(b.f)(egh)*c+(a.d)(b.h)(efg)*c-(a.d)(e.f)(bgh)*c
-(a.d)(e.h)(bfg)*c+(a.d)(f.g)(beh)*c-(a.d)(g.h)(bef)*c
-(a.f)(b.d)(egh)*c-(a.h)(b.d)(efg)*c+(b.d)(e.f)(agh)*c
+(b.d)(e.h)(afg)*c-(b.d)(f.g)(aeh)*c+(b.d)(g.h)(aef)*c -(a^b)
@VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 170 msec
%37 >> simplify(r34,true)
-(a^b) @ VectorAlg(MachineInteger)
                                  Comp: 0 msec, Interp: 4220 msec
```

Appendix A

Source Code

```
#include "algebra" #include "aldorio"

macro {
  MI==MachineInteger;
  TREE==ExpressionTree;
  TEXT==TextWriter;
}

-- The categories and domains for the package --

define VectorSpcCategory(R:Join(ArithmeticType,
ExpressionType),n:MI==3):Category==with
{
  *: (R,%)->%;
  *: (%,R)->%;
  +: (%,%)->%;
  -: (%,%)->%;
  -:     %->%;
  =: (%,%)->Boolean;
  default
  {
    import from R;
    (x:%)-(y:%):%==x+(-1)*y;
    (x:%)*(r:R):%==r*x;
    -(x:%):%==(-1)*x;
  }
}

define VectorAlgCategory(R:Join(ArithmeticType,
ExpressionType)):Category == VectorSpcCategory(R) with
{
  vector:      Symbol->%;
  scalarZero:  ()->%;
  vectorZero:  ()->%;
  realVector?: %->Boolean;
  simplify:    (%,flag:Boolean==false)->%;
  identity?:   (%,%)->Boolean;
  s3p:   (%,%,%)->%;
  *:     (%,%)->%;
  apply: (%,%)->%;
  ^:     (%,%)->%;
  <<:    (TextWriter,%)->TextWriter;
  default
  {
    s3p(x:%,y:%,z:%):%==apply(x^y,z);
  }
}

VectorAlg(R:Join(ArithmeticType,ExpressionType)):
Join(ExpressionType,VectorAlgCategory R)== add
{
  Term ==Record(coe:R, sca:List List String, vec:List String);
  Rep  ==List Term;
  scaTerm ==Record(coe2:R, sca2:List List String);
  scaList ==List scaTerm;
  mixTerm ==Record(scal:scaList,vec2:List String);
  import from MI, R, Term, Rep, TEXT, TREE, String, Boolean;
  import from scaTerm, scaList, mixTerm;

-- The local functions for the domain "VectorAlg" --

  local szero?(xx:Term): Boolean=={xx.coe=0 and empty? xx.vec;}

  local vzero?(xx:Term): Boolean==
  {
    s:String:="0";
    xx.vec=[s] or (xx.coe=0 and ~empty? xx.vec);
  }

  local append(l:List List String, r:List List String): List List String==
  {
    tp:List List String:=copy(l);
    append!(tp,r);
  }

  local append(l:List String, r:List String): List String==
  {
    tp:List String:=copy(l);
    append!(tp,r);
  }

  local append(l:Rep, r:Rep): Rep==
  {
    tp:Rep:=copy(l);
    append!(tp,r);
  }

  local append(l:scaList, r:scaList): scaList==
  {
    tp:scaList:=copy(l);
    append!(tp,r);
  }

  local simplify0(xx:Rep):Rep==
  {
    l: List Term:=empty;
    tf: MI:=0;
    for tx in xx repeat{
      szero? tx =>l:=l;
      vzero? tx => tf:=1;
      l:=cons(tx,l);
    }
    empty? l and tf=0 =>[[0,[],[]]];
    empty? l and tf=1 =>[[0,[],["0"]]];
    reverse! l;
  }

-- Vector algebra operations --

  vector(a:Symbol):%=={per [ [1,[],[name a]] ];}

  scalarZero():%=={per [ [0,[],[]] ];}

  vectorZero():%=={per [ [0,[],["0"]] ];}

  realVector?(x:%):Boolean==
  {
    import from Boolean;
    xx:=rep x;
    empty? xx => false;
    t:Boolean:=true;
    for tx in xx repeat
    {
      if empty? tx.vec then
      {
        t:=false;
        break;
      }
    }
    t;
  }

  (r:R) * (x:%): %==
  {
    xx:=rep x;
    xx:= [[r*p.coe,p.sca,p.vec] for p in xx];
    per simplify0(xx);
  }

  (s:%) * (x:%): %==
  {
    ss:=rep s;
    xx:=rep x;
    l:List Term:=empty;
    for ts in ss repeat{
      assert(empty? ts.vec);
      for tx in xx repeat{
        m:=[ts.coe*tx.coe, append(ts.sca,tx.sca), tx.vec]@Term;
        l:=cons(m,l);
      }
    }
    per simplify0 reverse! l;
  }

