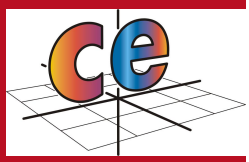
Optimizing a Geometric Algebra Compiler for parallel architectures using a Table-Based Approach

Optimieren eines Geometrische Algebra Compilers für parallele Architekturen durch Anwendung eines Tabellen-basierten Ansatzes

Bachelor-Thesis von Christian Steinmetz

26.10.2011

TECHNISCHE UNIVERSITÄT DARMSTADT



Optimizing a Geometric Algebra Compiler for parallel architectures using a Table-Based Approach
Optimieren eines Geometrische Algebra Compilers für parallele Architekturen durch Anwendung eines Tabellen-basierten Ansatzes

Vorgelegte Bachelor-Thesis von Christian Steinmetz

1. Gutachten: Dr. Ing. Dietmar Hildenbrand

Tag der Einreichung: 26.10.2011

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 26. Oktober 2011

(C. Steinmetz)

Abstract

Geometric Algebra is an algebra, which permits a calculation in an intuitive and an elegant way. Since a CPU is not able to calculate with Geometric Algebra directly, one need a compiler. Gaalop is such a compiler, which compiles a CLUScript, so that it can be used for instance in *C*-programs or FPGA implementations. Gaalop recently needs Maple™ for optimizing Geometric Algebra algorithms. The first main issue of this work is to implement a table-based approach in Gaalop in order to eliminate the need of Maple, since Maple is a commercial software. As a second main issue, we implemented a tool in Gaalop for optimizing Geometric Algebra algorithms for using on parallel architectures.

Contents

List of abbreviations

ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
CAS	Computer Algebra System
CPU	Central Processing Unit
CSV	Comma Separated Values
DLL	Dynamic Link Library
DSL	Domain Specific Language
FPGA	Field Programmable Gate Array
Gaalop	Geometric Algebra Algorithms Optimizer
GAPP	Geometric Algebra Algorithm Parallelism Program
GCD	Gaalop Compiler Driver
GPS	Global Positioning System
GUI	Graphical User Interface
IDE	Integrated Development Environment
TBA	Table-Based Approach
UML	Unified Modeling Language

1 Introduction

In this chapter, we give a motivation of this thesis, describe our contributions and give a short overview about the contents of the following chapters.

1.1 Motivation

First we give a motivation of this thesis.

1.1.1 Background of this work

Geometric Algebra facilitates a geometric calculation. An explicit representation of geometric objects like points, spheres or other is possible. It is also possible to calculate directly with these explicit representations. As described in section 3, a resulting calculation by hand is very expensive and long. But it is possible to use a Geometric Algebra compiler for creating this calculation. Also, the compiler is able to optimize the generated calculation and to generate code in many programming languages. Such a compiler is Gaalop. It compiles a CLUScript, which represent the Geometric Algebra algorithm, to source code. This generated source code does not contain Geometric Algebra any more, and so it can be simply used in programming projects. A disadvantage of the recent version of Gaalop is, that it uses Maple, which is a commercial software, for the transformation of Geometric Algebra algorithms.

The CLUScript code is intended for serial execution. The improvement of performance of architectures, which have a serial execution model, has certain limits. A solution is to compute calculations on parallel architectures. But finding parallel segments in a serial program is non-trivial. In paper [15], the GAPP language is introduced, which can be executed on parallel architectures with high performance. The paper contains an example, where a calculation in CLUScript code is transformed to an equivalent GAPP program.

1.1.2 Issues

In this thesis, we will implement a table-based approach in Gaalop in order to eliminate the need of Maple, because Maple is a commercial software. The second issue is making Gaalop able to create code for using on parallel architectures. Therefore we use the GAPP language and implement a tool in Gaalop, that transforms a CLUScript to an equivalent GAPP program. This will allow high-performance computing of Geometric Algebra algorithms.

1.2 Contributions

We make some contributions to Gaalop. In this chapter, we present these contributions briefly.

1.2.1 The Table-Based Approach

We implemented the Table-Based Approach plugin in Gaalop.

What follows is an overview over the advantages and the disadvantages of the Table-Based Approach.

Advantages of Table-Based Approach with Gaalop

Our table-based approach is fast, efficient and accurate. The *optional* Maxima interface optimizes the resulting algorithm on low-level mathematics. The user able to use external programs, like Maxima, but there is no duty to use external programs. Also we do not need a Clifford lib[1] any more. For higher performance of generated algorithms, we added some optimizations as for example dead-code elimination. We can handle arbitrary algebras. Theoretically, there is no dimension constraint.

[1] http://math.tntech.edu/rafal/

Disadvantages of Table-Based Approach with Gaalop

But there are also a few of disadvantages of Table-Based Approach. So the user has to create a multiplication table prior of the optimization. For the 2d, 3d and the Conformal Geometric Algebra, we have included such a multiplication table. The next disadvantage is, that storage of multiplication table is inefficient due to many entries. Especially for higher-dimensional algebras the requirements for storage of the tables is a big disadvantage.

1.2.2 The Java plugin

We implemented also a code generator that generates Java code. So, Geometric Algebra algorithms can be easily used in a Java project. See section 9.2 for details.

1.2.3 The GAPP optimizer

We added also a support for parallel architectures. There were no implementation hints given in the description of the instruction set in paper[15]. Only an example was given. Therefore we must introduce a way to implement the transformation process. Note, that we must also extend the data structure of Gaalop to support GAPP, but we had to ensure that recent plugins (code parser, optimizers and code generators) would still work.

1.2.4 The GAPP Pretty Printer plugin

We implemented also a code generator that prints out a human-readable version of the GAPP instructions. See section 9.4 for details.

1.3 Structure of this thesis

At first, we give a short overview over the related work in chapter 2. In chapter 3, we present some Geometric Algebra facts, which are used in this thesis. Since we upgrade Gaalop, a Geometric Algebra algorithm compiler, we give an overview over the structure of Gaalop in chapter 4. Design Patterns are used intensively in the source code of Gaalop, so we give a description of the used Design Patterns in chapter 5. Chapter 6 presents the Table-Based Approach, which is used to optimize Geometric Algebra algorithms, and its integration in Gaalop. In chapter 7, we present some optimizations to optimize the algorithm after applying the Table-Based Approach. Since we make Gaalop supporting parallel architectures, we describe the transformation process in chapter 8. A manual for the new Gaalop plugins is given in chapter 9. To validate our contributions, we developed some tests, which we present in chapter 10. In chapter 11, we compare the results of our plugins with the recent plugins. A conclusion and the overview over the future work concludes this thesis.

2 Related Work

There are some other tools for optimizing Geometric Algebra algorithms. Some examples are Gaigen, GMac and Gaalet. What follows is a short description of those tools and a comparison with Gaalop.

2.1 Gaalop with Maple optimization

Gaalop depends in its recent implementation on Maple. Maple is a commercial software. So you must have a license for running Maple. Using the table-based approach presented in this thesis, the use of Maple can be omitted, so you do not need a license any more.

As replacement we use Maxima. Maxima is a free open-source software, which is a powerful tool in optimizing mathematical calculations. But we want to emphasise, that there is no need to use Maxima. Since the usage of Maxima can improve performance of the generated code, we recommend the usage of Maxima.

The code architecture of Gaalop contains many Design Patterns. We describe some used Design Patterns in chapter 5. The usage of Design Patterns make the code more clearly and easier to reuse.

Gaalop optimizes CLUScripts. CLUScripts can be visualized by using the program CLUCalc/CLUViz[1].

2.2 Gaigen 2

Gaigen 2[2] uses data like the dimension, metric and the base of a Geometric Algebra for optimizing. It has its own DSL for describing the Geometric Algebra algorithm.

What follows is a short list of the advantages and the disadvantages of Gaigen 2 in comparison with Gaalop with Table-Based Approach.

Advantages of Gaigen 2

- Handling of an arbitrary Geometric Algebra without a given multiplication table prior to optimization.

Disadvantages of Gaigen 2

- No mathematical optimization after removal of Geometric Algebra
- An own DSL is used for describing the Geometric Algebra algorithm. This DSL is not usable for other programs. Gaalop optimizes CLUScript, which can be used for visualisation by using the program CLUCalc / CLUViz.
- The performance of the generated code depends strongly on the effort of the user in designing the algorithm, because the user is able to bias on which optimizations will be done by the tool. With Gaalop, the user do not have to create an optimal optimized algorithm, because the Gaalop tool optimizes the algorithm itself. But also like in Gaigen, the user of Gaalop has the possibility to support the optimization by setting pragmas. The pragmas and their usage are described in section 7.3.

2.3 GMac

Ahmad Hosney Awad Eid introduced in his dissertation[11] an approach for compiling Geometric Algebra algorithms with the use of Mathematica. But Mathematica is commercial software and this is the big disadvantage compared to our Gaalop with Table-Based Approach.

Advantages of GMac

- Single-pass code generation[3]

[1] www.clucalc.info

[2] http://staff.science.uva.nl/~fontijne/gaigen2.html

[3] Single-pass code generation means that the compiler needs only one pass reading the algorithm and writing the output. Remark: Single-pass compiling might lead to unintuitive programming. Multi-pass-compilers are more clearly because they separate the compiling process in many logical steps and so programming is more easier, especially when programming in a team.

Disadvantages of GMac

- No control flow is supported yet. With GCD (described below), Gaalop is capable of supporting control flow.
- Needs Mathematica (a commercial software) to operate.

2.4 Gaalet

Gaalet, which is described in [18], uses the C++ Compiler to optimize Geometric Algebra algorithms. There exist templates for representing multivectors and Geometric Algebra operations. The algorithms are represented by using these templates. There is no need for communication to an external program like Gaalop, Gaigen or CAS. This is the main difference to all other Geometric Algebra optimizers.

Advantages of Gaalet

- Easy, neatless integration in C++ source code
- No need of an external program
- Control flow is handled automatically and so it is fully supported without restrictions

Disadvantages of Gaalet

- No optimization is possible (for example mathematical low-level optimizations).
- Writing the Geometric Algebra algorithm can be long and this leads to a more difficult reading of source code.
- Bounded to C++ as input.
- No explicit output of optimized algorithms is possible.

2.5 GCD

GCD is a tool for Gaalop for embedding Geometric Algebra algorithms in *C/C++* code in an elegant and comfortable way. With the help of GCD, Gaalop is capable of using control flow. Our table-based approach has been tested successfully with GCD. For more information, we refer to [10].

3 Essentials of Geometric Algebra

Every Geometric Algebra has base vectors. In the Conformal Geometric Algebra these are: e_1, e_2, e_3, e_+, e_-. Also another bases are possible, for example the base $e_1, e_2, e_3, e_\infty, e_0$ with $e_0 = \frac{1}{2}(e_- + e_+)$ and $e_\infty = e_- + e_+$. We use the last-mentioned one, because interpretation of this base is more intuitive and so creation of algorithms is easier.

The number of base elements is equal to the dimension of the algebra. Every base vector has also a signature, which is the square of the base vector.

3.1 Blades

The combinations of base elements using the outer product forms the set of *blades*. The order and the inner order of the blades is not unique and so it can be chosen arbitrary. We use the blade order for Conformal Geometric Algebra shown in table 3.1.

A sum of weighted blades is called *multivector*.

3.2 The products of Geometric Algebra

There a many products included in a Geometric Algebra. Three of them are used frequently. The first is the well-known dot product. In Geometric Algebra, it is called more commonly "the inner product"., The second is the outer product and the third is the geometric product.

For a more detailed description of Geometric Algebra, we refer to [12].

3.2.1 The inner product

The outer product is shown as: "$\cdot$"

The inner product performs a grade degradation. With the inner product, the angle between two vectors can be calculated for example. The inner product of two perpendicular vectors is zero.

3.2.2 The outer product

The outer product is shown as: "$\wedge$".

It performs a grade increment. It is anti-symmetric, linear and associative. The outer product is a measure of parallelism, so the outer product of two parallel vectors is 0.

3.2.3 The geometric product

The geometric product is the main product in Geometric Algebra. It is used for example for doing transformations with translators and rotors. The outer and the inner product can be derived from geometric product:

$$u \cdot v = \frac{1}{2}(uv + vu)$$

$$u \wedge v = \frac{1}{2}(uv - vu)$$

.

3.3 Representations in Conformal Geometric Algebra

We provide a short list of some representations in Conformal Geometric Algebra in table 3.2.

index	blade	grade
0	1	0
1	e_1	1
2	e_2	1
3	e_3	1
4	e_∞	1
5	e_0	1
6	$e_1 \wedge e_2$	2
7	$e_1 \wedge e_3$	2
8	$e_1 \wedge e_\infty$	2
9	$e_1 \wedge e_0$	2
10	$e_2 \wedge e_3$	2
11	$e_2 \wedge e_\infty$	2
12	$e_2 \wedge e_0$	2
13	$e_3 \wedge e_\infty$	2
14	$e_3 \wedge e_0$	2
15	$e_\infty \wedge e_0$	2
16	$e_1 \wedge e_2 \wedge e_3$	3
17	$e_1 \wedge e_2 \wedge e_\infty$	3
18	$e_1 \wedge e_2 \wedge e_0$	3
19	$e_1 \wedge e_3 \wedge e_\infty$	3
20	$e_1 \wedge e_3 \wedge e_0$	3
21	$e_1 \wedge e_\infty \wedge e_0$	3
22	$e_2 \wedge e_3 \wedge e_\infty$	3
23	$e_2 \wedge e_3 \wedge e_0$	3
24	$e_2 \wedge e_\infty \wedge e_0$	3
25	$e_3 \wedge e_\infty \wedge e_0$	3
26	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty$	4
27	$e_1 \wedge e_2 \wedge e_3 \wedge e_0$	4
28	$e_1 \wedge e_2 \wedge e_\infty \wedge e_0$	4
29	$e_1 \wedge e_3 \wedge e_\infty \wedge e_0$	4
30	$e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	4
31	$e_1 \wedge e_2 \wedge e_3 \wedge e_\infty \wedge e_0$	5

Table 3.1.: Blade order in Conformal Geometric Algebra, taken from [15]

entity	standard representation	direct representation
Point	$P = x + \frac{1}{2}\mathbf{x}^2 e_\infty + e_0$	
Sphere	$S = P - \frac{1}{2}r^2 e_\infty$	$S^* = P_1 \wedge P_2 \wedge P_3 \wedge P_4$
Plane	$\pi = \mathbf{n} + d e_\infty$	$\pi^* = P_1 \wedge P_2 \wedge P_3 \wedge e_\infty$
Circle	$Z = S_1 \wedge S_2$	$Z^* = P_1 \wedge P_2 \wedge P_3$
Line	$L = \pi_1 \wedge \pi_2$	$L^* = P_1 \wedge P_2 \wedge e_\infty$
Point Pair	$Pp = S_1 \wedge S_2 \wedge S_3$	$Pp = P_1 \wedge P_2$

Table 3.2.: List of some representations in Conformal Geometric Algebra, taken from [14]

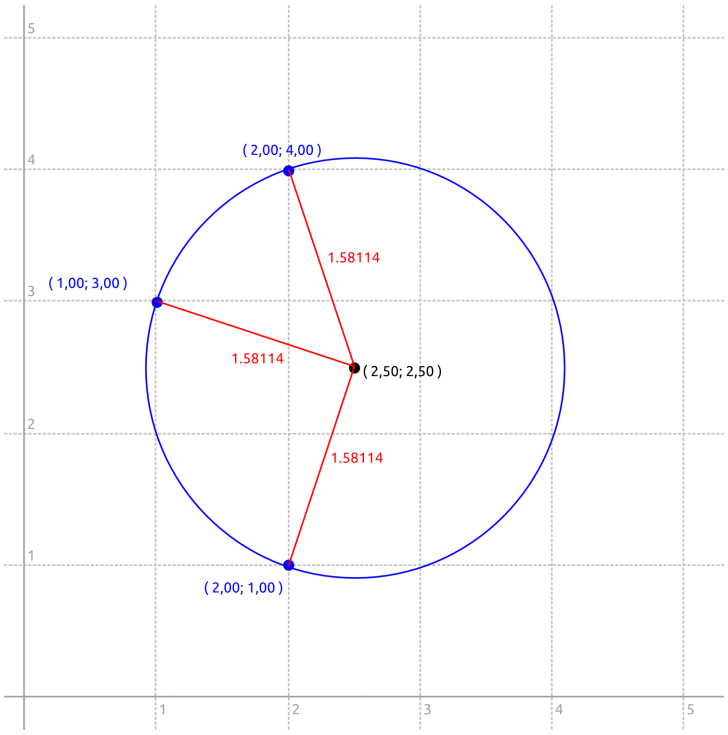


Figure 3.1.: Visualization of the scenario for the exemplary calculation

3.4 Exemplary calculation

To show the elegance of Geometric Algebra, we demonstrate an example of a calculation in Geometric Algebra. We choose the following scenario:

3.4.1 Scenario description

Suppose you have three points in two-dimensional euclidean space. The coordinates of these three points are $p_1, p_2, p_3 \in \mathbb{R}^2$. These three points lie on a circle. The task is now to determine the centre of this circle. See 3.1 for a visualisation of an instance of this problem.

