On the Refinement of Algebraic Concepts

May 18, 2005

1 Overview

As the document became more lengthy than expected in the beginning, we give an overview of the ideas with references to examples or detailed description.

1. Algebraic concepts are only approximately modeled and only in rare cases exactly, section 3.
2. Pure algebraic concepts require multi-type concepts, a primary type and an operation type in form of a functor, section 4
3. Formal specification of these concepts, section 4.1–4.7.
4. With pure algebraic concepts the same function can be used w.r.t. addition and multiplication, e.g., section 5.5.
5. Even more, any type can have an arbitrary number of operations, e.g., section 4.9 and 4.10.
6. Operations can be arbitrary as long as they fulfill the properties, e.g. string concatenation in section 5.4.
7. Some primary type may need an `operator+` and a functor. We try to avoid double definition, section 5.
8. Types with appropriate operators can be used directly in pure algebraic functions with default functors, section 5.5.
9. Associativity and commutativity are handled with type traits, section 6. Algorithm implementations may check whether a certain needed property is given for a certain operation and react in different ways, for instance
   - Throw an exception,
   - Disable or enable the implementation, or
   - Dispatch between different implementations.
10. Additive concepts are refinements of pure algebraic concepts.

11. An informal definition of this refinement is not sufficient. An implementable definition is required, i.e. if type `T` models for instance AdditiveMonoid then it also models Monoid and can be used as parameter of every function requiring Monoid.
12. Additive concepts only need one type, pure algebraic need two. How can it be a refinement?
13. Defining the addition of additive concepts as functor types requires that the functors must always be passed as an extra parameter to each function using an additive concept. This handling would be too burdensome and is not acceptable therefore.
14. Refinement from multi-type to single-type concepts can be realized with the default functors, section 7.
15. Concepts with two operations are straight-forward refinements of additive and multiplicative concepts, section 8.
16. A complex concept hierarchy does not imply complicated programs as shown in different examples.

2 Introduction

Using generic programming concepts to define algebraic structures creates a symbiosis between generic programming and algebra. The idea to abstract from irrelevant behavior of algebraic structures and to focus on the essential properties was the archetype for concepts in generic programming, i.e. writing algorithms that rely only on essential properties of data types. Applying this technique to characterize different algebraic structures is therefore only natural.

The definition of concepts for linear algebra raises questions like

- Are additive groups refinements of groups?
- How to define a group without addition or multiplication?
- How elements of this groups can be used if there are no + or * operators?

One way to address these topics is to define pure groups, pure semi-groups etc only as theoretical background of additive and multiplicative pure groups, pure semi-groups. In this case all implementations can be realized with + and * and the concepts of pure algebraic structures does not impact the programming.

We prefer to not limit a generic library to only two operations. There can be much more than two binary operations associated with one type and an operator based implementation would be far too restrictive. Any operation that fulfills for instance the requirements of commutative monoid should be usable for any algorithm that requires a commutative monoid. On the other hand, two computations are sometimes based on the same implementation except that

one function uses always addition where the other one uses only multiplication. These functions can be implemented with the same template function, where the operation is a template parameter. The meaning of the computations changes of course for different operators. For instance the same function could calculate a (very slow) division with the addition and a logarithm with the multiplication.

3 Are there Models?

To be precise, floating point numbers are not associative due to rounding errors. How strong these errors affect the results depends on the numerical stability of the calculation. In particular, the division by differences of almost equal values can amplify errors dramatically $1/(a-b)$, where $a \approx b$. Note that the amplification not only affects rounding errors of operations. Due to the finite precision, the input values are already rounded to the next floating point number and this difference can grow to an arbitrary size, too.

There are other subtle issues concerning bit-equality of floating point numbers. Some processors have a higher internal precision, which is kept as long as values are stored in registers. Storing this values forces rounding to the normal precision. Therefor, two floating point values computed with exactly the same operations in exactly the same order are not guaranteed to be bit-identical if the storing behavior was different. Of course, these tiny differences can be arbitrarily amplified with numerically instable computations.