  (x:%) + (y:%): %==
  {
    per append(rep x, rep y);
  }

  apply(x:%, y:%): %==
  {
    ~realVector? x =>
    {stdout<<"The first argument is not really vector!"<<newline;
     per [[0,[],[]]];}
    ~realVector? y =>
    {stdout<<"The second argument is not really vector!"<<newline;
     per [[0,[],[]]];}
    xx:=rep x;
    yy:=rep y;
    l:List Term:=empty;
    for tx in xx repeat{
      n1:MI:=#tx.vec;
      for ty in yy repeat{
        n2:MI:=#ty.vec;
        if n1=1 and n2=1 then
        {
          m:=[tx.coe*ty.coe,cons([tx.vec(1),ty.vec(1)],
              append(tx.sca,ty.sca)),[]]@Term;
          l:=cons(m,l);
        }
        else if n1=1 and n2=2 then
        {
          m:=[tx.coe*ty.coe,cons([tx.vec(1),ty.vec(1),ty.vec(2)],
              append(tx.sca,ty.sca)),[]]@Term;
          l:=cons(m,l);
        }
        else if n1=2 and n2=1 then
        {
          m:=[tx.coe*ty.coe,cons([tx.vec(1),tx.vec(2),ty.vec(1)],
              append(tx.sca,ty.sca)),[]]@Term;
          l:=cons(m,l);
        }
        else if n1=2 and n2=2 then
        {
          tp1:List List String:=[[tx.vec(1),ty.vec(1)],[tx.vec(2),ty.vec(2)]];
          tp2:List List String:=[[tx.vec(1),ty.vec(2)],[tx.vec(2),ty.vec(1)]];
          m1:=[tx.coe*ty.coe,append(tp1,append(tx.sca,ty.sca)),[]]@Term;
          m2:=[(-1)*tx.coe*ty.coe,append(tp2,append(tx.sca,ty.sca)),[]]@Term;
          l:=cons(m2,cons(m1,l));
        }
      }
    }
    per simplify0 reverse! l;
  }

  (x:%) ^ (y:%): %==
  {
    ~realVector? x =>
    {stdout<<"The first argument is not really vector!"<<newline;
     per [[0,[],[]]];}
    ~realVector? y =>
    {stdout<<"The second argument is not really vector!"<<newline;
```

```
 per [[0,[],[]]];}
xx:=rep x;
yy:=rep y;
l:List Term:=empty;
for tx in xx repeat{
  n1:MI:=#tx.vec;
  for ty in yy repeat{
    n2:MI:=#ty.vec;
    if n1=1 and n2=1 then
    {
      m:=[tx.coe*ty.coe,append(tx.sca,ty.sca),append(tx.vec,ty.vec)]@Term;
      l:=cons(m,l);
    }
    else if n1=1 and n2=2 then
    {
      m1:=[tx.coe*ty.coe,cons([tx.vec(1),ty.vec(2)],
           append(tx.sca,ty.sca)),[ty.vec(1)]]@Term;
      m2:=[(-1)*tx.coe*ty.coe,cons([tx.vec(1),ty.vec(1)],
           append(tx.sca,ty.sca)),[ty.vec(2)]]@Term;
      l:=cons(m2,cons(m1,l));
    }
    else if n1=2 and n2=1 then
    {
      m1:=[tx.coe*ty.coe,cons([tx.vec(1),ty.vec(1)],
           append(tx.sca,ty.sca)),[tx.vec(2)]]@Term;
      m2:=[(-1)*tx.coe*ty.coe,cons([tx.vec(2),ty.vec(1)],
           append(tx.sca,ty.sca)),[tx.vec(1)]]@Term;
      l:=cons(m2,cons(m1,l));
    }
    else if n1=2 and n2=2 then
    {
      m1:=[tx.coe*ty.coe,cons([tx.vec(1),tx.vec(2),ty.vec(2)],
                append(tx.sca,ty.sca)),[ty.vec(1)]]@Term;
           m2:=[(-1)*tx.coe*ty.coe,cons([tx.vec(1),tx.vec(2),ty.vec(1)],
                append(tx.sca,ty.sca)),[ty.vec(2)]]@Term;
           l:=cons(m2,cons(m1,l));
         }
       }
     }
     per simplify0 reverse! l;
   }

-- I/O functions for the domain --

   local writeTerm(port:TEXT, t:Term):()==
   {
     szero? t => port<<0$R;
     vzero? t => port<<"0";
     if t.coe~=1 then port<<t.coe;
     tp:=t.sca;
     while ~empty? tp repeat
     {
       t1:=first tp;
       tp:=rest tp;
       if #t1=1 then port<<first t1;
       else if #t1=2 then
       {
         port<<"("<<t1(1)<<"."<<t1(2)<<")";
       }
       else
       {
         port<<"(";
         for s in t1 repeat port<<s;
         port<<")";
```

```
    }
  }
  if ~empty? t.vec then
  {
    if (t.coe~=1 or ~empty? t.sca) then port<<"*";
    #t.vec=1 => port<<first t.vec;
    port<<"("<<t.vec(1)<<"^"<<t.vec(2)<<")";
  }
}

(port:TEXT)<<(x:%):TEXT==
{
  xx:=rep x;
  empty? xx => port<<0$R;

  if R has OrderedArithmeticType then
  {
    import from R pretend OrderedArithmeticType;
    tp:=first xx;
    if tp.coe<0 then
    {
      port<<"-";
      writeTerm(port,[(-1)*tp.coe, tp.sca, tp.vec]@Term);
    }
    else
    { writeTerm(port,tp);}
    xx:=rest xx;
    while ~empty? xx repeat
    {
      tx:=first xx;
      xx:=rest xx;
      if tx.coe<0 then
      {
        port<<"-";
        writeTerm(port,[(-1)*tx.coe, tx.sca, tx.vec]@Term);
      }
      else
      {
        port<<"+";
        writeTerm(port,tx);
      }
    }
  }
  else
  {
    writeTerm(port,first xx);
    xx:=rest xx;
    while ~empty? xx repeat
    {
      tx:=first xx;
      xx:=rest xx;
      port<<"+";
      writeTerm(port,tx);
    }
  }
  port;
}