3.4.2 Solution in linear algebra

To solve this problem in linear algebra, consider the points as corners of a triangle $\triangle p_1 p_2 p_3$. The circle which lies on the corners of this triangle is the circumscribed circle of the triangle.

Geometrically, the centre point of the circumscribed circle is the intersection point of the perpendicular bisectors of the sides of the triangle. So the task is to determine the intersection point.

As an instance of the problem, we choose the following points:

$$\vec{p_1} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \vec{p_2} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \vec{p_3} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

The perpendicular bisectors of the sides of the triangle are:

$$L_1 = \begin{pmatrix} 1.5 \\ 2 \end{pmatrix} + s \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}$$

$$L_2 = \begin{pmatrix} 1.5 \\ 3.5 \end{pmatrix} + t \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

with $s, t \in \mathbb{R}$.

The intersection point is defined by $L_1 = L_2$ for some s, t. Solving the linear equation system yields: $s = t = 1$. So, the intersection point is located at $\begin{pmatrix} 2.5 \\ 2.5 \end{pmatrix}$.

3.4.3 Solution in Geometric Algebra

The calculation in Geometric Algebra is more intuitive than the linear algebra one. One do not need to have the idea of the circumscribed circle, because the calculation in Geometric Algebra is straightforward. We choose the Conformal Geometric Algebra for calculation.

At first, the euclidean points are transformed into Geometric Algebra space: An euclidean point p is in Geometric Algebra given by $p + 0.5(p \cdot p)e_\infty + e_0$. So the representation of the three points is:

$$\begin{aligned} p_1 &:= 2e_1 + 1e_2 + 2.5e_\infty + e_0 \\ p_2 &:= 1e_1 + 3e_2 + 5e_\infty + e_0 \\ p_3 &:= 2e_1 + 4e_2 + 10e_\infty + e_0 \end{aligned}$$

The circle which is given by these three points can be build up using the outer product. [1]

$$c = (p_1 \wedge p_2 \wedge p_3)^*$$

The (not normalized) centre point m of the circle is:

$$m = ce_\infty c$$

Now, m must be normalized.

$$m_{\text{normalized}} = -\frac{p}{p \cdot e_\infty}$$

Fetching the e_1 and e_2 coefficient of $m_{\text{normalized}}$ gives the coordinates of the centre points. The complete calculation can be found in the appendix A.

3.4.4 CLUScript

We provide a CLUScript, which performs the calculation of the previous section. It is listed in listing 3.1.

[1] Note, that a dualization, which is marked with an $*$, is necessary, because the representation of the circle is the "Outer Product Null Space"-variant.

Listing 3.1: CLUScript: Calculating the centre point of a circle defined by three points

```
// CLUScript for calculating the centre point of a circle defined by three points,
// author: Christian Steinmetz, 2011

DefVarsN3(); // sets the Conformal Geometric Algebra
:IPNS; // visualize the Inner-Product-Null-Space

// add slider for choosing the coordinates of the three input points
x1 = Slider("x1",-10,10,0.1,2,"The x-coordinate of the first point");
y1 = Slider("y1",-10,10,0.1,1,"The y-coordinate of the first point");

x2 = Slider("x2",-10,10,0.1,1,"The x-coordinate of the second point");
y2 = Slider("y2",-10,10,0.1,3,"The y-coordinate of the second point");

x3 = Slider("x3",-10,10,0.1,2,"The x-coordinate of the third point");
y3 = Slider("y3",-10,10,0.1,4,"The y-coordinate of the third point");

// create three vectors of the coordinates of the three coordinate pairs
v1 = x1*e1+y1*e2;
v2 = x2*e1+y2*e2;
v3 = x3*e1+y3*e2;

// create three conformal points from the three vectors v1,v2,v3
// and visualize them in blue
:Blue;
:p1 = v1 + 0.5*v1*v1*einf + e0;
:p2 = v2 + 0.5*v2*v2*einf + e0;
:p3 = v3 + 0.5*v3*v3*einf + e0;

// use the OPNS-Version of creating a circle of three conformal points
// by using the outer product and visualize it in red
:Red;
:c = *(p1^p2^p3);

// extract the centre point of the circle c
m = c*einf*c;

// normalize the centre point m and visualize it in white
:White;
:mnor = -m/(m.einf);

// output the normalized centre point mnor
?mnor;
```

4 Description of Gaalop

Current processor architectures are not directly able to calculate with Geometric Algebra.

To take advantage of the above described elegance of Geometric Algebra usage, there are a lot of programs, which takes code that contains Geometric Algebra, and outputs code with no Geometric Algebra. Some main implementations are Gaigen and Gaalop[1], which uses Maple™[2] to do the transformation.

4.1 The internal representation of Gaalop

The internal representation can be separated in two different graphs: The control flow graph and the expression graph. A visualisation of both graphs is given in figure 4.1.

4.1.1 Control flow graph

The control flow graph is a graph which contains nodes. Every node represents a construct of the algorithm. There are nodes for assignments, constant loops and conditional constructs. The begin of the CLUScript is represented with a *StartNode*, the end with an *EndNode*. If-conditions and assignments can contain expressions. To represent the expression, an expression graph is used.

4.1.2 Expression graph

The expression graph is a data structure, which can represent mathematical expressions. There are two main types of expressions: Binary operations and unary operations.

A binary operation contains two subexpressions, an unary operation contains only one subexpression. As an example the addition is a subtype of the binary operation, because the addition calculates the sum of two operands. Additionally there are float constants, variables and base vectors, which are subtypes of expression. They are terminals, so they do not have any subexpressions.

4.2 Recent data flow of Gaalop

The recent data flow of Gaalop is shown in figure 4.2. The data flow of Gaalop is separated in three stages:

1. Code Parser
2. Optimizer
3. Code Generator

What follows is a description of each stage.

4.2.1 Code Parser

In general, the task of a code parser is to read the input source file, check it for consistence according to a set of rules and build up an internal representation. Mostly, an abstract syntax tree (AST) is used for the internal representation because it represents the behaviour of a construct and does not store concrete code fragments. As an example, the AST of an assignment is shown in figure 4.3. Therefore, it does not matter if an assignment is stated like “set a to 5” or “$a := 5$” or in an other way. Only the special behaviour “assign variable a to the value 5” is stored.

Having this fact in mind, following stages, that use the internal representation do not need to distinguish between concrete syntaxes, but use only the fact, that an assignment has to be made at this program code point. This results in a simplification for all stages that use the internal representation.

A parser has two substages: A *lexer* and a *parser*. The task of the lexer is to group the single chars to groups called *tokens*. While grouping chars to tokens, whitespaces are ignored. The parser use the tokens to build up the abstract syntax tree.

Gaalop is created to optimize CLUScripts, so the code parser of Gaalop parses CLUScripts into the internal representation.

[1] available at `http://www.gaalop.de`

[2] `http://www.maplesoft.com/products/Maple/index.aspx`

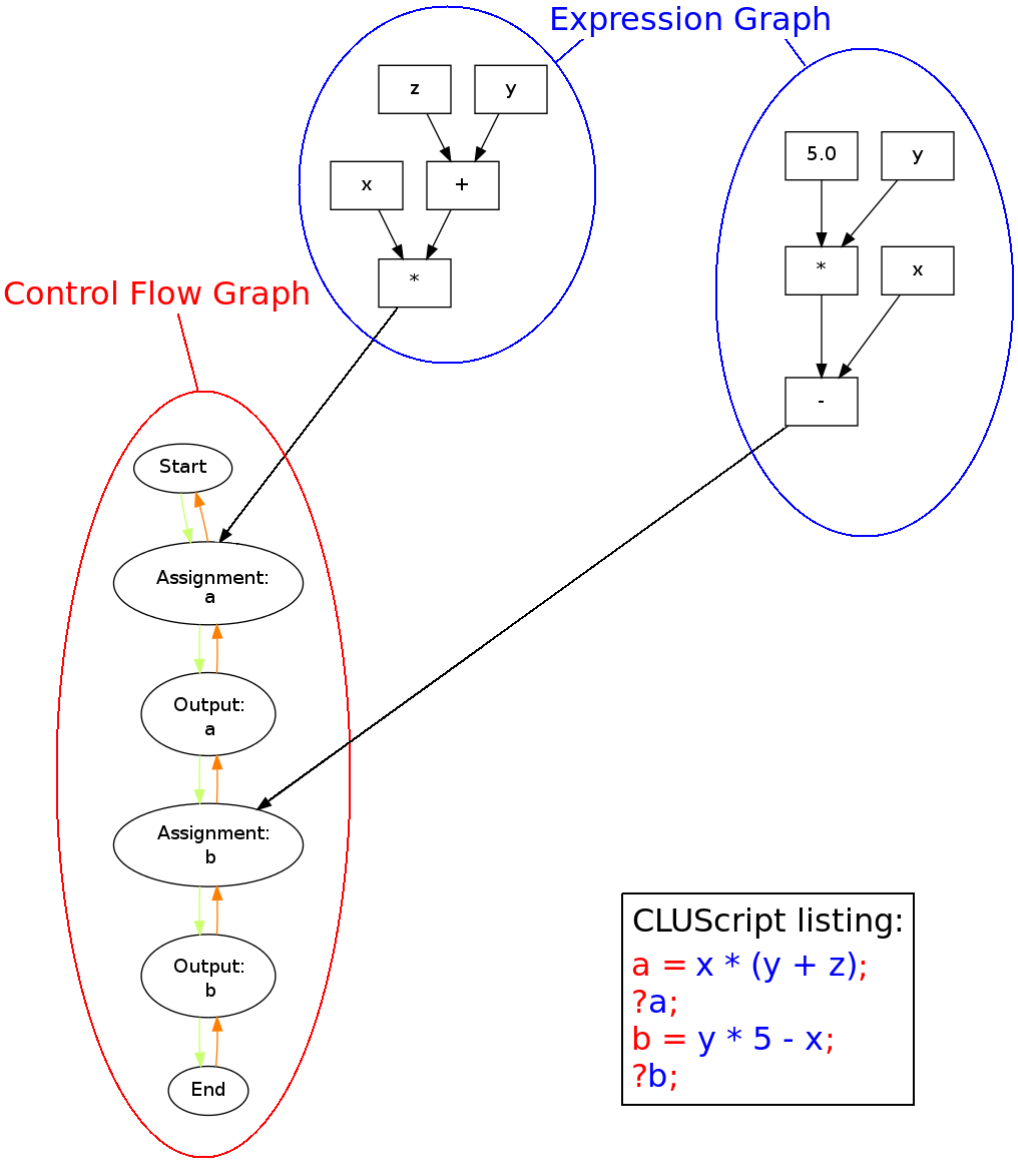


Figure 4.1.: Visualization of Control Flow Graph and Expression Graph

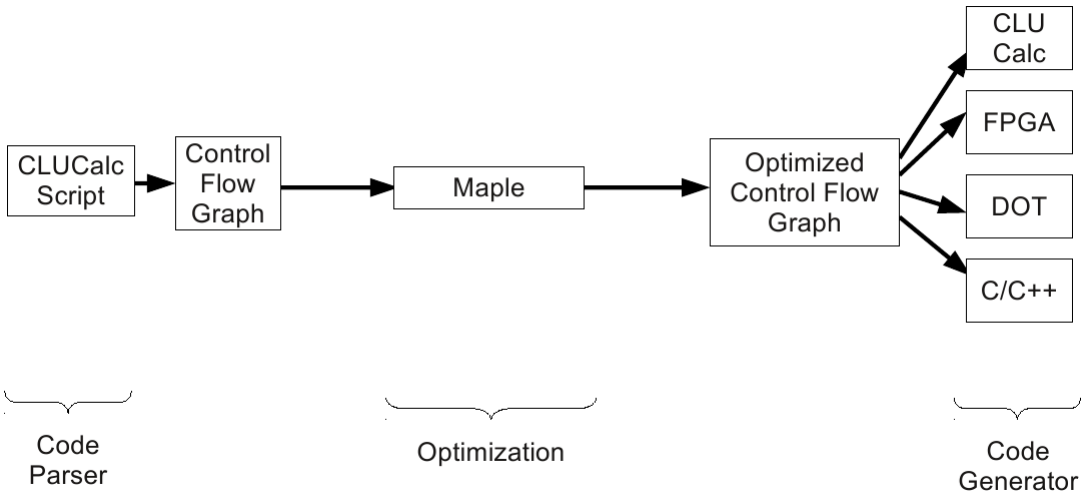


Figure 4.2.: The data flow of Gaalop

4.2.2 Optimizer

The internal representation can make use of Geometric Algebra. The optimizer stage eliminates the Geometric Algebra operations and base vectors. This can be done in several ways. The recent implementation uses Maple for elimination of Geometric Algebra.

Maple is able to calculate with the Conformal Geometric Algebra by using the Clifford lib for Maple with the two base elements e_+ and e_-.

Since many CLUScripts use another (more intuitive) base for calculating (e_0 and e_∞ instead of e_+ and e_-), it is necessary to tell Maple, how to interpret another base elements. This is done for instance with the rules given in [12]: $e_0 = \frac{1}{2}(e_- + e_+)$ and $e_\infty = e_- + e_+$.

4.2.3 Code Generator

The code generator stage is used to export the internal representation. There are many output formats possible. Since the optimizer stage has eliminated all Geometric Algebra constructs, it is possible to generate source code for C/C++, Java, Verilog and other programming languages. The Verilog code generator produces code for a FPGA. This makes it easy to develop hardware solutions, which can operate very fast.

It is also possible to generate CLUScript code. This can be used for simple manually testing. Another very useful output format for visualisation is DOT[3].

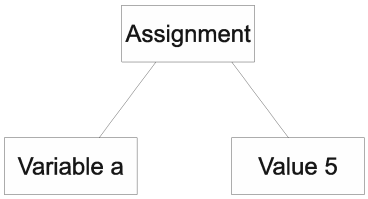


Figure 4.3.: The abstract syntax tree of an assignment of the value 5 to a variable a

[3] http://www.graphviz.org

5 Design Patterns

Design patterns are very important for efficient structured programming. They allow reusage and simple extensibility of existing software to fulfil new requirements.

Since Gaalop uses some Design Patterns, we provide a short description of them. For a detailed description and more Design Patterns, we refer to [13].

5.1 The Visitor Design Pattern

The *Visitor Design Pattern* provides the ability of adding functionalities to some classes (for example in a data structure) in a simple way while not manipulating the class implementation. In contrast to manipulating every class, an interface is used, which is called *AbstractVisitor*. Every class has its own *visit()*-method in the abstract visitor. In a *ConcreteVisitor*, which implements the *AbstractVisitor*, the new functionality can be integrated by overriding each *visit()*-method. Each class in the data structure has to implement an *accept()* method, which calls the class-according *visit()*-method in a *ConcreteVisitor* instance, which is a parameter of the *accept()*-method.

See figure 5.1 for the structure of this Design Pattern. The class *Addition* overrides the *accept()*-method, which calls the *visitAddition()*-method in a given visitor. As *Addition* instance, which is a parameter of the *visitAddition()*-method, simply choose the *this*-pointer. Analogously the class *Reverse* overrides the *accept()*-method, who calls the *visitReverse()*-method in a given visitor.

For providing a new functionality for all classes in the data structure, the only thing that must be done is deriving a new *ConcreteVisitor* from *AbstractVisitor* and implementing all *visit()* methods. It is not needed to modify all classes.

5.1.1 Usage in Gaalop

Especially in Gaalop, this Design Pattern is very helpful, because we have the ability to implement a varying number of code generators. In programming without usage of this Design Pattern, one has to implement a method for every code generator in every *Node* and *Expression* type. Additionally, every code generator must implement the walk through the data structure itself. So there is a lot of double and long code.

With the help of the Visitor Design Pattern, creation of new code generators or optimization strategies is easier.

Explicit usages in Gaalop

In Gaalop, the Visitor Design Pattern is used in the data structure of *ControlFlowGraph* and *Expression*. Both *ControlFlowGraph* and *Expression* have their own abstract visitor.

A traversal visitor is implemented in the Gaalop API, so a concrete visitor can be called on a whole control flow graph or expression, and the data structure can be visited in a traversal. See listing 5.1 for the source of the *ExpressionVisitor* interface. For all *OptimizationStrategy* instances, using these both visitors can be very helpful for a short implementation of transformation algorithms, which is easy to understand.