These examples shall underline the importance of numerical stability in floating point calculation, and wherever possible instable algorithms should be replaced by more stable ones, e.g. instead of using standard Gaußian elimination applying pivoting or even iterative linear solvers. Unfortunately, the numerical instability often does not belong to the algorithm but to problem itself, like in weather simulation. In this case, errors grow inevitably during the computation.[1]

Rounding errors are usually handled in comparisons by allowing a certain tolerance ε, which can be absolute $a =_\varepsilon b \stackrel{\text{def}}{:=} |a-b| < \varepsilon$ or relative $a =_\varepsilon b \stackrel{\text{def}}{:=} |a-b|/\max(a,b) < \varepsilon$. The drawback of this technique is that the modified equality is not a equality relation because the transitivity is lost

$$a =_\varepsilon b \vee b =_\varepsilon c \quad \not\to \quad a =_\varepsilon c.$$

Many algebraic structures require completeness in their definition, i.e. each Cauchy sequence is converting against an element of the set. Floating point numbers are not even dense in real numbers like rational numbers so that is already impossible to define an arbitrary Cauchy sequence in floating point numbers.

Another imperfection of floating point numbers is that they are not closed under addition, i.e. for two extreme large numbers the sum is not a valid floating point number. Therefore, the requirements of the simplest algebraic structure,

[1] This is the reason why weather can only be forecasted for some days.

a magma (or groupoid), are not fulfilled for addition nor for multiplication. It is possible, however questionable, to regard $\pm\infty$ in ANSI/IEEE standard 754-1985 numbers as valid floating point values. Thus the operations would be closed but only in rare cases programs return correct results once $\pm\infty$ values are involved.

Although floating point numbers have usually nearly symmetric exponent ranges – e.g. from -64 to +63 – some very small value or some very large values w.r.t. the magnitude may not have valid reciprocal values. Concerning the multiplicative inversion, $\pm\infty$ cannot be regarded as a valid result because $1/a = \infty$ does not imply $a \star \infty = 1$!

The closure of operations is also not given for integers, both signed and unsigned. An interesting property of all integer formats is that all operations are associative even if overflows or underflows occur and the result might be completely wrong.

More precisely, computer `int` numbers behaves only in the wrong way if they are considered as representation of (infinite) integers. If 32 bit `int`s are regarded as cyclic ring from -2,147,483,648 to 2,147,483,647 then both addition and multiplication are closed. Unsigned 32 bit `int`s build a cyclic ring from 0 to 4,294,967,295 also providing closure under both operations.[2] In other words, whether an overflow or underflow of `int` causes a wrong result depends on the point of view. However, in most cases it is regarded as an error.

Using extended data formats does not solve this problem, it only delays it. Even dynamical extension of number formats does not allow arbitrary large values since the memory is limited. Thus, no infinite set can be correctly represent in a computer. Rather than refusing all imperfect opportunities to represent real numbers, floating point numbers can be considered as models of it, whereby the term model is not used as in generic programming but as in physics. A physical model of an object does not necessarily have all properties but the essential properties sufficiently approximated in order to describe the object's behavior with satisfactory exactness, eventually in a restricted environment. For value ranges far enough from the limits (maximum, minimum and 0) and numerically stable calculations, the calculated results are usually close enough to the correct values.

To what extend these criteria are true must be evaluated separately for each algorithm and each range of possible input data. If the resulting exactness allows a satisfying approximation of the algebraic structures' behavior is finally left to the user's decision. In which form a linear algebra library addresses these problems is subject of ongoing discussions and omitted in this paper.

To give a positive counter-example to these problems with infinite sets, we show in section 5.6 that the computational representation of finite sets can model the requirements of algebraic structures exactly.

[2] Signed and unsigned `int`s of the same size are isomorphic w.r.t. addition and multiplication. For 32 bit -2,147,483,648 corresponds to 2,147,483,648, -2,147,483,647 to 2,147,483,649, ... and finally -1 to 4,294,967,295.

4 Pure Algebraic Concepts

In distinction to algebraic structures with an addition or multiplication as operation, we call algebraic structures with an arbitrary binary operation pure algebraic structure. All characteristics of this operation are either explicitly defined or deducible from the explicit definitions. Concepts defining the requirements of pure algebraic structures are called pure algebraic concepts.

All concepts in this section characterize modules of two types: a functor that represents an operation and a type the functor operates on.

4.1 Magma

A *Magma* – also called groupoid[3] – is a set of elements (T) and an operation (op) over the set. The set must be closed under this operation

$$a, b \in T \quad \rightarrow \quad \mathrm{op}(a, b) \in T.$$

Refinement of

Assignable

Notation

{`T, Op`}	are types that build a model of Magma.
`a, b`	are objects of type `T`.
`op`	is an object of type `Op`.

Valid Expressions

- Operation
  `op(a, b)`
  Return Type: `T` or a type convertible to `T`.