(x:%) = (y:%): Boolean==
{
  identity?(x,y);
}

extree(x:%):TREE==
  {
    extree "0";
  }

-- function for checking vector identities --

  identity?(x:%, y:%): Boolean==
  {
    z:=simplify(x-y);
    zz:=copy rep z;
    #zz=1 and (vzero?(first(zz)) or szero?(first(zz))) => true;
    l:List Term:=empty;
    if #zz > 2 then
    {
      tt:Boolean:=false;
      stp:List Term:=empty;
      while ~same?(per zz, per stp) repeat
      {
        stp:=copy zz;
        l:=rule01(zz);
        tpz:=simplify per l;
        tpzz:=rep tpz;
        #tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
          => {tt:=true; break;}
        if #tpzz<#zz then zz:=tpzz;
        l:=rule01(zz,true);
        tpz:=simplify per l;
        tpzz:=rep tpz;
        #tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
          => {tt:=true; break;}
        if #tpzz<#zz then zz:=tpzz;
        l:=rule11(zz);
```

```
tpz:=simplify per l;
tpzz:=rep tpz;
#tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
  => {tt:=true; break;}
if #tpzz<#zz then zz:=tpzz;
l:=rule02(zz);
tpz:=simplify per l;
tpzz:=rep tpz;
#tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
  => {tt:=true; break;}
if #tpzz<#zz then zz:=tpzz;
l:=rule02(zz,true);
tpz:=simplify per l;
tpzz:=rep tpz;
#tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
  => {tt:=true; break;}
if #tpzz<#zz then zz:=tpzz;
l:=rule03(zz);
tpz:=simplify per l;
tpzz:=rep tpz;
#tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
  => {tt:=true; break;}
if #tpzz<#zz then zz:=tpzz;
l:=rule04(zz);
tpz:=simplify per l;
tpzz:=rep tpz;
#tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
  => {tt:=true; break;}
if #tpzz<#zz then zz:=tpzz;
l:=rule04(zz,true);
tpz:=simplify per l;
tpzz:=rep tpz;
```

```
      #tpzz=1 and (vzero?(first(tpzz)) or szero?(first(tpzz)))
        => {tt:=true; break;}
      if #tpzz<#zz then zz:=tpzz;
    }
    tt => tt;
    stdout<<per zz<<" = "<<"0 ??? "<<newline<<newline;
    tt;
  }
  else
  {
    stdout<<z<<" = "<<"0 ??? "<<newline<<newline;
    false;
  }
}

-- The Orders for the package --

local listOrder?(l1:List String, l2:List String): Boolean==
{
  empty? l1 => true;
  empty? l2 => false;
  n1:=#l1;
  n2:=#l2;
  n1<n2 =>true;
  n1>n2 =>false;
  tt:Boolean:=true;
  for i in 1..n1 repeat
  {
    l1(i)<l2(i) => { tt:=true; break; }
    l1(i)>l2(i) => { tt:=false; break; }
  }
  tt;
```

```
}

local listEq?(l1:List String, l2:List String): Boolean==
{
  empty? l1 => empty? l2;
  n1:=#l1;
  n2:=#l2;
  n1~=n2 => false;
  tt:Boolean:=true;
  for i in 1..n1 repeat
  {
    l1(i)~=l2(i) => {tt:=false; break; }
  }
  tt;
}

local dblListOrder?(l1:List List String, l2:List List String):
Boolean==
{
  empty? l1 => true;
  empty? l2 => false;
  n1:=#l1;
  n2:=#l2;
  n1<n2 =>true;
  n1>n2 =>false;
  tt:Boolean:=true;
  for i in 1..n1 repeat
  {
    listOrder?(l1(i),l2(i)) and ~listEq?(l1(i),l2(i))
      =>{ tt:=true; break; }
    ~listOrder?(l1(i),l2(i)) => { tt:=false; break; }
  }
  tt;
}

local dblListEq?(l1:List List String, l2:List List String): Boolean==
{
  empty? l1 => empty? l2;
  n1:=#l1;
  n2:=#l2;
  n1~=n2 => false;
  tt:Boolean:=true;
  for i in 1..n1 repeat
  {
    ~listEq?(l1(i),l2(i)) => {tt:=false; break; }
  }
  tt;
}

local termOrder?(t1:Term, t2:Term): Boolean==
{
  szero? t1 or vzero? t1 => true;
  szero? t2 or vzero? t2 => false;
  listOrder?(t1.vec,t2.vec) and ~listEq?(t1.vec,t2.vec) => true;
  ~listOrder?(t1.vec,t2.vec) => false;
  dblListOrder?(t1.sca,t2.sca) and ~dblListEq?(t1.sca,t2.sca) =>true;
  ~dblListOrder?(t1.sca,t2.sca) => false;
  true;
}

local termEq?(t1:Term, t2:Term): Boolean==
{
  t1.coe=t2.coe and dblListEq?(t1.sca,t2.sca) and listEq?(t1.vec,t2.vec);
}
```