Since the *OptimizationStrategy* instances operate directly on the control flow graph, all code generators may use the visitors, too. Also for code generators is the visitor pattern an admirably choice for programming.

5.1.2 Advantages and Disadvantages

What follows is an overview over the most suitable advantages and the disadvantages for Gaalop of using the Visitor Design Pattern. For a complete list of the advantages and the disadvantages, we refer to [13].

Advantages

- Methods, which operate on the data structure, won't have to be implemented in all classes, only in a concrete visitor. This makes the algorithm easy to understand, clear and the classes in the data structure do not need to be modified.
- Other visitors may benefit form existing visitors by deriving from them.
- The classes in data structure must contain only one *accept()*-method for providing a potentially endless series of functionalities

Listing 5.1: Source of ExpressionVisitor

```
1  package de.gaalop.dfg;
2
3  /**
4   * This interface needs to be implemented by classes that want to iterate over data flow graphs.
5   * <p/>
6   * It provides a method for each concrete class that a node in a dataflow graph can be of.
7   */
8  public interface ExpressionVisitor {
9
10
11     void visit(Subtraction node);
12
13     void visit(Addition node);
14
15     void visit(Division node);
16
17     void visit(InnerProduct node);
18
19     void visit(Multiplication node);
20
21     void visit(MathFunctionCall node);
22
23     void visit(Variable node);
24
25     void visit(MultivectorComponent node);
26
27     void visit(Exponentiation node);
28
29     void visit(FloatConstant node);
30
31     void visit(OuterProduct node);
32
33     void visit(BaseVector node);
34
35     void visit(Negation node);
36
37     void visit(Reverse node);
38
39     void visit(LogicalOr node);
40
41     void visit(LogicalAnd node);
42
43     void visit(LogicalNegation node);
44
45     void visit(Equality node);
46
47     void visit(Inequality node);
48
49     void visit(Relation relation);
50
51     void visit(FunctionArgument node);
52
53     void visit(MacroCall node);
54
55 }
```

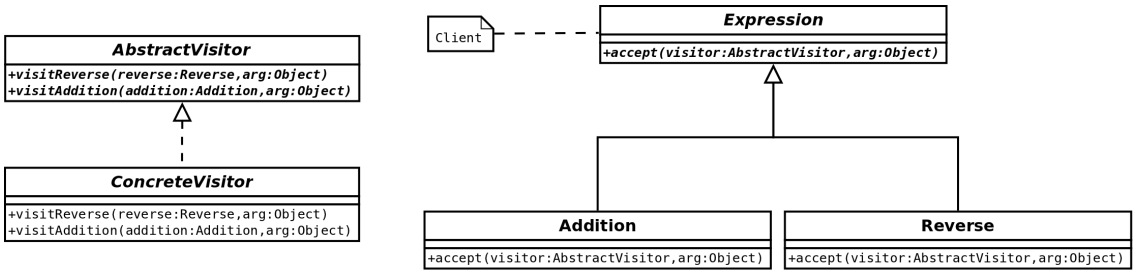


Figure 5.1.: Structure of the Visitor Design Pattern

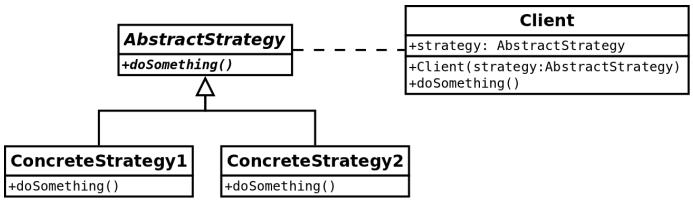


Figure 5.2.: Structure of the Strategy Design Pattern

Disadvantages

- Changing the data structure (for example adding or deleting classes) results in a necessary modification of the abstract visitor, which results in most cases in a modification of all concrete visitors.
- The "encapsulation" can be compromised. Encapsulation is a term of the field of Software Engineering. From the book *Design Patterns* [13], section 1.6:

  > "Operations are the *only* way to change an object's internal data. Because of these restrictions, the object's internal state is said to be **encapsulated**; it cannot be accessed directly, and its representation is invisible from outside the object."

- For beginners in programming, using this method may seem to be a bit confusing, but this Design Pattern is very useful and results in a good design.

5.2 The Strategy Design Pattern

In the *Strategy Design Pattern*, it is easy to choose and switch between algorithms for a given task. The task is represented in an interface *Strategy*. Every algorithm, which solves the task, implements this interface. Clients can use the interface *Strategy* for communicating with the unknown implementation to do the task. In their constructor, clients are informed, which concrete strategy they should use.

See figure 5.2 for a visualisation of the structure of the Strategy Design Pattern.

5.2.1 Usage in Gaalop

In Gaalop, the Strategy Design Pattern is used in all three stages: *Code Parser*, *OptimizationStrategy* and *Code Generator*. See, for example, the source of the *OptimizationStrategy* class in listing 5.2.

Listing 5.2: Source of OptimizationStrategy

```
1  package de.gaalop;
2
3  import de.gaalop.cfg.ControlFlowGraph;
4
5  /**
6   * This interface describes a strategy for source-to-source compilation which
7   * consists of a simple graph to graph transformation on the control flow graph
8   * that has been produced by the frontend.
9   */
10 public interface OptimizationStrategy {
11
12     /**
13      * Transforms a graph by applying optimization or analysis operations.
14      *
15      * @param graph The control flow graph to operate on.
16      * @throws OptimizationException If any error occurs during the transformation.
17      */
18     void transform(ControlFlowGraph graph) throws OptimizationException;
19
20 }
```

Every *OptimizationStrategy* must implement the *transform()*-method, which can throw an *OptimizationException*. Note, that there is no return value, so the graph must be modified in-place. This approach of modifying graph in-place has the advantage, that the graph do not need to be copied while transforming. This saves memory and reduces operating time.

Note also, that there is no constraint on the graph to be fulfilled from *OptimizationStrategy*. So optimizing the graph is optional.

This can be used to test the CLUCalc parser and CLUCalc generator by simply adding an *OptimizationStrategy* instance which has an empty body of the *transform()* method and testing if the readed CLUScript and the generated CLUScript have the same behaviour.

In the Table-Based Approach plugin we implemented also the Strategy Design Pattern in gathering all optimizations.

5.2.2 Advantages and Disadvantages

What follows is an overview over the most suitable advantages for Gaalop of using the Strategy Design Pattern. There are no grave disadvantages for using the Strategy Design Pattern in Gaalop, because Gaalop is designed for using as drag & drop application, where users can easily integrate their plugins without modification of the existing source. For a complete list of advantages and disadvantages, we refer to [13].

Advantages

- The Strategy Design Pattern gathers up all algorithms for code parsing, optimizing and code generating.
- New plugins are added easily.
- There are no conditional statements needed for choosing the certain strategy.

5.3 Other Design Patterns used in Gaalop

Gaalop contains more Design Patterns than Visitor and Strategy Design Pattern. More Design Patterns, that are used in Gaalop, are the *Facade Design Pattern* and the *Observer Design Pattern*. Please refer to [13] for a detailed description.

6 Table-Based Approach and integration in Gaalop

6.1 Mathematical background

We took the idea from [15] for a description of the mathematical background.

A multivector m can be written as

$$m = \sum_{i=0}^{2^n-1} m_i E_i$$

where n is the dimension of the algebra and E_i is the blade at index i. Let us denote a linear product which is distributive over addition as $\diamond$. Then

$$a \diamond b = \left(\sum_{i=0}^{2^n-1} m_i E_i \right) \diamond \left(\sum_{j=0}^{2^n-1} m_j E_j \right)$$

is the multiplication of two multivectors a and b. This can be written as

$$a \diamond b = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} m_i m_j E_i \diamond E_j$$

since the linear product $\diamond$ is distributive over addition. So the products $E_i \diamond E_j$ can be precomputed and stored in the multiplication table.

Since the inner, outer and geometric products are linear products, which are distributive over addition, they are products which fulfil the requirements of the product $\diamond$.

So the usage of the multiplication table for the three products (inner, outer and geometric product) is possible, and this is what we use in the Table-Based Approach.

6.2 New Gaalop plugins

We decided to create two new *OptimizationStrategy* classes: TBA and GAPP. The new data flow graph is shown in figure 6.1. Note, that the usage of Maple is still possible. Also we created some new code generators: The *Java Plugin*, which exports Java compilable source and the *GAPP Pretty Printer Plugin*, which creates a human readable source code for a GAPP machine. We use the Java plugin also for our tests. All new plugins and their usage are described in chapter 9.

6.3 Assumptions

We decided to make some assumptions to simplify the transforming process. What follows is a listing of all assumptions:

No multiple assignments

We decided to forbid multiple assignments. To make this decision clear, please take a look at listing 6.1.

Listing 6.1: CLUScript with multiple assignments

```
1 a = e1;
2 b = a^e2;
3 b = a^e3;
4 ?b;
```

There is an assignment to blade $e_1 \wedge e_2$ in line 2. All other blades have a zero coefficient. Line 3 contains an assignment to blade $e_1 \wedge e_3$.

So, what is the contents of the multivector b in line 4? There are two possibilities: $b = e_1 \wedge e_2 + e_1 \wedge e_3$ or $b = e_1 \wedge e_3$.

Which is the correct one? The answer is not clear, since it depends on assumptions for clearing multivectors before overwriting them. To avoid this confusion, we decided to forbid multiple assignments in general.

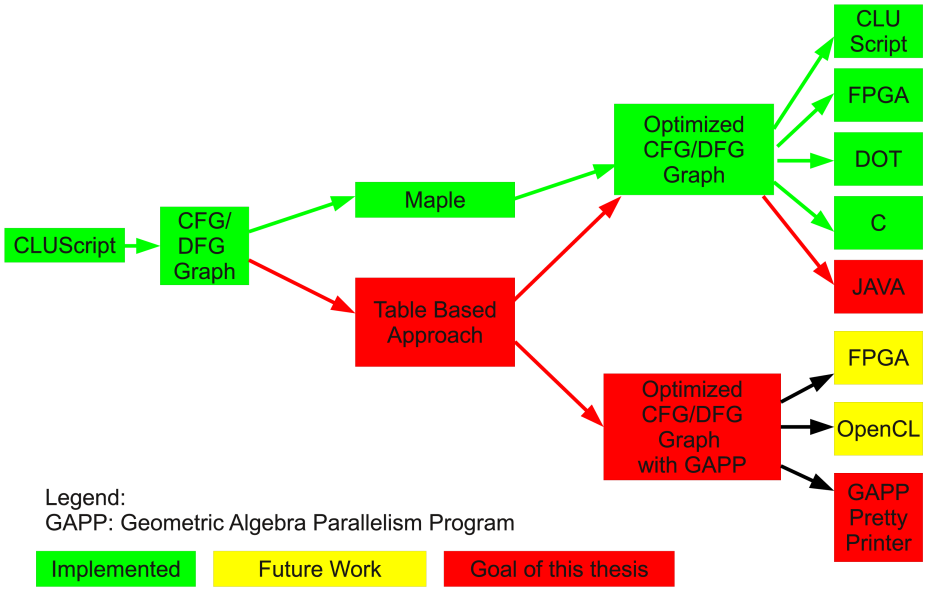


Figure 6.1.: Data Flow Graph of Gaalop with the new plugins

No control flow

We decided to forbid the usage of control flow in the CLUScript. So we do not need any data flow analysis of the CLUScript and optimization process is easier.

Indeed, control flow in CLUScripts is not needed at all, because there is GCD, which handles all control flow externally, and sends only the program parts without control flow to Gaalop.

Scalar operations for math functions

In the current implementation, math functions are allowed only for scalar arguments. We do this for being consistent with the Maple optimization. A correct handling of math functions is a part of future work.

6.4 Integration in Gaalop

To use our Table-Based Approach in Gaalop, we have to create a new *OptimizationStrategy*, which transforms the graph with Geometric Algebra constructs to a graph which does not contain Geometric Algebra constructs any more.

This is done as follows: Recall first, that only programs without control flow are permitted. So, only the *AssignmentNode* instances have to be visited. Every *AssignmentNode* instance has an *Expression* instance, which can contain Geometric Algebra. By using the Visitor Design Pattern, every call of the *visit()* method of a subexpression puts a *MvExpressions* object in a *HashMap*. The HashMap connects an *Expression* object with a *MvExpressions* object.

MvExpressions

An instance of MvExpressions has own expressions for each blade. Think of a statement like in listing 6.2.

Listing 6.2: Sample Assignment

```
a = e1;
```

The resulting MvExpressions object of the base vector e_1 is $\begin{pmatrix}0 & 1 & 0 & \dots & 0\end{pmatrix}^T$. The MvExpressions object of an addition is the sum of its subexpressions.

MvExpressions of products

If a product like the inner product is visited, then at first the two subexpressions are visited. See listing 6.3 for the implementation.

As one can see, the code goes through both MvExpressions objects from subexpressions. If a special blade is not null, then a loop through the outer MvExpressions is started. If both blades are not null, then the result of the product of the two blades, which can be found in the according multiplication table, is queried. If this result is not zero, then the

product of both expression from the two blades and the result from multiplication table is built and stored in the result MvExpressions instance.

Listing 6.3: Implementation of creating MvExpressions for products using the multiplication table

```
/**
         * Calculates the product of two MvExpressions
         * @param typeProduct The type of the product
         * @param left The first factor
         * @param right The second factor
         * @return The product
         */
        private MvExpressions calculateUsingMultTable(Products typeProduct, MvExpressions left, MvExpressions right) {
            MvExpressions result = createNewMvExpressions();

        for (int bladeL = 0;bladeL<bladeCount;bladeL++)
            if (left.bladeExpressions[bladeL] != null)
            {
                for (int bladeR = 0;bladeR<bladeCount;bladeR++)
                    if (right.bladeExpressions[bladeR] != null)
                    {
                        Expression prodExpr = new Multiplication(left.bladeExpressions[bladeL],
                            right.bladeExpressions[bladeR]);
                        Multivector prodMv = usedAlgebra.getProduct(typeProduct,bladeL,bladeR);

                        double[] prod = prodMv.getValueArr();

                        for (int bladeResult = 0;bladeResult<bladeCount;bladeResult++)
                            if (Math.abs(prod[bladeResult])>EPSILON) {
                                Expression prodExpri = new Multiplication(prodExpr,
                                    new FloatConstant((float) prod[bladeResult]));
                                if (result.bladeExpressions[bladeResult] == null)
                                    result.bladeExpressions[bladeResult] = prodExpri;
                                else
                                    result.bladeExpressions[bladeResult] =
                                        new Addition(result.bladeExpressions[bladeResult],prodExpri);
                            }
                    }
            }
            return result;
        }
```

Further progression

If the *AssignmentNode* expression returns from visitation, the resulting MvExpressions is queried in the HashMap. For all blades in MvExpressions, which are not null, a new *AssignmentNode* is created (of course with a multivector component as destination variable) and inserted before the current *AssignmentNode*. After inserting all not null blades, the current *AssignmentNode* is removed from the control flow graph.

6.5 Organization of multiplication tables

For better human reading of multiplication tables, we decided to use a comma-separated-value (CSV) format for storage of multiplication tables. There are two files that must be contained in a directory to describe an algebra. What follows is a description of the file formats.

Blades file

This file must be named “blades.csv” and has the following structure: The file begins with a line which contains all base vectors of the algebra, separated by the character “;”. The next lines contain all blades from the algebra. The order of the blades must be in the canonical order of the base vectors from first line. See listing C.1 in appendix for the blades file for Conformal Geometric Algebra.

Products file

This file must be named “products.csv” and has the following structure: Every line in the file stores three products of a multiplication of two blades. The order of the lines is arbitrary, but for better reading a sorted order is recommended. The data are separated by the character “;” and have the following meaning:

1. Blade of first factor
2. Blade of second factor
3. Inner product of the two blades

4. Outer product of the two blades

5. Geometric product of the two blades

A blade is represented as the index of this blade with a trailing "E". The indices are given by the order of blades listed in the blades file. For the built-in Conformal Geometric Algebra the blades with their indices are listed in table 3.1. If a product is zero, typing a zero is not needed. In this case the date can be empty.

Let us give an example: We choose e_1 and $e_3 \wedge e_\infty$ as two blades. The inner product is zero, the outer and geometric product are $-e_3 \wedge e_1 \wedge e_\infty$.

This yields the following line:

```
E1;E13;;E19;E19
```

See listing C.2 in appendix for a small extract of the large products file for the built-in Conformal Geometric Algebra.

6.6 Implementation aspects

Why using IMultTable?

We define an interface for communicating with the multiplication table. This has the advantage that we are able to support many (different) implementations.

Let us give an example. We want to make Gaalop to support the 16-dimensional Geometric Algebra. So we need to store $3 * 2^{2*16}$ entries. This might be very memory intensive for storage. But if we have the possibility to write an algorithm for calculating the products of two arbitrary blades, we can implement this by implementing the IMultTable interface.

So, by using the interface IMultTable, we have the absolute freedom of implementation. We can fall back on Table-Based Approach or calculate the product at Gaalop runtime.

Of course, calculating the product of two arbitrary blades may lead to a slow performance of compiler, but it can be more efficient than wasting memory for storage of huge tables, especially when using high-dimensional algebras. Note, that Daniel Fontijne described a way for an efficient computation of products in [12].

6.7 Exemplary pass of a CLUScript with Table-Based Approach

Please find the exemplary pass of a CLUScript at appendix B.

7 Optimizations for Table-Based Approach

In this chapter we describe some optimizations of the control flow graph after eliminating Geometric Algebra with the Table-Based Approach.

7.1 Description of some optimizations

Optimizations of a code segment requires the knowledge of the structure of the programming language. CLUScripts permit simple control flow like loops and if-conditions. Since GCD[1] extracts the code from the bodies of loops and if-conditions, we can assume that we only have a single *basic block*[2]. Since all instructions in a basic block are evaluated under any circumstances, optimization is much easier.

This allows the usage of the following optimizations.

Constant Folding Constant folding evaluates operations with constant operators at compile time and replaces the operations with the result of the evaluation. So this operation will not be evaluated at runtime of the generated code and the performance of the generated program increases. Constant folding is done also by Maxima. We implemented Constant folding for those users who do not want to use Maxima.

Constant Propagation Constant propagation searches variables, whose values are known at compile time, e.g. they are the destination of an constant assignment. The reading usage of this variable can be replaced by the constant value. Further application of the constant folding optimization may bring further performance improvement.

Removing unused assignments This is also known as *Dead code elimination*. Since the compiler knows at compile time, which blades of which variables are outputted, the compiler is able to decide, which blades are used for which output blade.

If a destination variable is not used to calculate an outputted blade, it is not needed to perform the assignment and so it can be removed.

A popular way of implementation of the dead code elimination is the *mark&sweep* method. At first, all assignments of outputted blades (in literature, they are called *critical operations*) are marked including all operands to calculate the value of an assignment. After this mark pass, all non-marked operations are removed.

Replace once used expressions This optimization reduces only the code size. It does not increase the performance. If the destination variable of an assignment is used exactly once, this usage can be replaced by the value of the assignment, and the assignment can be removed.

7.2 Applicability of optimizations

All mentioned optimizations can be directly applied to the control flow graph, while assuming that the graph does not contain any control flow.

Even the compiler is in GCD-mode, all optimizations can be applied, because GCD handles the configuration of each subCLUScript. Since only the variables are exported, which are marked with the question mark, the compiler knows all exported blades. All other blades are not used in further code segments. So all mentioned optimizations can be applied in GCD-mode.

7.3 Pragma output

At an output point, denoted by "?aVar;" in a CLUScript code, the contents of the complete multivector is computed.

But in most cases, the user knows the particular blades, which he wants to know. Therefore optionally a pragma line can be added to CLUScript. An example is shown in listing 7.1.

[1] Gaalop Compiler Driver

[2] A basic block is the longest sequence of instructions without control flow.

Listing 7.1: A Pragma line

```
//#pragma output aVar\$1
```

The dollar character is the separator between the name of the variable and its blade index. In the example only the factor of blade with index 1 in the blade list is outputted, when reading a "?aVar;". All other factors are not outputted and therefore they can be seen as dead code, which can be eliminated.

7.4 Maxima

7.4.1 Why Maxima?

Consider a CLUScript like in listing 7.2. All mentioned optimizations applied on this listing will not increase performance. Of course, an optimization routine which optimizes this kind of expressions is implementable. But there are very good tools, which are additionally open-source, that implement these optimizations. So why not use them? One of those is *Maxima*[3].

Running Maxima with this listing results in an output like: $c = 3 * a$. This is the desired optimization. Maxima is able to simplify expressions symbolically.

Note, that the usage of Maxima is optional. This is different to the recent implementation of Gaalop, where the usage of Maple is mandatory.

Listing 7.2: Why using Maxima for further optimization

```
// a is an input variable
?c = 2 * a + a;
```

7.4.2 Interaction with Maxima

For communication with Maxima, we use the *ProcessBuilder* class[4], since there is no DLL available. In the user interface the user may specify a command for the Maxima executable. This path to the executable is used in a ProcessBuilder call.

Note, that Maxima is an open-source project, so the source code is freely available. Maxima is executable on many operating systems. Note also, that the programming language of Gaalop is Java, so both Maxima and Gaalop are platform-independent.

But before calling, a batch file is generated from the ControlFlowGraph.

The first line of the batch file is

```
display2d:false;
```

to make Maxima printing in a LaTEX-like format. After this, the batch file contains all assignment nodes and their expression values. The batch file ends with the line

```
quit();
```

for closing Maxima carefully.

The batch file is stored in the temp-directory of the operating system.

Maxima is executed using the command argument "-b" for batch file reading. The standard output stream is read from Gaalop and stored into a list of strings.

The first two lines and the last line of output can be ignored, since they contain only "batch;", "display2d:false;" and "quit".

Now, the rest of the list is interpreted: Maxima prints "(%i" at the begin of each input line and "(%o" at the begin of each output line. The input lines are ignored and the output lines are splitted by character "=" for variable and expression value string. An *AssignmentNode* instance will be created for each input line with the first part from the splitting operation as variable. The second part of the splitting operation is transformed into an expression graph by using ANTLR[1].

At the end of interpretation, the batch file is deleted.

[3] `http://maxima.sourceforge.net/`

[4] The ProcessBuilder class is a class in the Java API, which runs a command in a shell

7.4.3 What is simplified exactly by Maxima?

Maxima performs a symbolic simplification of the code listed in the batch file, which was created by Gaalop. Gaalop puts the destination variable name and the whole expression of every *AssignmentNode* in the batch file.

To make Maxima remembering preceding variables and their expressions, Gaalop orders Maxima to create an own function for every destination variable. By accessing a created function in an expression, Maxima inserts the preceding expression, which was assigned to the function, automatically. This is done at every *AssignmentNode*, except the destination variable is the exported value of a succeeding *StoreResultNode*.

Let us give an example. Consider a CLUScript like 7.3.

Listing 7.3: Listing for what is simplified exactly by Maxima

```
a = 5*x + 6*y;
b = 2*x - 3*y;
c = 2*x - 3*y;
d = a+b+c;
?d;
```

Since Maxima remembers every variable without question mark, Maxima optimizes to listing7.4.

Listing 7.4: Listing of the output for what is simplified exactly by Maxima

```
d = 9*x;
?d;
```

So the usage of Maxima reduces the number of operations from 11 to 1 operation. This shows a great potential of optimization, when using Maxima.

7.4.4 Discussion

Using Maxima for optimization reminds on the usage of Maple for the recent implementation. Maple was used for transforming the Geometric Algebra algorithm by using the Clifford lib for Maple. One may argue, that Maxima is able to do this, too.

The answer is yes, but there is one important disadvantage by using Maxima or Maple for transforming: The user have to integrate the rules for transforming Geometric Algebra calculations in Maxima or Maple. By using Gaalop with the Table-Based Approach he has only to load the tables into Gaalop, as described in section 6.5. So, using the Table-Based Approach is easier for the user, so we decided to implement this approach.

This leads to the fact, that all Geometric Algebras are allowed in general when using the Table-Based Approach. Theoretically, there are no restrictions on the dimension of the Geometric Algebra. But in practice, the multiplication tables can be very large in higher-dimensional algebras. This is the real disadvantage of using a multiplication table. In this case, one have to integrate real calculations of products of blades, so one has to implement *IMultTable* interface, described in section 6.6.

8 GAPP

This chapter makes extensively use of [15].

At first we describe the GAPP language.

8.1 Description of GAPP

There are six commands in GAPP language. We describe them briefly. [1]

resetMv Creates a multivector with a given number of entries. All entries of the multivector are zero after this operation.

assignMv Assigns values or scalar variables to an existing multivector.

setMv Copies parts of a vector or multivector to a multivector.

assignVector Assigns values or scalar variables to a vector.

setVector Creates a vector out of parts of a multivector.

dotVectors Performs a dot product of at least two vectors and saves the result in a multivector component.

Extra commands

We introduce two extra commands *calculateMv* and *calculateMvCoeff*. For instance, this is necessary to compute the function *sin()*, since the six recent commands do not support this in an efficient way. For understanding this, remind that the computation of the function *sin()* can be done in a series expansion. So, generating GAPP code for the computation of the function *sin()* has the following consequences:

- There are many GAPP instructions to be created and to be stored.
- We oblige the hardware or software, which implements the GAPP language, the way for computation of the function *sin()*. If the hardware has another (more efficient) approach to compute the function, this approach will not be used.

So, we decided to add the two extra commands, which oblige the computation of a function, but not the way of computing the result. So we have a higher abstraction level and the hardware / software has more freedom to implement the computation.[2]

Syntax: **calculateMv** $\mathrm{mv} = \mathrm{op}(\mathrm{mv}_1, \mathrm{mv}_2)$;

This command performs a calculation on two multivectors mv_1 and mv_2 and stores the result in a destination multivector mv.

If an operation is an unary operation, e.g. only one operand is necessary, mv_2 is *null*.

Syntax: **calculateMvCoeff** $\mathrm{mv}_{\mathrm{dest}} = \mathrm{op}(\mathrm{mv}_1, \mathrm{mv}_2)$;

This command performs a calculation on two multivectors mv_1 and mv_2 and stores the result as a blade coefficient *dest* in a destination multivector mv.

If an operation is an unary operation, e.g. only one operand is necessary, mv_2 is *null*.

What follows is a short overview of the operations that might be used in the two commands.

[1] Please find the detailed commands list in [15] Table 1

[2] Remark: The hardware must only implement the operations of *calculateMv/calculateMvCoeff*, which are used in the current CLUScript. It is not mandatory to implement an operation in hardware, if the operation is never used in the CLUScript.

Unary operations
ABS Calculates the absolute value.

ACOS Calculates the arccosine (result is in radians).

ASIN Calculates the arcsine (result is in radians).

ATAN Calculates the arctangent (result is in radians).

CEIL Ceils the operand to the next integer.

COS Calculates the cosine (argument must be in radians).

EXP Calculates the exponentiation of the argument with base e.

FACT Calculates the factorial of an integer.

FLOOR Floors the operand to the next integer.

LOG Calculates the natural logarithm.

INVERT Performs an inversion of a multivector.

SIN Calculates the sine (argument must be in radians).

SQRT Calculates the square root.

TAN Calculates the tangent (argument must be in radians).

Binary operations
DIVISION Performs a division.

EXPONENTIATION Performs an exponentiation.

We do not make any restrictions to the arguments of the math functions (for example *exp* or *cos*). In particular, we allow also whole multivectors as arguments.

Mathematically the result of an exponentiation of a multivector is computed for example by a Taylor series. We decided to make no optimizations, because the GAPP language is only an instruction set for GAPP machines. So we give more freedom and more possibilities for optimizations to the backends (e.g hardware backends), which evaluates the instructions.

Note also, that we do not need to implement logic functions, since the control flow in a *ControlFlowGraph* is not permitted at all.

8.2 The transformation process

Parallel architectures provide computing with high performance. But for taking advantage of parallel computing, the program must be programmed for parallel architectures. CLUScript is a programming language, which does not allow parallel computing itself.

With GAPP as intermediate language, the user is able to use parallel architectures for computing Geometric Algebra algorithms with high performance. As stated in [15], using the Table-Based Approach leads to a scalar multiplication of two vectors with many entries. This scalar multiplication can be done in parallel very efficiently.

Let us give an example. Consider a calculation like

$$S[0] = a_1 b_1 c_1 + a_2 b_2 c_2 + a_3 b_3 c_3 + a_4 b_4 c_4$$

This calculation can be written as

$$S[0] = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

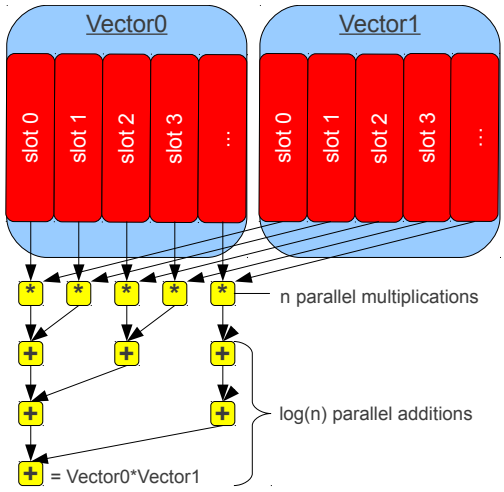


Figure 8.1.: Parallel scalar product of two vectors, as performed by *dotVectors*. This figure is taken from [15].

OpenCL[3] can use this multiplication chain for creating instructions for parallel hardware. A visualisation of calculating scalar products in parallel is shown in figure 8.1. So the transformation process consists of finding and creating scalar multiplications of two or more vectors. Additionally to the GAPP paper, also mathematical functions must be transformed using the extra command "calculateMv".

8.2.1 Description of the transformation process

At first we give a short overview about the stages in the transformation process. All steps will be explained below.

The transformation process stages

1. Elimination of Geometric Algebra with the Table Based Approach and optimizations.
2. Integrating input variables in a special multivector.
3. Grouping expressions to sums and products.
4. Building dot products from some groups.
5. Creating GAPP instructions from dot products and remaining groups.
6. Optimizing GAPP code.

The stages 3-5 are performed on each *AssignmentNode* separately.

Elimination of Geometric Algebra with the Table Based Approach and optimizations

Since the GAPP Plugin is an *OptimizationStrategy*, the input is a *ControlFlowGraph*, which is created by a Code Parser Plugin. This *ControlFlowGraph* contains Geometric Algebra, so the first step is that the TBA Plugin is executed with all of its optimizations. See chapter 6 for details. Additionally we enforce Maxima to give out sum of products by covering every expression with the *expand* command. This leads to a more efficient generated code, as described later.

After executing the TBA Plugin, the real transformation process is started.

Integrating input variables in a special multivector

The GAPP language supports input variables only by assigning them to multivectors. There are two possibilities for realizing this:

[3] `http://www.khronos.org/opencl`

1. Assign input variables when they are needed.

2. Assign all input variables before the first *AssignmentNode*.

The first possibility creates n separate GAPP instructions, if n is the number of input variables. The second possibility creates only one GAPP instruction before the first *AssignmentNode*.

So we decided to choose the second possibility as default. The vector, which holds all input values, has the prefix "inputsVector". Note, that the GAPP plugin arranges all input variables in alphabetical order by their names for aesthetic reasons.

Grouping expressions to sums and products

Recognizing dot products in an expression graph is a difficult task when processing expression elements separately. Our approach is to build groups from separate expressions. These groups have a higher level of abstraction than single expression elements, so recognition is easier.

Please take a look again at the example in section 8.2. The graph of the example and its GAPP transformation process is visualized in figure 8.2. One can see, that the dot product S is a sum of four products. Each product consists of three factors. In general, a dot product is always a sum of at least two products, and each product consists of at least one factor. [4] Remind, that we enforce Maxima to give out sums of products by setting the *expand* command.

So, how to find dot products? As mentioned above, the main idea is to build groups from single expressions:

- A series of at least one *Addition* or *Subtraction* expression is grouped to a *Sum* object.
- A series of at least one *Multiplication* expression is grouped to a *Product* object.

The process of grouping is the task of the class *ExpressionCollector*, which implements the *ExpressionVisitor* interface.

The storage of created groups

The creation of grouping elements is a nice approach for making recognition easier, but how can we store these groups?

A first approach is to declare *Sum* and *Product* to subclasses of *Expression*, because then they can be stored in-place at the right position in the expression graph. But this (first) approach has a grave disadvantage: The *Expression* class declares an abstract method *accept()*, because this is a required part of the Visitor Design Pattern. So this method must also be implemented by *Sum* and *Product*. To implement them with an empty body is not a good idea, because even a simple *Pretty Printer Visitor* for expressions produces no output, when reaching these two classes. To implement them by calling the given *ExpressionVisitor* instance is also not a good idea, because then one has to add a new *visit()*-method in the *AbstractVisitor* and this results in a necessary modification of all ExpressionVisitor implementations.

Due to the non-applicability of the first approach, we developed a second approach for storage: We create a new data structure, which is used internally from the GAPP *OptimizationStrategy* plugin.

The ParallelObject data structure

The ParallelObject data structure consists of following classes:

Constant Represents a constant with a float value.

DotProduct Represents a dot product. The dot product is internally organized as *LinkedList of ParallelVector* instances.