```
  local permute4():List List MI==
  {
    import from Set MI,List MI,List List MI;
    a:Set MI:=[1,2,3,4];
    out:List List MI:=empty;
    for i in a repeat
    {
      for j in a-([i] @ Set MI) repeat
      {
        for k in a-([i,j] @ Set MI) repeat
        {
          for l in a-([i,j,k] @ Set MI) repeat
          {
            out:=cons([i,j,k,l],out);
          }
        }
      }
    }
    out;
  }

-- The function for simplifying vector expressions --

  simplify(x:%, flag:Boolean==false):%==
  {
    xx:=copy rep x;
    empty? xx => x;
    lo:Rep:=empty;
    for tx in xx repeat
    {
      tpx:Term:=[tx.coe, copy tx.sca, copy tx.vec];
```

```
#tpx.vec=2 and tpx.vec(1)=tx.vec(2) => lo:=lo;
if #tpx.vec=2 and tpx.vec(1)>tpx.vec(2) then
{
  tpx.vec:=[tpx.vec(2),tpx.vec(1)];
  tpx.coe:=(-1)*tpx.coe;
}
empty? tpx.sca => lo:=cons(tpx, lo);
li:List List String:=empty;
for ts in tpx.sca repeat
{
  n:=#ts;
  n=1 => {li:=cons(ts,li); }
  n=2 => {if ts(1)>ts(2) then { li:=cons([ts(2),ts(1)],li); }
  else {li:=cons(ts,li);} }
  n=3 =>
  {
    if ts(1)=ts(2) or ts(2)=ts(3) or ts(3)=ts(1) then
      {tpx.coe:=0; li:=tpx.sca; break; }
    else
    {
      tp:List String:=empty;
      ms:=min(min(ts(1),ts(2)),ts(3));
      (ll,nn):=find(ms,ts);
      if nn=1 then { tp:=ts; }
      else if nn=2 then { tp:=[ts(2),ts(3),ts(1)]; }
      else if nn=3 then { tp:=[ts(3),ts(1),ts(2)]; }
      if tp(2)>tp(3) then
      {
        tp:=[tp(1),tp(3),tp(2)];
        tpx.coe:=-tpx.coe;
      }
      li:=cons(tp,li);
      }
    }
  }
  tpx.sca:=sort!(li,listOrder?);
  lo:=cons(tpx,lo);
}
lo:=reverse lo;
lo:=simplify0(lo);
empty? lo => per lo;
yy:List Term:=sort!(lo,termOrder?);
lo:=[first yy];
yy:=rest yy;
for tx in yy repeat
{
  t1:=first lo;
  listEq?(tx.vec,t1.vec) and dblListEq?(tx.sca,t1.sca) =>
    lo:=cons([tx.coe+t1.coe,t1.sca,t1.vec],rest lo);
  lo:=cons(tx,lo);
}
lo:=reverse simplify0 lo;

if flag and #lo>2 then
{
  stp:List Term:=empty;
  while ~same?(per lo, per stp) repeat
  {
    stp:=copy lo;
    tpzz:List Term:=rule01(lo);
    l:%:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule01(lo,true);
    l:%:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule11(lo);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule02(lo);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule02(lo,true);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule03(lo);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule04(lo);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
    tpzz:=rule04(lo,true);
    l:=simplify per tpzz;
    if #(rep l)<#lo then lo:=rep l;
    #lo<3 => break;
  }
  per lo;
}
else
{
      per lo;
    }
  }

-- implementations of transformation rules --

  import from Set String;
  import from Set List String;
  import from List Set String;
  import from List Set List String;

  local setsGcd0 (lt:List Set String):Set String==
  {
    #lt=1 => first lt;
    tp0:=first lt;
    rslt:=rest lt;
    for tx in rslt repeat
    {
      tp:=union(tp0,tx);
      tp:=tp-(tp0-tx);
      tp:=tp-(tx-tp0);
      tp0:=tp;
    }
    tp0;
  }

  local setsGcd (lt:List Set List String):Set List String==
  {
    #lt=1 => first lt;
    tp0:=first lt;
    rslt:=rest lt;
    for tx in rslt repeat
    {
      tp:=union(tp0,tx);
      tp:=tp-(tp0-tx);
      tp:=tp-(tx-tp0);
      tp0:=tp;
    }
    tp0;
  }

  local extend? (xx:List Term):Boolean==
  {
    realVector?(per xx) => false;
    tt:Boolean:=true;
    n2:MI:=0;
    n3:MI:=0;
    for tx in xx repeat
    {
      tp:=tx.sca;
      n:=#tp;
      n<3 => {tt:=false; break;}
      if #tp(n)=2
      then n2:=n2+1;
      else if #tp(n-1)=#tp(n) and #tp(n)=3
      then n3:=n3+1;
    }
    if n2<3 or n2<12*n3 then tt:=false;
    tt;
  }