ExtCalculation Represents an extended calculation. It contains a type which extends the *MathFunction* types with the *division* and *invert* operation.

ParVariable Represents a variable.

MvComponent Represents a multivector component.

Product Represents a product of ParallelObject instances.

Sum Represents a sum of ParallelObject instances.

[4] A dot product of two products, where each product consists of one factor, degenerates to a simple addition

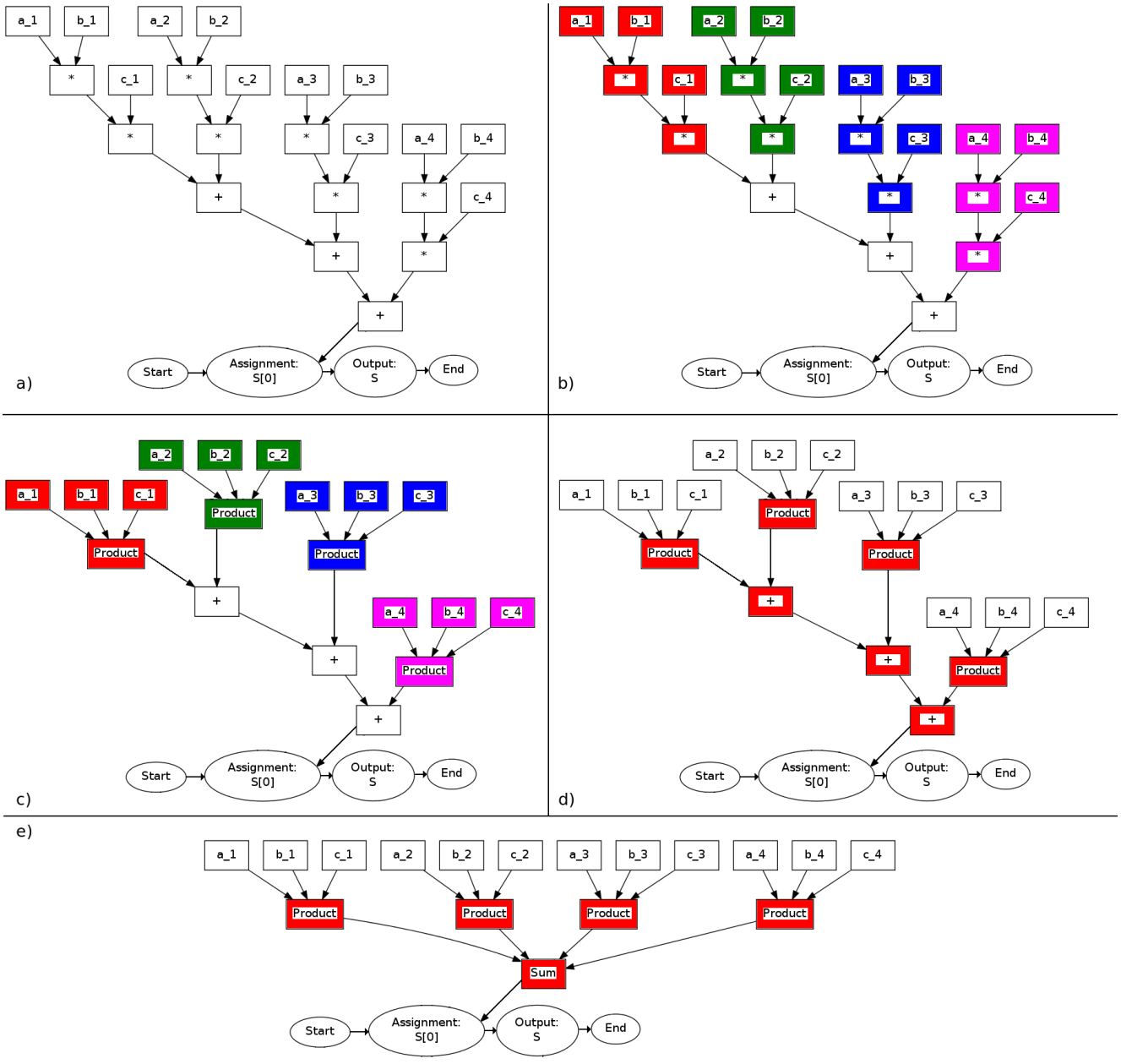


Figure 8.2.: Visualisation of the GAPP transformation for the example in section 8.2. a) graph. b) graph with marked factors. c) graph with marked products generated from previous marked factors. d) graph with marked summands. e) graph with marked sums generated from previous marked summands.

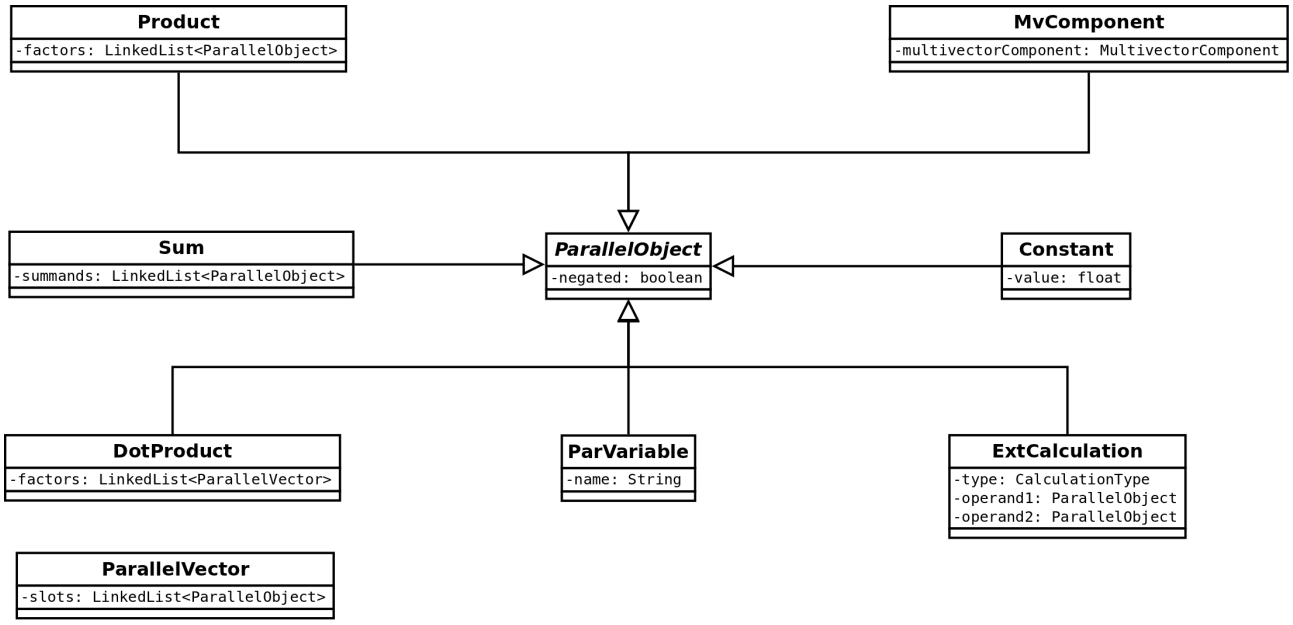


Figure 8.3.: The ParallelObject data structure

All classes are subclasses of the abstract class *ParallelObject*. Of course, we include a *accept()*-method for using the Visitor Design Pattern. The according *AbstractVisitor* is the *ParallelObjectVisitor* interface. For grouping subtractions to a sum, we include an attribute *negate* in the ParallelObject class.

Please find the UML[5]-diagram of the ParallelObject data structure in figure 8.3.

The remaining grouping process

With the ParallelObject data structure, every expression graph can be easily grouped. Note, that Negation expressions can be transformed by inverting the *negate* attribute. Please find for the example in section 8.2 the visualisation of the ParallelObject data structure in figure 8.2e.

While grouping summands to sums and factors to products, the *ExpressionCollector* is called again for handling subexpressions. Since every *visit()*-method returns a new ParallelObject instance which represents the transformed expression, calling the *ExpressionCollector* at first time returns the ParallelObject which represents the whole expression in the new data structure. This returned value is stored for using in the next stages.

Building real dot products from some groups

In the previous stage, we have transformed the expression graph into the ParallelObject data structure. The task of this stage is to create real[6] dot products in the ParallelObject data structure.

Finding the start points of dot products is the task of the class *DotProductsFinder*, the insertion behoves to the class *DotProductCreator*. Both classes implement the *ParallelObjectVisitor* interface.

If a called *visit()*-method in *DotProductsFinder* returns not-null, then it is a DotProduct instance and this must be used for replacement of the recent ParallelObject which is called.

The only method, which returns a value that is not-null is the *visitSum()*-method, since a real dot product starts always with a *Sum* instance.

If the call of the *DotProductsFinder* on the root ParallelObject returns a non-null object, the root ParallelObject is replaced with the returned object.

[5] Unified-Modeling-Language

[6] We define a *real* dot product as a dot product, where the vectors have more than one slot. If the vectors of a dot product have only one slot, the dot product degenerates to a product.

Applying the described method to the sum and products given in our example (visualized in 8.2e) yields the equation from above:

$$S[0] = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

Creating GAPP instructions from dot products and remaining groups

In the previous stage, we have created dot products, so we can now use the ParallelObject data structure to create the GAPP instructions, which is the task of the class *GAPPCreator*.

This stage creates new temporary variables. For this reason, a set of all variable names, that are used in the ControlFlow-Graph, is a parameter in the constructor. With this set of used variable names, it is safe to create new temporary variables without overwriting others, since the new variables names are unused before transformation.

In the GAPPCreator class, the parameter *arg* can hold a *GAPPMultivectorComponent* to indicate the destination of the called operation. Every *visit()*-method returns the destination, if such one is created in the method.

Product

An instance of a Product is transformed into an instance of a DotProduct. The GAPPCreator is then applied to the new DotProduct. The *visitProduct()* returns the result of the *accept()*-method call.

Constant

Constants have to be assigned to the destination multivector by using the *assignMv* instruction in the GAPP language. The *negate* attribute can be easily handled by negating the selector.

MvComponent

MvComponents have to be assigned to the destination multivector by using the *setMv* instruction in the GAPP language. Also here, the *negate* attribute can be easily handled by negating the selector.

ExtCalculation

ExtCalculation has also a companion piece in the GAPP language. So transformation here is trivial.

Note, that also *ExtCalculations* can be negated. To transform the *negate* attribute the only way is to create a new temporary variable, which is the destination of the ExtCalculation. In a second step the new temporary variable is the source for a *setMv*-instruction, where all selectors are inverted. The destination of the *setMv*-instruction is the recent destination.

DotProduct

The first step of transforming a DotProduct is to make sure, that all inner elements are terminals [7] , if an element is not a terminal, a new assignment to a new temporary variable will be created and a MvComponent will be put into DotProduct.

The second step of transforming a DotProduct is to create the vectors for the *dotVectors*-operation. The only types of ParallelObject, that may occur in a DotProduct, are the terminals Constant and MvComponent. Since we want to create an optimal code, we distinguish four constellations in a vector:

1. Only MvComponents from one multivector
2. Only Constants
3. Only MvComponents from more than one multivector
4. A mix of all two types

Case 1: Only MvComponents from one multivector

The transformation of one multivector is simple, since only one *setVector*-instruction must be created. The blade indices of the MvComponents correspond to the selectors of the *setVector*-instruction. If the MvComponent is negated, then the according selector is also negated.

[7] A terminal is here a ParallelObject which has no children, e.g. Constant, MvComponent, ParVariable

Case 2: Only Constants

Also the transformation of only constants is simple. At first, a new temporary multivector is created. This is filled by using the *assignMv*-instruction. The indices of the Constant instances correspond to the selectors of the *setVector*-instruction. If the Constant is negated, then the according selector is also negated.[8] As next step, the created multivector must be copied to vector by using the *setVector*-instruction.

Case 3: Only MvComponents from more than one multivector

To transform MvComponents from more than one multivector, a new temporary multivector is created at first. As next step, all different names of MvComponents, which occur in the ParallelVector, are collected in a set of strings. Then this set of string is used for iterating. In every iteration, a new *setMv*-instruction is created and the selectors of it are filled with the indices of the ParallelVector, where the name of the MvComponent is equal to the current iteration string. As last step, the created multivector must be copied to a vector by using the *setVector*-instruction.

Case 4: A mix of all two types

To handle a mix of all types, at first all constants will be transformed like in case 2. As next step, all MvComponents are transformed like in case 3. As last step, the created multivector must be copied to a vector by using the *setVector*-instruction.

Optimizing GAPP code

We built in some optimizations of the generated GAPP code. The optimization process is called after the generation of the GAPP instructions. What follows is a description of the build-in optimizations.

Removal of some assignMv instructions

Some generated *assignMv*-instructions assign the value 0 to a multivector component. Since every multivector is reset by *resetMv* before it is used, these assignments waste performance. The implemented optimization removes all values in the *assignMv*-instructions, if they assign the value 0. If there are no values left in the *assignMv*-instruction, it is removed.

Merging of some setMv instructions

Sometimes several *setMv*-instructions can be merged together, because they have the same source vector or multivector and the same destination multivector. This case occurs because the TBA plugin produces an own *AssignmentNode* for every blade of a multivector component. However, the GAPP code is created per *AssignmentNode*.

For an example, see line 25 in listing 9.3.

So, how to find the *setMv*-instructions for merging? We use the *CFGGAPPVisitor* for walking through the graph. Also we use a map which maps a string to a list of *GAPPSetMv* instructions. When visiting an *AssignmentNode*, the following operations are performed:

- If the variable name of the last *AssignmentNode* destination does not equal the variable name of the current *AssignmentNode*, the procedure *prepareMap()* is called and then the map is cleared.
- The included GAPP instructions are visited.
- The next SequentialNode is visited.

The procedure *prepareMap()*: The procedure *prepareMap()* iterates through the values of the map. In every iteration the list is investigated. If the list has more than one element, all GAPPSetMv instructions are mergeable. So the selectors of all list entries, except of the first element, are integrated in the first *setMv*-instruction. The other *setMv*-instructions are removed.

Every *setMv*-instruction is put to the map, when it is visited. If the destination name already exists, this *setMv*-instance is appended to the end of the list.

[8] Note that negation of constants is also possible negating the value itself.

9 Additional work and user guide for the new Gaalop plugins

In this chapter we describe the new features of Gaalop and their usage.

9.1 TBA plugin

9.1.1 General compilation

To compile with usage of the Table-Based Approach, load or type in your CLUScript file, choose the TBA-Plugin in the ComboBox and press the “Optimize” button for choosing the desired code generator plugin. After the compilation process, the output is shown automatically. See figure 9.1 for a screenshot of Gaalop with TBA usage.

9.1.2 Switching the algebra

The Conformal Geometric Algebra is set as the algebra being used by default. If you want to compile the CLUScript with another algebra, you have to take two actions:

1. Insert at the beginning of the CLUScript a line which contains your algebra dimension in the following format:

   ```
   //#pragma algebraDimension n
   ```

   where *n* is the dimension of the algebra to use.

2. Move to the configuration panel by clicking the “Configure” button and move to the “TBA” register pane. Type in the textfield “algebra” the path to the directory, which contains the three algebra definition files. If the text of the textfield equals “5d”, then the build-in multiplication table for the Conformal Geometric Algebra is used. See section 6.5 for a description of the definition files.

Note that at this moment, arbitrary algebras are not parseable by CLUCalc Parser. So, only 2d, 3d and the 5d algebra are allowed. The reason is, that Gaalop was not designed for choosing the algebra at Gaalop runtime. The Table-Based Approach is able to work with arbitrary algebras at Gaalop runtime. In section 10.3 we demonstrate this feature. Changing the CLUCalc Parser, so that it supports arbitrary algebras at Gaalop runtime is not trivial. This is a part of future work.

9.1.3 Other compilation options

To make the user controlling the compile process, there are some other options included in the configuration panel. See figure 9.1.3 for a screenshot of the TBA register in the configuration panel. What follows is a description of the options.

Maxima usage
By default, Maxima usage is disabled, because it is optional. For those users, who want to use Maxima, the usage Maxima can be enabled by clearing the “optMaxima” check box. The path to the executable of Maxima, which is used to invoke Maxima, is automatically set for Windows and Linux operating systems. If the user uses another operating system or has installed Maxima in a different path, he can change the Maxima command in the according text field.

One expression removal optimization
With the checkbox “optOneExpressionRemoval” the user can control, if the compiler should remove those assignments, which destination variable is used only at one time. By default, this optimization is enabled.

Constant propagation optimization
With the checkbox “optConstantPropagation” the user can control, if the compiler should do a constant folding and constant propagation. This evaluates calculations with constant operands by compile time. By default, this optimization is enabled. When activating Maxima, this optimization is senseless, because Maxima has optimizes with the Constant propagation too. The Constant propagation optimization is implemented for those users, who do not want to use Maxima.