  local cond? (p1:scaTerm,p2:scaTerm,p3:scaTerm):Boolean==
  {
    tt:Boolean:=(p1.coe2=p2.coe2 or p1.coe2=(-1)*p2.coe2);
    tt:=tt and (p2.coe2=p3.coe2 or p2.coe2=(-1)*p3.coe2);
    tt:=tt and (p3.coe2=p1.coe2 or p3.coe2=(-1)*p1.coe2);
    ~tt => false;
    l1:=p1.sca2 pretend Set List String;
    l2:=p2.sca2 pretend Set List String;
    l3:=p3.sca2 pretend Set List String;
    sg:=setsGcd([l1,l2,l3]);
    s1:=(l1-sg) pretend List List String;
    s2:=(l2-sg) pretend List List String;
    s3:=(l3-sg) pretend List List String;
    tt:= #s1=2 and #s2=2 and #s3=2 and #s1(1)*#s1(2)=6;
    ~(tt and #s2(1)*#s2(2)=6 and #s3(1)*#s3(2)=6) => false;
    s1:=sort!(s1,listOrder?);
    s2:=sort!(s2,listOrder?);
    s3:=sort!(s3,listOrder?);
    t12:=s1(1) pretend Set String;
    t22:=s2(1) pretend Set String;
    t32:=s3(1) pretend Set String;
    t13:=s1(2) pretend Set String;
    t23:=s2(2) pretend Set String;
    t33:=s3(2) pretend Set String;
    ut2:=union(union(t12,t22),t32);
    ut3:=union(union(t13,t23),t33);
    it2:Set String:=setsGcd0([t12,t22,t32]);
    it3:Set String:=setsGcd0([t13,t23,t33]);
    tt:= #ut2=4 and #ut3=4 and #it2=1 and #it3=1;
    ~(tt and (ut2-it2)=(ut3-it3) and #(ut2-it2)=3) => false;
    true;
  }

  -- we assume #pi.vec=2 --
  local cond11? (p1:Term,p2:Term,p3:Term):Boolean==
  {
    tt:Boolean:=(p1.coe=p2.coe or p1.coe=(-1)*p2.coe);
    tt:=tt and (p2.coe=p3.coe or p2.coe=(-1)*p3.coe);
    tt:=tt and (p3.coe=p1.coe or p3.coe=(-1)*p1.coe);
    ~tt => false;
    l1:=p1.sca pretend Set List String;
    l2:=p2.sca pretend Set List String;
    l3:=p3.sca pretend Set List String;
    sg:=setsGcd([l1,l2,l3]);
    s1:=(l1-sg) pretend List List String;
    s2:=(l2-sg) pretend List List String;
    s3:=(l3-sg) pretend List List String;
    tt:= #s1=1 and #s2=1 and #s3=1 and #s1(1)=#s2(1);
    ~(tt and #s2(1)=#s3(1) and #s3(1)=2) => false;
    t11:=s1(1) pretend Set String;
    t21:=s2(1) pretend Set String;
    t31:=s3(1) pretend Set String;
    t12:=p1.vec pretend Set String;
    t22:=p2.vec pretend Set String;
    t32:=p3.vec pretend Set String;
    ut1:=union(union(t11,t21),t31);
    ut2:=union(union(t12,t22),t32);
    it1:Set String:=setsGcd0([t11,t21,t31]);
    it2:Set String:=setsGcd0([t12,t22,t32]);
    ~(#it1=1 and #it2=0 and #ut2=3 and #setsGcd0([ut1,ut2])=3)=> false;
    true;
  }

  local distribute (p:mixTerm):List Term==
  {
    l:List Term:=empty;
    for tx in p.scal repeat
    {
      l:=cons([tx.coe2, tx.sca2, p.vec2], l);
    }
    l;
  }

  local same?(x:%, y:%): Boolean==
  {
    z:=simplify(x-y);
    zz:=rep z;
    #zz=1 and (vzero?(first(zz)) or szero?(first(zz))) => true;
    false;
  }

  local comb(p:mixTerm):List Term==
  {
    l1:=(p.scal(1)).sca2 pretend Set List String;
    l2:=(p.scal(2)).sca2 pretend Set List String;
    l3:=(p.scal(3)).sca2 pretend Set List String;
    sg:=setsGcd([l1,l2,l3]);
    s1:=(l1-sg) pretend List List String;
    s2:=(l2-sg) pretend List List String;
    s3:=(l3-sg) pretend List List String;
    s1:=sort!(s1,listOrder?);
    s2:=sort!(s2,listOrder?);
    s3:=sort!(s3,listOrder?);
    t12:=s1(1) pretend Set String;
    t22:=s2(1) pretend Set String;
    t32:=s3(1) pretend Set String;
    t13:=s1(2) pretend Set String;
    t23:=s2(2) pretend Set String;
    t33:=s3(2) pretend Set String;
    ut2:=union(union(t12,t22),t32);
    ut3:=union(union(t13,t23),t33);
    it2:Set String:=setsGcd0([t12,t22,t32]);
    it3:Set String:=setsGcd0([t13,t23,t33]);
    d:=it2(1);
    h:=it3(1);
    a:=(ut2-it2)(1);
    b:=(ut2-it2)(2);
    c:=(ut2-it2)(3);
    lg:=sg pretend List List String;
    coe:R:=(p.scal(1)).coe2;
    tp1:=[coe,append([[d,a],[h,b,c]],lg),p.vec2]@Term;
    tp2:=[coe,append([[d,b],[a,h,c]],lg),p.vec2]@Term;
    tp3:=[coe,append([[d,c],[a,b,h]],lg),p.vec2]@Term;
    same? (per [tp1,tp2,tp3], per distribute p) =>
      [[coe,append([[d,h],[a,b,c]],lg),p.vec2]];
    tp1:=[coe,append([[d,a],[h,c,b]],lg),p.vec2]@Term;
    tp2:=[coe,append([[d,c],[a,h,b]],lg),p.vec2]@Term;
    tp3:=[coe,append([[d,b],[a,c,h]],lg),p.vec2]@Term;
    same? (per [tp1,tp2,tp3], per distribute p) =>
      [[coe,append([[d,h],[a,c,b]],lg),p.vec2]];
    distribute p;
  }

  -- we assume #pi.vec=2 --
  local comb11(p1:Term, p2:Term, p3:Term):List Term==
  {
    l1:=p1.sca pretend Set List String;
    l2:=p2.sca pretend Set List String;
    l3:=p3.sca pretend Set List String;
    sg:=setsGcd([l1,l2,l3]);
    s1:=(l1-sg) pretend List List String;
    s2:=(l2-sg) pretend List List String;
    s3:=(l3-sg) pretend List List String;
    t11:=s1(1) pretend Set String;
    t21:=s2(1) pretend Set String;
    t31:=s3(1) pretend Set String;
    t12:=p1.vec pretend Set String;
    t22:=p2.vec pretend Set String;
    t32:=p3.vec pretend Set String;
    ut2:=union(union(t12,t22),t32);
    it1:Set String:=setsGcd0([t11,t21,t31]);
    d:=it1(1);
    a:=ut2(1);
    b:=ut2(2);
    c:=ut2(3);
    lg:=sg pretend List List String;
    tp1:=[p1.coe,cons([d,a],lg),[b,c]]@Term;
    tp2:=[p1.coe,cons([d,b],lg),[c,a]]@Term;
    tp3:=[p1.coe,cons([d,c],lg),[a,b]]@Term;
    same? (per [tp1,tp2,tp3], per [p1,p2,p3]) =>
      [[p1.coe,cons([a,b,c],lg),[d]]];
    tp1:=[p1.coe,cons([d,a],lg),[c,b]]@Term;
    tp2:=[p1.coe,cons([d,b],lg),[a,c]]@Term;
    tp3:=[p1.coe,cons([d,c],lg),[b,a]]@Term;
    same? (per [tp1,tp2,tp3], per [p1,p2,p3]) =>
      [[p1.coe,cons([a,c,b],lg),[d]]];
    [p1,p2,p3];
  }

  local rule01(yy:List Term, flag:Boolean==false):List Term ==
  {
    #yy<3 => yy;
    ~realVector?(per yy) => yy;
```

```
xx:=copy yy;
xx:=sort!(xx,termOrder?);
l:List Term:=empty;
for tx in xx repeat
{
  if #tx.vec ~= 1 or empty? tx.sca then l:=cons(tx,l);
  else
  {
    n:=#tx.sca;
    lt:List String:=empty;
    rt:List List String:=empty;
    if flag then
    {
      lt:= first reverse tx.sca;
      rt:= rest reverse tx.sca;
    }
    else
    {
      for i in 1..n repeat
      {
        #tx.sca(i)=3 =>
        {
          lt:=tx.sca(i);
          rt:=append([tx.sca(j) for j in 1..(i-1)],
                     [tx.sca(j) for j in (i+1)..n]);
          break;
        }
      }
    }
    d:= first tx.vec;
    if #lt ~= 3 then l:=cons(tx,l);
    else
```

```
      {
        l:=cons([tx.coe,append(rt,[[d,lt(3)]]),[lt(1),lt(2)]],l);
        l:=cons([tx.coe,append(rt,[[d,lt(1)]]),[lt(2),lt(3)]],l);
        l:=cons([tx.coe,append(rt,[[d,lt(2)]]),[lt(3),lt(1)]],l);
      }
    }
  }
  l;
}

local rule02(yy:List Term, flag:Boolean==false):List Term ==
{
  #yy<3 => yy;
  ~realVector?(per yy) => yy;
  xx:=copy yy;
  xx:=sort!(xx,termOrder?);
  l:List Term:=empty;
  for tx in xx repeat
  {
    if #tx.vec ~= 2 or empty? tx.sca then l:=cons(tx,l);
    else
    {
      n:=#tx.sca;
      lt:List String:=empty;
      rt:List List String:=empty;
      if flag then
      {
        lt:= first reverse tx.sca;
        rt:= rest reverse tx.sca;
      }
      else
      {
        for i in 1..n repeat
        {
          #tx.sca(i)=3 =>
          {
            lt:=tx.sca(i);
            rt:=append([tx.sca(j) for j in 1..(i-1)],
                       [tx.sca(j) for j in (i+1)..n]);
            break;
          }
        }
      }
      d:=tx.vec(1);
      h:=tx.vec(2);
      if #lt ~= 3 then l:=cons(tx,l);
      else
      {
        l:=cons([tx.coe,append(rt,[[lt(2),d],[lt(3),h]]),[lt(1)]],l);
        l:=cons([-tx.coe,append(rt,[[lt(2),h],[lt(3),d]]),[lt(1)]],l);
        l:=cons([tx.coe,append(rt,[[lt(3),d],[lt(1),h]]),[lt(2)]],l);
        l:=cons([-tx.coe,append(rt,[[lt(3),h],[lt(1),d]]),[lt(2)]],l);
        l:=cons([tx.coe,append(rt,[[lt(1),d],[lt(2),h]]),[lt(3)]],l);
        l:=cons([-tx.coe,append(rt,[[lt(1),h],[lt(2),d]]),[lt(3)]],l);
      }
    }
  }
  l;
}