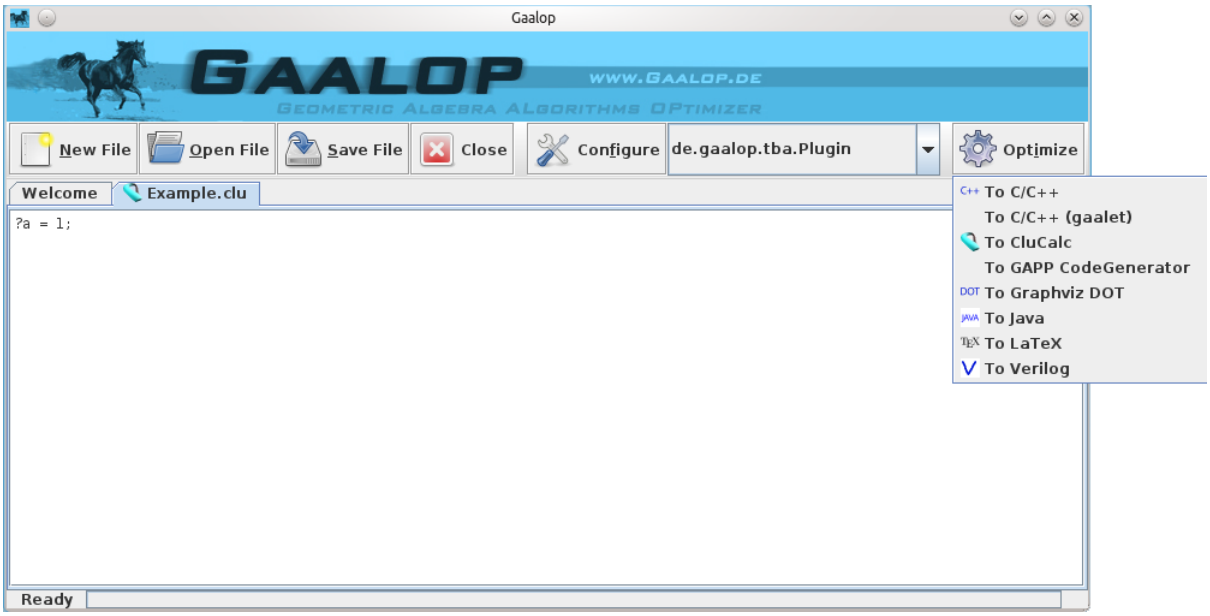


Figure 9.1.: Screenshot of Gaalop with TBA usage

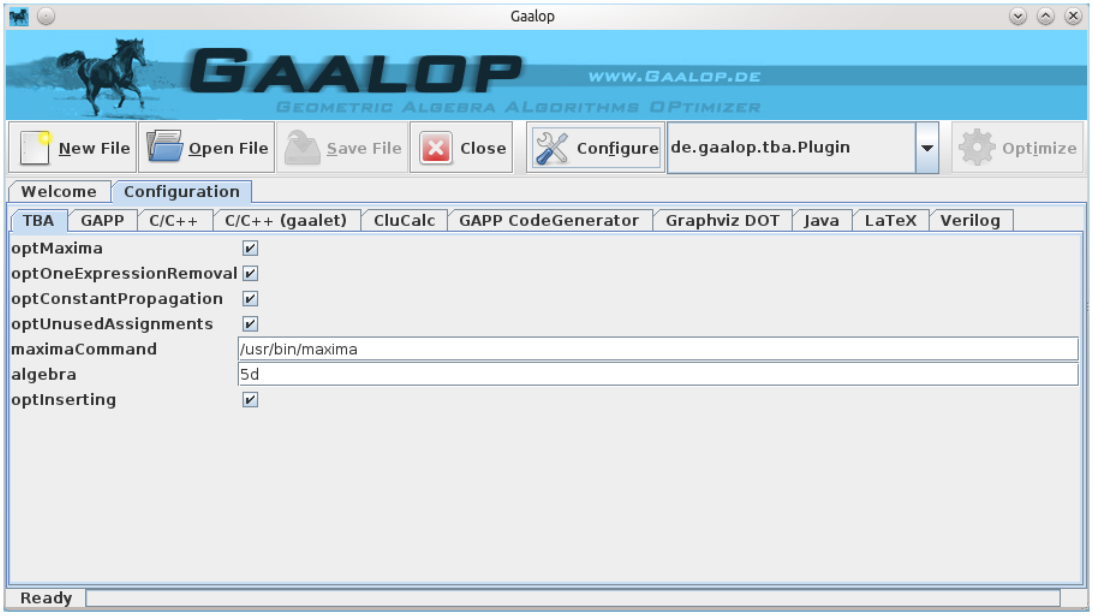


Figure 9.2.: Screenshot of the TBA register in the configuration panel of Gaalop

Unused assignments removal optimization

With the checkbox “optUnusedAssignments” the user can control, if the compiler should remove unused assignments. An assignment is unused, if its destination is never read later. Then it can be removed. By default, this optimization is enabled.

Inserting non-marked expressions

With the checkbox “optInserting” the user can control, if the compiler make Maxima remember previous equations. This enables Maxima to optimize over many AssignmentNodes, except for the case that a variable which is marked with the question mark "?". By default, this optimization is enabled.

9.2 Java plugin

In the recent implementation of Gaalop, a plugin for generating Java code hasn't been implemented yet. We implemented such a code generator also for testing our implementation of Table-Based Approach.

As a contribution to a good design for projects of users, a CLUScript compilation results in two output files: The “GAProgram” interface and the calculation class with the CLUScript name as class name.

The “GAProgram” interface provides all methods which can accessed from other classes to query calculation results. See listing 9.1 for the interface source. Please find also the description of all methods in the JavaDoc in this listing.

Listing 9.1: Source of interface GAProgram

```
1
2
3  import java.util.HashMap;
4
5  /**
6   * Performs the calculations specified in a Geometric Algebra Program
7   */
8  public interface GAProgram {
9
10     /**
11      * Performs the calculation
12      */
13     public void calculate();
14
15     /**
16      * Returns the value of a variable
17      * @param varName The variable name, specified in the Geometric Algebra program
18      * @return The value of the variable with the given name
19      */
20     public float getValue(String varName);
21
22     /**
23      * Sets the value of a variable
24      * @param varName The variable name, specified in the Geometric Algebra program
25      * @param value The value
26      * @returns <value>true</value> if the setting was successful, <value>false</value> otherwise
27      */
28     public boolean setValue(String varName, float value);
29
30     /**
31       * Returns all values in a map name->value
32       * @return The map which contains all values
33       */
34      public HashMap<String,Float> getValues();
35
36 }
```

Integration of Geometric Algebra in your project

For integration of Geometric Algebra algorithms in your project, you can use both generated files. Sometimes an algorithm in Geometric Algebra in your project is developed later or by another team member. Also this is not a problem: The developer of the project can use the GAProgram interface for implementing the access and the concrete implementation can be added later easily, because the generated Java computation class implements the interface.

Important hint

The dollar sign is used to separate the variable name and the blade index. The choice of the dollar sign as separator is a little bit of confusing. The reason is that the usage of a dollar sign in a CLUScript identifier is not allowed, but in a Java identifier it is permitted.

9.3 GAPP plugin

To make Gaalop compile the CLUScript in GAPP Code, choose the GAPP Plugin from the checkbox and press the “Optimize” button for choosing the desired code generator.

At the moment, only the *GAPP Pretty Printer Plugin* is available.

After the compilation process, the output is shown automatically.

9.4 GAPP Pretty Printer plugin

The GAPP Pretty Printer plugin creates human readable source code for a GAPP machine from a *ControlFlowGraph* which is decorated with GAPP members. Every *AssignmentNode* in the *ControlFlowGraph* creates one paragraph in the generated source. A comment is inserted at the beginning of every paragraph, which contains a representation of the *AssignmentNode* in *C* code style.

With this comment, the user can simply trace the program flow without the need of understanding all GAPP instructions. Listing 9.3 shows the generated code for a CLUScript, which is shown in listing 9.2. Note, that for instance in line 3 of listing 9.2, a multivector c is created with 8 empty entries, because a $3d$ algebra is used.

Listing 9.2: Source of an CLUScript for generating GAPP code

```
1 //#pragma algebraDimension 3
2
3 ?a=a1*e1+a2*e2+a3*e3;
4 ?b=b1*e1+b2*e2+b3*e3;
5 ?c=a*b;
6 d=a+c;
7 ?f=a^d;
```

Listing 9.3: Source of generated GAPP code from code listed in 9.2 in a pretty printed format

```
//a[1] = inputsVector[0]
assignVector inputsVector = [a1,a2,a3,b1,b2,b3];
resetMv a[8];
setMv a[1,2,3] = inputsVector[0,1,2];

//a[2] = inputsVector[1]

//a[3] = inputsVector[2]

//b[1] = inputsVector[3]
resetMv b[8];
setMv b[1,2,3] = inputsVector[3,4,5];

//b[2] = inputsVector[4]

//b[3] = inputsVector[5]

//c[0] = ((a[3] * b[3]) + (a[2] * b[2])) + (a[1] * b[1])
resetMv c[8];
setVector ve0 = a[3,2,1];
setVector ve1 = b[3,2,1];
dotVectors c[0] = <ve0,ve1>;

//c[4] = (a[1] * b[2]) - (a[2] * b[1])
setVector ve2 = a[1,-2];
setVector ve3 = b[2,1];
dotVectors c[4] = <ve2,ve3>;

//c[5] = (a[1] * b[3]) - (a[3] * b[1])
setVector ve4 = a[1,-3];
setVector ve5 = b[3,1];
dotVectors c[5] = <ve4,ve5>;

//c[6] = (a[2] * b[3]) - (a[3] * b[2])
setVector ve6 = a[2,-3];
setVector ve7 = b[3,2];
dotVectors c[6] = <ve6,ve7>;

//f[1] = a[1] * c[0]
resetMv f[8];
setVector ve8 = a[1];
setVector ve9 = c[0];
dotVectors f[1] = <ve8,ve9>;

//f[2] = a[2] * c[0]
setVector ve10 = a[2];
setVector ve11 = c[0];
dotVectors f[2] = <ve10,ve11>;

//f[3] = a[3] * c[0]
setVector ve12 = a[3];
setVector ve13 = c[0];
dotVectors f[3] = <ve12,ve13>;

//f[4] = 0

//f[5] = 0

//f[6] = 0

//f[7] = ((a[1] * c[6]) - (a[2] * c[5])) + (a[3] * c[4])
setVector ve14 = a[1,-2,3];
setVector ve15 = c[6,5,4];
dotVectors f[7] = <ve14,ve15>;
```

10 Tests

Testing is always a very important development step in software engineering.

Proving complete correctness is not possible for software by only creating tests. However, with tests, you can discover many implementation errors.

10.1 The test framework

Having a good framework for testing, creation of new tests will be easier.

We create an interface *GenericTestable*, according to listing 10.1

Listing 10.1: Generic Testable source

```
1  package de.gaalop.tba;
2
3  import java.util.LinkedList;
4
5  /**
6   * Defines an interface for a test
7   */
8  public interface GenericTestable {
9
10     /**
11      * Returns the CLUScript to be used for testing
12      * @return The CLUScript
13      */
14     public String getCLUScript();
15
16    /**
17     * Returns the InputOutputs for testing
18     * @return The InputOutputs
19     */
20     public LinkedList<InputOutput> getInputOutputs();
21
22     /**
23      * Returns the used algebra in test program
24      * @return The used algebra
25      */
26     public UseAlgebra getUsedAlgebra();
27
28 }
```

Additionally, we create a TestCreator class, where all implementations of GenericTestable are listed. Running the TestCreator class builds Java test classes, which then can be compiled and tested for example by the IDE.

To create a new test, everything one has to do is to implement this interface, add it to the TestCreator class and run the TestCreator class.

10.2 Our tests for table-based approach

To prove a partial correctness of our table-based approach, we create a number of scenarios. There are two types of scenarios: The accept-scenarios and the fail-scenarios.

10.2.1 The accept-scenarios

Accept-scenarios are created to test, if the compiler do, what it should. By compiling, no error occurrence is permitted. Only warnings are permitted.

What follows is a short description of some tested scenarios:

Circle
The circle test is described in section 3.4. What follows is a short enumeration of the tested problem instances.

No variables A CLUScript will be compiled with the points (5,2), (3,9), (6,4) as input. So there are no variables.

One variable Like “No variables” but now there is one coordinate which is a variable and 50 different values are tested on compiled Java source.

Only variables All coordinates of the points are variables and so the points (5,2), (3,9), (6,4) are tested.

GPS

The well-known Global-Positioning-System is used for determining the position on earth. Four satellites, which send signals, must be in the reception area of the GPS receiver. All satellites send their exact position and time, so the receiver can calculate its position.

We use a simplification of this scenario for a test of our compiler: We assume that we know the distance from GPS receiver to each satellite. Since the the transit time of the signal is now not used, only three satellites are needed.

The position of the GPS receiver is now the intersection of three spheres, whose centre points are the satellites and radius are the distances.

Like in circle-test, there are several problem instances:

No variables A CLUScript will be compiled with the points (1,1,1), (0,0,1), (0,1,0) and the distances 0.6, 0.7, 0.5 as input. So there are no variables.

Only variables All coordinates of the points and distances are variables and so the points (1,1,1), (0,0,1), (0,1,0) and distances 0.6, 0.7, 0.9 are tested.

10.2.2 The fail-scenarios

Fail-scenarios are created to test, if the compiler reports errors by compiling CLUScripts with errors. If no error occurs, the test fails.

What follows is a short description of tested scenarios

Multiple Assignments test

Multiple assignments to one variable is not permitted. The reason for this is described in 6.3. Before TBA optimizes the code, it is checked, if there are no multiple assignments. Making sure, this check is done correctly, this test was created.

The code which is test, is listed in 10.2.

Listing 10.2: CLUScript with multiple assignments which should report errors in a compile process

```
a = e1;
b = a^e2;
b = a^e3;

d = a^e3;
?b;
?d;
```

Control Flow

Control flow is not permitted, too.

We test this with a CLUScript shown in listing 10.3.

Listing 10.3: CLUScript with control flow which should report errors in a compile process

```
a = 1;
c = 4;
if (a == t) {
  a = 2;
} else {
  a = 3;
}
b = a + 1;
?a;
?b;
?c;
```

10.3 Tests for other algebras

The table-based approach can be used with other Geometric Algebras than Conformal Geometric Algebra. To prove this ability, we have tested a compile run with some test cases.

For other algebra, the user must add a

```
//#pragma algebraDimension 3
```

line to the CLUScript. Here the dimension of the algebra is 3.

This must be done because the CLUCalc parser instantiates the algebra mode to N3. Using this pragma, the algebra mode can be changed easily.

What follows is a description of the test cases.

10.3.1 2d Geometric Algebra

For testing the two-dimensional Geometric Algebra we calculate the product of two complex numbers. The imaginary unit can be represented by $i := e_1 \wedge e_2$, since $i^2 = (e_1 \wedge e_2) * (e_1 \wedge e_2) = -1$. For a more detailed description on complex numbers in Geometric Algebra, we refer to [16].

You may argue, that this computation is also possible with the Conformal Geometric Algebra. This is true, so we calculate the dualization of the product of two complex numbers. This dualized multivector must not contain $e0$. If it contains $e0$, then a wrong algebra is used.[1] For checking the dualization we dualize three times more and this should give the original product.[2] See listing 10.4 for the test source and listing 10.5 for the compiling results.

Listing 10.4: Geometric Algebra algorithm for testing the 2d space

```
//#pragma algebraDimension 2
i = e1^e2;

a = 5 + 6*i;
b = 3 + 4*i;

?c = a*b;

?cdual = *(c);
?cd = *(*(*(cdual)));
```

Listing 10.5: Output of compiling listing 10.4

```
      c$0 = (-9.0f); // 1.0;
      c$3 = 38.0f; // e1^e2;
      cdual$0 = 38.0f; // 1.0;
      cdual$3 = 9.0f; // e1^e2;
      cd$0 = (-9.0f); // 1.0;
      cd$3 = 38.0f; // e1^e2;
```

10.3.2 3d Geometric Algebra

For testing the three-dimensional Geometric Algebra, we take an example from [15]. See listing 10.6 for the test source and listing 10.7 for the compiling results.

Note, that in contrast to the calculation of the example in the paper, that the low-level mathematic simplification of Maxima simplifies f$4, f$5 and f$6 to zero.

Listing 10.6: Geometric Algebra algorithm for testing the 3d space

```
//#pragma algebraDimension 3

a=a1*e1+a2*e2+a3*e3;
b=b1*e1+b2*e2+b3*e3;
?f=a^(a+a*b);
```

[1] The dual of a multivector is calculated by multiplying $-I$ in 2d, 3d and Conformal Geometric Algebra. I is the pseudoscalar. It is defined as the blade with the highest *grade*. The grade of a blade is equal to the number of base vectors in a blade.

[2] Note, that at this moment it is not possible to prove that TBA supports other algebras this way, because the dualization operation is replaced in the CLUCalcParser Plugin. One way to prove the support for other algebras at this moment is to investigate the indices of outputted multivectors. For instance, you see in line 2 of listing 10.5 the index 3 for the blade $e1 \wedge e2$. This is true only for the two-dimensional algebra, because in the 5-dimensional algebra the blade $e1 \wedge e2$ is located at index 6 (see table 3.1).

Listing 10.7: Generated Java code from compiling listing 10.6

```
f$1 = (a1$0 * ((((a3$0 * b3$0) + (a2$0 * b2$0)) + (a1$0 * b1$0)))); // e1;
f$2 = (a2$0 * ((((a3$0 * b3$0) + (a2$0 * b2$0)) + (a1$0 * b1$0)))); // e2;
f$3 = (a3$0 * ((((a3$0 * b3$0) + (a2$0 * b2$0)) + (a1$0 * b1$0)))); // e3;
f$4 = 0.0f; // e1^e2;
f$5 = 0.0f; // e1^e3;
f$6 = 0.0f; // e2^e3;
f$7 = (((a1$0 * (((a2$0 * b3$0) - (a3$0 * b2$0)))) - (a2$0 * (((a1$0 * b3$0) - (a3$0 * b1$0))))))
       + (a3$0 * (((a1$0 * b2$0) - (a2$0 * b1$0))))); // e1^e2^e3;
```

11 Results using Table-Based Approach / GAPP & optimizations

11.1 Results using Table-Based Approach & Optimizations

11.1.1 General improvements

We want to emphasise, that Maple is not needed for the transformation process any more, because the Table-Based Approach has supplanted it. However, Maple is still usable for transformation. Note also, that all recent code generators are still usable. Furthermore, we implemented the a first important step for supporting arbitrary algebras.

11.1.2 Performance measure

To measure a trend of the performance of the generated code by TBA and the optimizations, we use the project of Molecular Dynamics. Please find further informations about this project in [19].

The project of Molecular Dynamics contains many solvers. The task of the solvers is to compute a new state of all molecules from a given state. There are solvers implemented in the project of Molecular Dynamics, that use only the CPU with conventional approaches. Patrick Charrier has included a solver that uses Geometric Algebra. He uses the *OpenCL* backend to create the source for the solver.

His recent solver uses Maple to compile the Geometric Algebra algorithms. With our implementation he has compiled the Geometric Algebra algorithms without the use of Maple.

After compilation, the simulation needs the solve times, that are shown in figure 11.2 and figure 11.1. One can see, that using TBA without Maxima instead of Maple reduces the solving time by 49.7%. Using TBA with Maxima instead of Maple reduces the solve time by 50.2%. So we see, that using TBA is very effective itself. The usage of Maxima increases performance only slightly in this example. But is important to note, that the usage of Maxima can bring great performance improvements. See section 7.4.3 for details. So we recommend strongly the usage of Maxima.

11.2 Results using GAPP

Measuring the performance of the generated GAPP code is more difficult because no hardware implementation for the GAPP language at this moment. So we are not able to give any benchmarks. Additionally, the performance depends extremely on the hardware configuration. The creation of hardware solutions that use GAPP is part of the future work.

But we see for example in the listing 9.2 and listing 9.3, that the application of our algorithm to transform CLUScript code to GAPP code results in a program, which has a great performance on parallel architectures.

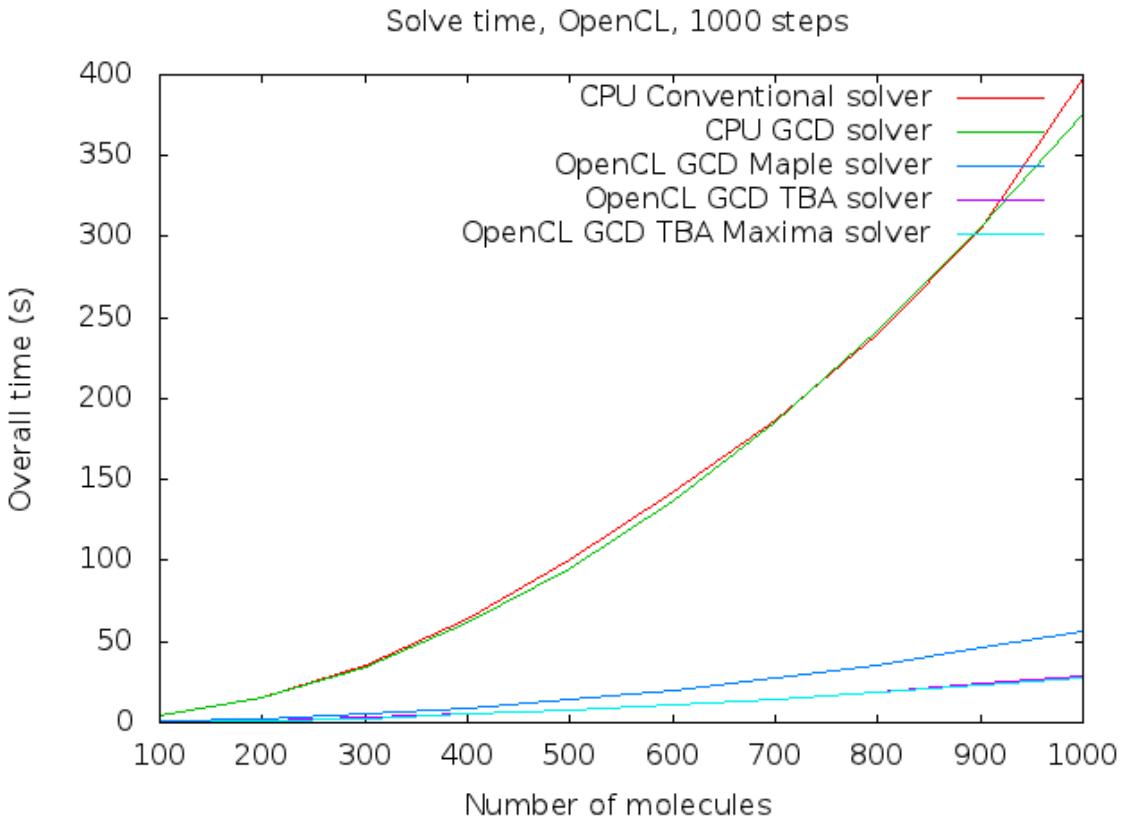


Figure 11.1.: Solve times for Molecular Dynamics [19] for all solvers. Image courtesy of Patrick Charrier.

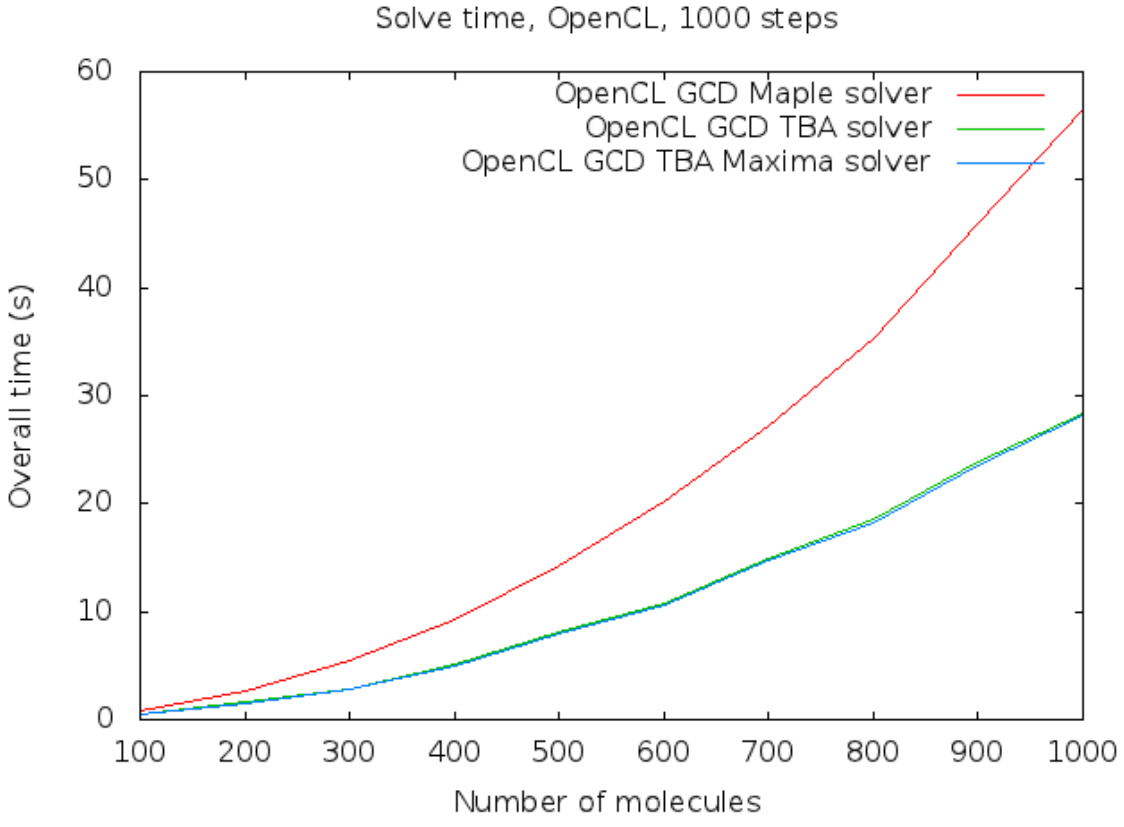


Figure 11.2.: Solve times for Molecular Dynamics [19] for solvers with TBA. Image courtesy of Patrick Charrier.

12 Conclusion

It was a leading issue of this work to eliminate the use of Maple and making Gaalop able to generate code for parallel architectures. We have shown a way to implement the transformation process. This makes Gaalop a powerful tool for creating high performance applications. Because we have eliminated the use of Maple, the new version of Gaalop is not depending on any commercial software.

Exemplary, we present Gaalop in action in the project of Molecular Dynamics, where we reached a speed up of the factor two. Also we built in a support for higher dimensional algebras.

The integrated tool for generating GAPP code simplify the creation of efficient algorithms on parallel architectures.

In summary, we see great potential for the future. Because Gaalop is well-designed, new plugins can be integrated easily. We hope that Gaalop in its current version will be helpful for running the generated code with great performance on parallel architectures while profiting of the elegance of Geometric Algebra.

13 Future Work

Code generators

To benefit of the parallelism of GAPP, one has to create code generators which create parallel programs. This can be done for example on FPGA or in OpenCL code. The future work is to create such code generators which use the parallelism by the GAPP plugin.

Make CLUCalc Parser plugin supporting other algebras

As mentioned in section 9.1.2, the CLUCalc Parser plugin does not support other algebras currently. The future work is to modify the CLUCalc Parser so that it supports other algebras.

Integrate higher dimensional algebras

We have shown, that the table-based approach permits the easy usage of other algebras at Gaalop runtime. The future work is to integrate for example the nine-dimensional geometry algebra, which has been introduced in [20]. GCD is programmed for the usage with the Conformal Geometric Algebra. So GCD has to be modified, when using higher dimensional algebras.

Extend math functions for arbitrary multivectors

In the current implementation, math functions are allowed only for scalar arguments in the TBA plugin. We do this to be consistent with the Maple optimization. But many math functions are operable on arbitrary multivectors, for example the exp function. So the correct handling of multivectors is to be done as a future work.

Geometric Algebra Computers

GAPP represents the operations, which are needed for a calculation. These operations are instructions. The GAPP language is an instruction set for processors. The vision is to develop a processor, which instruction set is the GAPP language.

Creating a program for creation of multiplication tables

Creating manually multiplication table can be very hard. So it would be nice having a program, which automatically calculates the products which will be stored in the multiplication tables. As an input, one may choose the dimension of the algebra and their axioms. For further information we refer to [17]. Note, that is easy to upgrade Gaalop, so that it computes the product of two blades at Gaalop runtime. See section 6.6 for details on upgrading Gaalop. This saves memory especially for higher algebras.

Storage of the product tables

For easy reading and creation, the current multiplication table is in a simple human readable format. For supporting higher dimensional algebras in an efficient way, the format of the table entries must save more memory.

Further optimizations of the DotVectors

The generated GAPP code has still potential for optimization. For instance, the generation of GAPP code from dot products can be optimized further. The elements in a DotProduct can be reordered in every row for reducing the number of *setMv*-commands. Consider a DotProduct like

$$\begin{pmatrix} a_1 \\ c_2 \\ b_3 \end{pmatrix} * \begin{pmatrix} b_1 \\ a_2 \\ c_3 \end{pmatrix} * \begin{pmatrix} c_1 \\ b_2 \\ a_3 \end{pmatrix}$$

The creation of the vectors requires 9 *setMv*-commands because every vector has 3 different names.

An example for an optimal order is

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} * \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} * \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

With this order only 3 *setMv*-instructions are necessary. So we save 6 instructions and this increases performance.

However, finding optimal orders is not trivial for arbitrary constellations. So, the future work is to develop an efficient algorithm for finding an optimal order, if such exists. If not, a greedy approach can be developed for finding a good solution.

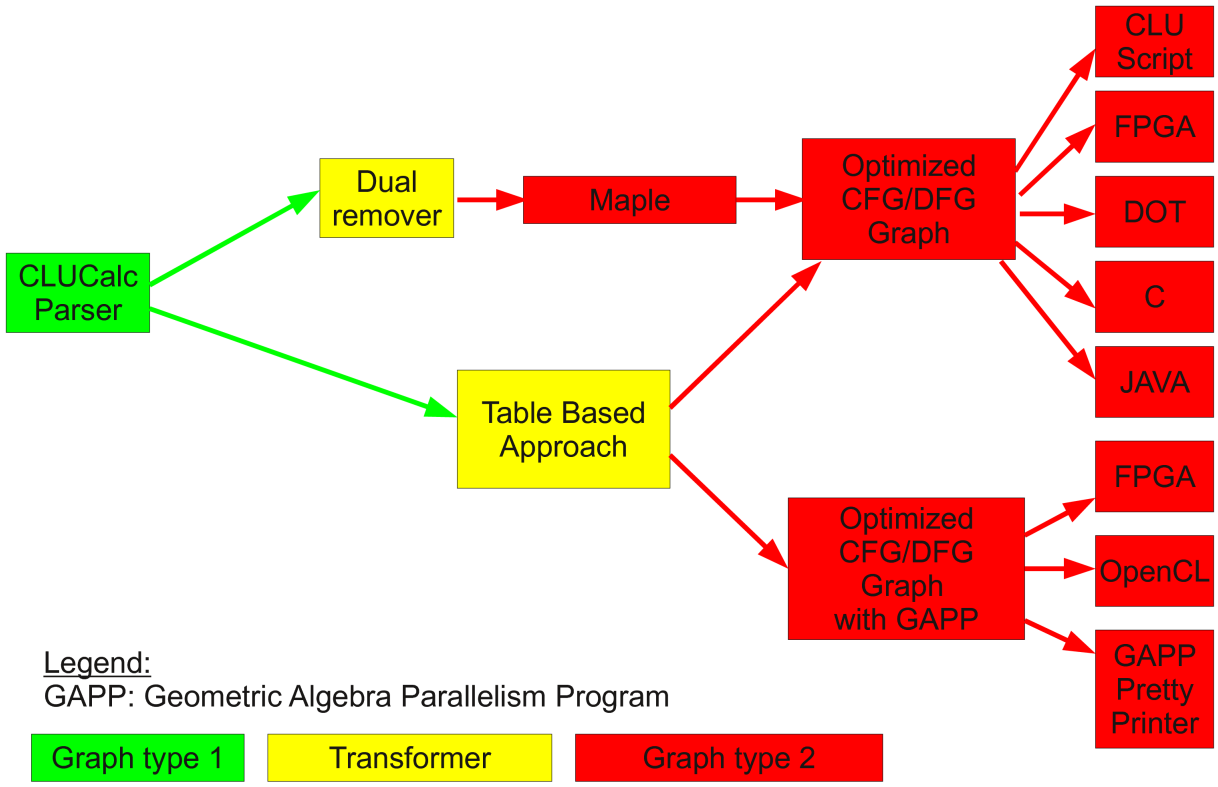


Figure 13.1.: Visualisation of the two graphs in the data flow of Gaalop

Intelligent "optInserting"

If the "optInserting"-option in the TBA Plugin is enabled, Maxima remembers all destination variables, except from those with a question mark. However, this can lead to large expressions. In some cases, e.g. a subexpression is used twice, this insertion can decrease performance of the generated program. So the future work is to implement a heuristic for deciding, which variable of an assignment should be remembered by Maxima.