local rule03(yy:List Term):List Term ==
{
  xx:=copy yy;
  #xx<3 => xx;
```

```
xx:=sort!(xx,termOrder?);
tp00:=first xx;
zz:=rest xx;
lo:=[ [[[tp00.coe,tp00.sca]],tp00.vec] ]@List mixTerm;
for tx in zz repeat
{
  tp01:=first lo;
  listEq? (tp01.vec2, tx.vec) =>
    lo:=cons([cons([tx.coe,tx.sca],tp01.scal),tp01.vec2],rest lo);
  lo:=cons([[[tx.coe,tx.sca]],tx.vec],lo);
}
l:List Term:=empty;
for tx in lo repeat
{
  L:scaList:=tx.scal;
  n:=#L;
  n<3 => l:=append(l, distribute tx);
  ttp2:scaList:=empty;
  for i in 1..n repeat
  {
    empty? L(i).sca2 or #(first (reverse L(i).sca2))<3 =>
      l:=append(l,distribute [[L(i)],tx.vec2]);
    ttp2:=cons(L(i), ttp2);
  }
  L:=ttp2;
  n:=#L;
  tt:Boolean:=false;
  ttp:scaList:=empty;
  ttp0:scaList:=empty;
  while n>2 repeat
  {
    for i in 1..(n-2) repeat
```

```
      {
        for j in (i+1)..(n-1) repeat
        {
          for k in (j+1)..n repeat
          {
            if cond? (L(i),L(j),L(k)) then
            {
              ttp:=[L(i),L(j),L(k)];
              ttp0:=append([L(s) for s in 1..(i-1)],
                           [L(s) for s in (i+1)..(j-1)]);
              ttp0:=append(ttp0,[L(s) for s in (j+1)..(k-1)]);
              ttp0:=append(ttp0,[L(s) for s in (k+1)..n]);
              L:=ttp0;
              tt:=true;
              break;
            }
          }
          if tt then break;
        }
        if tt then break;
      }
      empty? ttp => break;
      l:=append(l,comb [ttp,tx.vec2]);
      ttp:=empty;
      n:=#L;
    }
    if ~empty? L then l:=append(l,distribute [L,tx.vec2]);
  }
  l;
}

local rule04(yy:List Term, flag:Boolean==false):List Term ==
{
  xx:=copy yy;
  #xx<3 => xx;
  xx:=sort!(xx,termOrder?);
  if extend?(xx) then xx:=rep simplify per rule13(xx);
  l:List Term:=empty;
  for tx in xx repeat
  {
    tp0:=tx.sca;
    n:=#tp0;
    if n<2 then l:=cons(tx,l);
    else
    {
      lt31:List String:=empty;
      lt32:List String:=empty;
      if flag then
      {
        lt31:=tp0(n-1);
        lt32:=tp0(n);
        rt:List List String:=[tp0(j) for j in 1..(n-2)];
      }
      else
      {
        for i in 1..n-1 repeat
        {
          #tp0(i)=3 and #tp0(i+1)=3 =>
          {
            lt31:=tp0(i);
            lt32:=tp0(i+1);
            rt:=append([tp0(j) for j in 1..(i-1)],
                       [tp0(j) for j in (i+2)..n]);
            break;
          }
        }
      }
      if #lt31*#lt32 ~= 9 then l:=cons(tx,l);
      else
      {
        l:=cons([tx.coe,append(rt,[[lt31(2),lt32(1)],
                [lt31(3),lt32(2)],[lt31(1),lt32(3)]]),tx.vec],l);
        l:=cons([-tx.coe,append(rt,[[lt31(2),lt32(2)],
                [lt31(3),lt32(1)],[lt31(1),lt32(3)]]),tx.vec],l);
        l:=cons([tx.coe,append(rt,[[lt31(3),lt32(1)],
                [lt31(1),lt32(2)],[lt31(2),lt32(3)]]),tx.vec],l);
        l:=cons([-tx.coe,append(rt,[[lt31(3),lt32(2)],
                [lt31(1),lt32(1)],[lt31(2),lt32(3)]]),tx.vec],l);
        l:=cons([tx.coe,append(rt,[[lt31(1),lt32(1)],
                [lt31(2),lt32(2)],[lt31(3),lt32(3)]]),tx.vec],l);
        l:=cons([-tx.coe,append(rt,[[lt31(1),lt32(2)],
                [lt31(2),lt32(1)],[lt31(3),lt32(3)]]),tx.vec],l);
      }
    }
  }
  l;
}

local rule11(yy:List Term):List Term ==
{
  #yy<3 => yy;
  ~realVector?(per yy) => yy;
  xx:=copy yy;
  xx:=sort!(xx,termOrder?);
  l:List Term:=empty;
  lr:List Term:=xx;
```