Support of arbitrary algebras: Dualization Operation

In the current implementation, arbitrary algebras cannot be chosen in the GUI, because the CLUCalc Parser does not support arbitrary base vectors. Also the dualization operation is transformed to an inner product with the pseudoscalar. However, the pseudoscalar and the sign of the pseudoscalar depends on the used algebra. So, the design must be modified. It is also possible using two types of ControlFlowGraph and Expression graph. Type 1 contains the recent *ControlFlowGraph* and *Expression* graph with an extra *Expression*, which represents the dual operation. Type 2 is equal to the current data structure. See figure 13.1 for a visualisation of the data flow graph of Gaalop with two graphs. Note that in both cases the design of Gaalop must be changed.

Supporting prohibited variable names

If some variables are named as "n" and "normal", the CLUCalc Parser reports an error, because the Maple Optimization Strategy do not support variables with the names "n", "normal". However, in TBA using these names is permitted. So the future work is to make the CLUCalc Parser support "n" and "normal" as variable names, while paying attention to the requirement, that Maple still runs in a stable way.

A Exemplary calculation

Manually calculation of the centre of a circle through three given points in $\mathbb{R}^2$.

For simplicity, we choose these three points:

$$\vec{p_1} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \vec{p_2} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \vec{p_3} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

The representation of the three points in Conformal Geometric Algebra is:

$$\begin{aligned} p_1 &= 2e_1 + 1e_2 + 2.5e_\infty + e_0 \\ p_2 &= 1e_1 + 3e_2 + 5e_\infty + e_0 \\ p_3 &= 2e_1 + 4e_2 + 10e_\infty + e_0. \end{aligned}$$

Now, we compute the representation of the circle through the three points p_1, p_2, p_3.

$$\begin{aligned} c &= (p_1 \wedge p_2 \wedge p_3)^* \\ &= ((2e_1 + 1e_2 + 2.5e_\infty + e_0) \wedge (1e_1 + 3e_2 + 5e_\infty + e_0) \wedge p_3)^* \\ &= ((6e_1 \wedge e_2 + 10e_1 \wedge e_\infty + 2e_1 \wedge e_0 + e_2 \wedge e_1 \\ &\quad + 5e_2 \wedge e_\infty + e_2 \wedge e_0 + 2.5e_\infty \wedge e_1 + 7.5e_\infty \wedge e_2 + 2.5e_\infty \wedge e_0 \\ &\quad + e_0 \wedge e_1 + 3e_0 \wedge e_2 + 5e_\infty \wedge e_0) \wedge p_3)^* \\ &= ((5e_1 \wedge e_2 + 7.5e_1 \wedge e_\infty + e_1 \wedge e_0 + 2.5e_2 \wedge e_\infty \\ &\quad - 2e_2 \wedge e_0 - 2.5e_\infty \wedge e_0) \wedge (2e_1 + 4e_2 + 10e_\infty + e_0))^* \\ &= (50e_1 \wedge e_2 \wedge e_\infty + 5e_1 \wedge e_2 \wedge e_0 + 30e_1 \wedge e_\infty \wedge e_2 + \\ &\quad 7.5e_1 \wedge e_\infty \wedge e_0 + 4e_1 \wedge e_0 \wedge e_2 + 10e_1 \wedge e_0 \wedge e_\infty \\ &\quad - 5e_2 \wedge e_\infty \wedge e_1 - 2.5e_2 \wedge e_\infty \wedge e_0 - 4e_2 \wedge e_0 \wedge e_1 \\ &\quad - 20e_2 \wedge e_0 \wedge e_\infty - 5e_\infty \wedge e_0 \wedge e_1 - 10e_\infty \wedge e_0 \wedge e_2)^* \\ &= (15e_1 \wedge e_2 \wedge e_\infty - 3e_1 \wedge e_2 \wedge e_0 - 7.5e_1 \wedge e_\infty \wedge e_0 + 7.5e_2 \wedge e_\infty \wedge e_0)^* \\ &= (15e_1 \wedge e_2 \wedge e_\infty - 3e_1 \wedge e_2 \wedge e_0 - 7.5e_1 \wedge e_\infty \wedge e_0 + 7.5e_2 \wedge e_\infty \wedge e_0) \cdot (e_1 \wedge e_2 \wedge e_3 \wedge e_0 \wedge e_\infty) \\ &= (-15e_3 \wedge e_\infty - 3e_3 \wedge e_0 + 7.5e_2 \wedge e_3 + 7.5e_1 \wedge e_3) \end{aligned}$$

This is the representation of a circle through the three points. Now, we compute the centre of this circle.

$$\begin{aligned} m = ce_\infty c &= c(-3e_\infty \wedge e_3 \wedge e_0 - 3e_3 + 7.5e_\infty \wedge e_2 \wedge e_3 + 7.5e_\infty \wedge e_1 \wedge e_3) \\ &= (-15e_3 \wedge e_\infty - 3e_3 \wedge e_0 + 7.5e_2 \wedge e_3 + 7.5e_1 \wedge e_3)(-3e_\infty \wedge e_3 \wedge e_0 - 3e_3 + 7.5e_\infty \wedge e_2 \wedge e_3 + 7.5e_\infty \wedge e_1 \wedge e_3) \\ &= 9e_3e_0e_\infty e_3e_0 - 22.5e_3e_0e_\infty e_2e_3 - 22.5e_3e_0e_\infty e_1e_3 - 22.5e_2e_3e_\infty e_3e_0 + 56.25e_2e_3e_\infty e_2e_3 + \\ &\quad 56.25e_2e_3e_\infty e_1e_3 - 22.5e_1e_3e_\infty e_3e_0 + 56.25e_1e_3e_\infty e_2e_3 + 56.25e_1e_3e_\infty e_1e_3 \\ &= 9e_0e_\infty e_0 + 22.5e_0e_\infty e_2 + 22.5e_0e_\infty e_1 + 22.5e_2e_\infty e_0 + 56.25e_2e_\infty e_2 + \\ &\quad 56.25e_2e_\infty e_1 + 22.5e_1e_\infty e_0 + 56.25e_1e_\infty e_2 + 56.25e_3e_\infty e_3 \\ &= 9(e_0 \wedge e_\infty - 1)e_0 + 22.5(e_0 \wedge e_\infty - 1)e_2 + 22.5(e_0 \wedge e_\infty - 1)e_1 + \\ &\quad 22.5e_2(e_\infty \wedge e_0 - 1) - 56.25e_\infty + 22.5e_1(e_\infty \wedge e_0 - 1) - 56.25e_\infty \\ &= 9(-e_0 - e_0) + 22.5(e_0 \wedge e_\infty \wedge e_2 - e_2) + 22.5(e_0 \wedge e_\infty \wedge e_1 - e_1) + 22.5(e_2 \wedge e_\infty \wedge e_0 - e_2) \\ &\quad - 56.25e_\infty + 22.5(e_1 \wedge e_\infty \wedge e_0 - e_1) - 56.25e_\infty \\ &= -18e_0 - 22.5e_2 - 22.5e_1 - 22.5e_2 - 22.5e_1 - 112.5e_\infty \\ &= -18e_0 - 45e_2 - 45e_1 - 112.5e_\infty \\ &\cong 2.5e_1 + 2.5e_2 + 6.25e_\infty + e_0 \end{aligned}$$

This is the representation of the point $\vec{p_M} = \begin{pmatrix} 2.5 \\ 2.5 \end{pmatrix}$.

B Exemplary pass of a CLUScript with Table-Based Approach

For an example of the TBA transformation process, we take a modified example in 3d Geometric Algebra from [15]:

```
1 ?a=a1*e1+a2*e2+a3*e3;
2 ?b=b1*e1+b2*e2+b3*e3;
3 ?c=a*b;
4 ?d=a+d;
5 ?f=a^d;
```

Note, that we performed some low-level mathematical optimizations tacitly for clarity reasons.

$$a = \begin{pmatrix} 0 \\ a_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ a_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ a_3 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ a_1 \\ a_2 \\ a_3 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$b = \begin{pmatrix} 0 \\ b_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ b_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ b_3 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ b_1 \\ b_2 \\ b_3 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$c = \begin{pmatrix} 0 \\ a[1] \\ a[2] \\ a[3] \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} * \begin{pmatrix} 0 \\ b[1] \\ b[2] \\ b[3] \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} a[1]*b[1]+a[2]*b[2]+a[3]*b[3] \\ 0 \\ 0 \\ 0 \\ a[1]*b[2]-a[2]*b[1] \\ a[1]*b[3]-a[3]*b[1] \\ a[2]*b[3]-a[3]*b[2] \\ 0 \end{pmatrix}$$

$$d = \begin{pmatrix} 0 \\ a[1] \\ a[2] \\ a[3] \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} c[0] \\ 0 \\ 0 \\ 0 \\ c[4] \\ c[5] \\ c[6] \\ 0 \end{pmatrix} = \begin{pmatrix} c[0] \\ a[1] \\ a[2] \\ a[3] \\ c[4] \\ c[5] \\ c[6] \\ 0 \end{pmatrix}$$

$$f = \begin{pmatrix} 0 \\ a[1] \\ a[2] \\ a[3] \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} d[0] \\ d[1] \\ d[2] \\ d[3] \\ d[4] \\ d[5] \\ d[6] \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ a[1]*d[0] \\ a[2]*d[0] \\ a[3]*d[0] \\ a[1]*d[2]-a[2]*d[1] \\ a[1]*d[3]-a[3]*d[1] \\ a[2]*d[3]-a[3]*d[2] \\ a[1]*d[6]-a[2]*d[5]+a[3]*d[4] \end{pmatrix}$$

This yields the optimized algorithm, shown in listing B.1.

Listing B.1: Optimized algorithm of modified example from [15]

```
a[1] = a1; // e1
a[2] = a2; // e2
a[3] = a3; // e3
b[1] = b1; // e1
b[2] = b2; // e2
b[3] = b3; // e3
c[0] = a[3] * b[3] + a[2] * b[2] + a[1] * b[1]; // 1.0
c[4] = a[1] * b[2] - a[2] * b[1]; // e1^e2
c[5] = a[1] * b[3] - a[3] * b[1]; // e1^e3
c[6] = a[2] * b[3] - a[3] * b[2]; // e2^e3
d[0] = c[0]; // 1.0
d[1] = a[1]; // e1
d[2] = a[2]; // e2
d[3] = a[3]; // e3
d[4] = c[4]; // e1^e2
d[5] = c[5]; // e1^e3
d[6] = c[6]; // e2^e3
f[1] = a[1] * d[0]; // e1
f[2] = a[2] * d[0]; // e2
f[3] = a[3] * d[0]; // e3
f[4] = a[1] * d[2] - a[2] * d[1]; // e1^e2
f[5] = a[1] * d[3] - a[3] * d[1]; // e1^e3
f[6] = a[2] * d[3] - a[3] * d[2]; // e2^e3
f[7] = a[1] * d[6] - a[2] * d[5] + a[3] * d[4]; // e1^e2^e3
```

C Multiplication table listings

C.1 Blade file listing for Conformal Geometric Algebra

Listing C.1: Listing of the blade file for Conformal Geometric Algebra

```
1;e1;e2;e3;einf;e0
1
e1
e2
e3
einf
e0
e1^e2
e1^e3
e1^einf
e1^e0
e2^e3
e2^einf
e2^e0
e3^einf
e3^e0
einf^e0
e1^e2^e3
e1^e2^einf
e1^e2^e0
e1^e3^einf
e1^e3^e0
e1^einf^e0
e2^e3^einf
e2^e3^e0
e2^einf^e0
e3^einf^e0
e1^e2^e3^einf
e1^e2^e3^e0
e1^e2^einf^e0
e1^e3^einf^e0
e2^e3^einf^e0
e1^e2^e3^einf^e0
```

C.2 Extract of the products file listing for Conformal Geometric Algebra

Listing C.2: Listing of an extract of the products file for Conformal Geometric Algebra

```
1;1;0;1;1
1;E1;0;E1;E1
1;E2;0;E2;E2
1;E3;0;E3;E3
1;E4;0;E4;E4
1;E5;0;E5;E5
1;E6;0;E6;E6
1;E7;0;E7;E7
1;E8;0;E8;E8
1;E9;0;E9;E9
1;E10;0;E10;E10
1;E11;0;E11;E11
1;E12;0;E12;E12
1;E13;0;E13;E13
1;E14;0;E14;E14
1;E15;0;E15;E15
1;E16;0;E16;E16
1;E17;0;E17;E17
1;E18;0;E18;E18
1;E19;0;E19;E19
1;E20;0;E20;E20
1;E21;0;E21;E21
1;E22;0;E22;E22
1;E23;0;E23;E23
1;E24;0;E24;E24
1;E25;0;E25;E25
1;E26;0;E26;E26
1;E27;0;E27;E27
```

```
1;E28;0;E28;E28
1;E29;0;E29;E29
1;E30;0;E30;E30
1;E31;0;E31;E31
E1;1;0;E1;E1
E1;E1;1;0;1
E1;E2;0;E6;E6
E1;E3;0;E7;E7
E1;E4;0;E8;E8
E1;E5;0;E9;E9
E1;E6;E2;0;E2
E1;E7;E3;0;E3
E1;E8;E4;0;E4
E1;E9;E5;0;E5
E1;E10;0;E16;E16
E1;E11;0;E17;E17
E1;E12;0;E18;E18
E1;E13;0;E19;E19
E1;E14;0;E20;E20
E1;E15;0;E21;E21
E1;E16;E10;0;E10
E1;E17;E11;0;E11
E1;E18;E12;0;E12
E1;E19;E13;0;E13
E1;E20;E14;0;E14
E1;E21;E15;0;E15
E1;E22;0;E26;E26
E1;E23;0;E27;E27
E1;E24;0;E28;E28
E1;E25;0;E29;E29
E1;E26;E22;0;E22
E1;E27;E23;0;E23
E1;E28;E24;0;E24
E1;E29;E25;0;E25
E1;E30;0;E31;E31
E1;E31;E30;0;E30
```

List of Figures

List of Tables

Bibliography

[1] Antlr. http://www.antlr.org.

[2] Clifford lib for maple. http://math.tntech.edu/rafal/.

[3] Clucalc / cluviz. http://www.clucalc.info.

[4] Dot. http://www.graphviz.org.

[5] Gaalop. http://www.gaalop.de.

[6] Gaigen 2. http://staff.science.uva.nl/ fontijne/gaigen2.html.

[7] Maple. http://www.maplesoft.com/products/Maple/index.aspx.

[8] Maxima. http://maxima.sourceforge.net/.

[9] Opencl. http://www.khronos.org/opencl/.

[10] Patrick Charrier. Gaalop compiler driver. Bachelor Thesis, TU Darmstadt, 2011.

[11] Ahmad Hosney Awad Eid. *Optimized Automatic Code Generation for Geometric Algebra Based Algorithms with Ray Tracing Application*. PhD thesis, Port-Said, April 2010.

[12] Daniel Fontijne, Leo Dorst, and Stephen Mann. *Efficient Implementation of Geometric Algebra*. PhD thesis, University of Amsterdam, 2007. Available at `http://staff.science.uva.nl/~fontijne/phd.html`.

[13] Erich Gamma et al. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[14] Dietmar Hildenbrand. *Geometric Computing in Computer Graphics and Robotics using Conformal Geometric Algebra*. PhD thesis, TU Darmstadt, 2006.

[15] Dietmar Hildenbrand, Patrick Charrier, and Andreas Koch. Specialized machine instruction set for geometric algebra. submitted to AGACSE 2012.

[16] Christian Perwass. *Geometric Algebra with Applications in Engineering*, chapter 3.8.2. Springer, 2009.

[17] Christian Schwinn, Andreas Görlitz, and Dietmar Hildenbrand. Geometric algebra computing on the cuda platform. GraVisMa 2009, 2009.

[18] Florian Seybold. Gaalet - a c++ expression template library for implementing geometric algebra. VizWorkshop, 2010.

[19] Florian Seybold, Patrick Charrier, Dietmar Hildenbrand, M. Bernreuther, and D. Jenz. Runtime performance of a molecular dynamics model using conformal geometric algebra. Slides available at `http://www.science.uva.nl/~leo/agacse2010/talks_world/Seybold.pdf`, 2010.

[20] Julio Zamora-Esquivel. G 6,3 geometric algebra. 9th International Conference on Clifford Algebras and their Applications in Mathematical Physics, Weimar, 2011, July.