```
for tx in xx repeat
{
  #tx.vec=1 =>
  {
    l:=cons(tx,l);
    lr:=rest lr;
  }
  break;
}
n:=#lr;
n<3 => l:=append(l,lr);
tt:Boolean:=false;
ttp:List Term:=empty;
ttp0:List Term:=empty;
while n>2 repeat
{
  for i in 1..(n-2) repeat
  {
    for j in (i+1)..(n-1) repeat
    {
      for k in (j+1)..n repeat
      {
        if cond11? (lr(i),lr(j),lr(k)) then
        {
          ttp:=[lr(i),lr(j),lr(k)];
          ttp0:=append([lr(s) for s in 1..(i-1)],
                       [lr(s) for s in (i+1)..(j-1)]);
          ttp0:=append(ttp0,[lr(s) for s in (j+1)..(k-1)]);
          ttp0:=append(ttp0,[lr(s) for s in (k+1)..n]);
          lr:=ttp0;
          tt:=true;
          break;
          }
        }
        if tt then break;
      }
      if tt then break;
    }
    empty? ttp => break;
    l:=append(l,comb11(ttp(1),ttp(2),ttp(3)));
    ttp:=empty;
    n:=#lr;
  }
  if ~empty? lr then l:=append(l,lr);
  l;
}

local rule13(yy:List Term):List Term ==
{
  xx:=copy yy;
  #xx<3 => xx;
  xx:=sort!(xx,termOrder?);
  l:List Term:=empty;
  for tx in xx repeat
  {
    tp0:=tx.sca;
    n:=#tp0;
    if n<2 then l:=cons(tx,l);
    else
    {
      lt2:List String:=empty;
      lt3:List String:=empty;
      for i in 1..n-1 repeat
      {
          #tp0(i)=2 and #tp0(i+1)=3 =>
          {
            lt2:=tp0(i); lt3:=tp0(i+1);
            rt:=append([tp0(j) for j in 1..(i-1)],
                       [tp0(j) for j in (i+2)..n]);
            break;
          }
        }
        if empty? lt2 then l:=cons(tx,l);
        else
        {
          l:=cons([tx.coe,append(rt,[[lt2(1),lt3(1)],
                  [lt2(2),lt3(2),lt3(3)]]),tx.vec],l);
          l:=cons([tx.coe,append(rt,[[lt2(1),lt3(2)],
                  [lt3(1),lt2(2),lt3(3)]]),tx.vec],l);
          l:=cons([tx.coe,append(rt,[[lt2(1),lt3(3)],
                  [lt3(1),lt3(2),lt2(2)]]),tx.vec],l);
        }
      }
    }
    l;
  }

}
```

Bibliography

[1] Aldor.org. *Aldor Compiler User Guide.* http://www.aldor.org.

[2] Aldor.org. *Algebra User Guide and Reference Manual.* ftp://ftp-sop.inria.fr/cafe/software.

[3] A. Belmonte and P. B. Yasskin. *A Vector Calculus Package for Maple.* http://calclab.tamu.edu/maple/veccalc/.

[4] Sydney Chapman and T. G. Cowling. *The mathematical theory of non-uniform gases.* Cambridge University Press, 1939.

[5] J. Cunningham. *Vectors.* Heinemann Educational Books Ltd, London, 1969.

[6] J. W. Eastwood. Orthovec: version 2 of the reduce program for 3-d vector analysis in orthogonal curvilinear coordinates. *Comput. Phys. Commun.*, 64:121–122, 1991.

[7] B. Fiedler. Vectan 1.1. *Manual Math. Inst., Univ. Leipzig*, pages 1–22, 1997.

[8] J. Willard Gibbs. On the fundamental formulæ of dynamics. *American J. Math.*, II:49 – 64, 1879.

[9] J. Willard Gibbs. Elements of vector analysis. In *The Scientific Papers of J. Willard Gibbs.* Dover, 1961.

[10] The Mathlab Group. *Macsyma Reference Manual, Vol.II.* MIT, 1983.

[11] D. Harper. Vector33: A reduce program for vector algebra and calculus in orthogonal curvilinear coordinates. *Comput. Phys. Commun.*, 54:295–305, 1989.

[12] D. J. Jeffrey, S. Liang, and S. M. Watt. An abstract, coordinate-free, vector algebra package. In *Proc. Asian Symposium on Computer Mathematics, (ASCM 2005).* Seoul, South Korea, 2005.

[13] Edward Arthur Milne. *Vectorial Mechanics.* Methuen, 1948.

[14] W. K. Nicholson. *Linear algebra with applications (5th ed.).* McGraw-Hill Ryerson Limited, 2006.

[15] E. M. Patterson. *Solving Problems in Vector Algebra.* Oliver & Boyd Ltd, Edinburgh-London, 1968.

[16] M. Rost. On the dimension of a composition algebra. *Doc. Math. J. DMV*, 1:209–214, 1996.

[17] Z. K. Silagadze. Multi-dimensional vector product. *arXiv: math.ra*, 0204357, 2002.

[18] D. R. Stoutemyer. Symbolic computer vector analysis. *Computers & Mathematics with Applications*, 5:1–9, 1979.

[19] Peter Guthrie Tait. *An elementary treatise on quaternions, 3rd edition.* Cambridge University Press, 1890.

[20] W. M. Tang, H. Qin, and G. Rewoldt. Symbolic vector analysis in plasma physics. *Comput. Phys. Commun.*, 116:107–120, 1999.

[21] S. L. Tanimoto. *The Elements of Artificial Intelligence Using Common Lisp.* Computer Science Press, New York, 1990.

[22] S. Wolfram. *The Mathematica Book,3rd ed.* Wolfram Media/Cambridge University Press, 1996.