4.2 SemiGroup

A *Semi-Group* is a magma where the operation is associative.

Refinement of

Magma

Notation

{`T, Op`}	are types that build a model of SemiGroup.
`a, b, c`	are objects of type `T`.
`op`	is an object of type `Op`.

[3] We do not use this term here because of its ambiguity in algebra.

Associated Types

- Type trait for associativity checking
  `glas::is_op_associative`

Invariants

- Associativity
  `op(op(a, b), c)` = `op(a, op(b, c))`
- Associativity check
  `glas::is_op_associative<Op>::value` = `true`, confer section 6.

4.3 CommutativeSemiGroup

A *Commutative Semi-Group* is a commutative semi-group, obviously.

Refinement of

SemiGroup

Notation

{`T, Op`}	are types that build a model of CommutativeSemiGroup.
`a, b`	are objects of type `T`.
`op`	is an object of type `Op`.

Associated Types

- Type trait for commutativity checking
  `glas::is_op_commutative`

Invariants

- Commutativity
  `op(a, b)` = `op(b, a)`
- Commutativity check
  `glas::is_op_commutative<Op>::value` = `true`, confer section 6.

Models

- Non-negative double as `age` with functor type `pythagoras_t` computing Euclidean distance, see section 4.10.

4.4 Monoid

A *Monoid* is a semi-group with an identity.

Refinement of

SemiGroup

Notation

{`T, Op`}	are types that build a model of Monoid.
`a`	is an object of type `T`.
`op`	is an object of type `Op`.

Valid Expressions

- Identity
  `op.identity()`
  Return Type: `T` or a type convertible to `T`.

Invariants

- Commutativity from left
  `op(op.identity(), a)` = `a`
- Commutativity from right
  `op(a, op.identity())` = `a`

Models

- STL strings with concatenation-based functor and empty string as identity, see section 5.4

4.5 CommutativeMonoid

A *Commutative Monoid* is of course a commutative monoid or alternatively a commutative semi-group with an identity.

Refinement of

Monoid and CommutativeSemiGroup

Models

- Non-negative double as `age` with an addition-like functor type `ageAdd_t` like in section 4.9.
- Same type with functor type `pMonoid_t` computing Euclidean distance, see section 4.10.

4.6 Group

A *Group* is a monoid with an inverse function.

Refinement of

Monoid

Notation

`{T, Op}`	are types that build a model of Group.
`a`	is an object of type `T`.
`op`	is an object of type `Op`.

Valid Expressions

- Inverse element
  `op.inverse(a)`
  Return Type: `T` or a type convertible to `T`.

Invariants

- Cancellation from left
  `op(op.inverse(a), a)` = `op.identity()`

- Cancellation from right
  `op(a, op.inverse(a))` = `op.identity()`

4.7 AbelianGroup

An *Abelian Group* is a commutative group or alternatively a commutative monoid with an inverse function.

Refinement of

Group and CommutativeMonoid

Models

- `modN_t<n>` with functor implementing + or *, see section 5.6.

- Contingent: `int` with functor implementing +, see section 3.

- Contingent: `float` with functor implementing + or *, see section 3.

- Contingent: `complex<double>` with functor implementing + or *, same as `float`.

4.8 Examples for Pure Algebraic Functions

The following functions only rely on functors and do not require the existence of operators.

```
// {T, Op} must be a Magma
// T must be EqualityComparable
template <class T, class Op>
inline bool equalResults(const T& v1a, const T& v1b,
                         const T& v2a, const T& v2b, Op op) {
  return op(v1a, v1b) == op(v2a, v2b);
}

// {T, Op} must be a Monoid
// T must be EqualityComparable
template <class T, class Op>
inline bool identityPair(const T& v1, const T& v2, Op op) {
  return op(v1, v2) == op.identity();
}
```

To calculate something useful, EqualityComparable is additional required. As one can see from the code, the first function compares two results and the second one compares the result of an operation with the identity.

4.9 Age Example with Functor

In the following example we introduce a new type for non-negative `doubles`, which could be for instance represent an age of a person. The values cannot be changed and the constructor verifies that the positiveness.

```
// age_example1f.cpp
#include <iostream.h>
#include "algebraic_functions.hpp"

// User defined data types and operators
class age {
  double myAge;
 public:
  age(double m): myAge(m) {
    if (m < 0.0) throw "Negative Age"; }
  double sayAge() const {
    return myAge; }
};

inline bool operator==(const age& x, const age& y) {
  return x.sayAge() == y.sayAge(); }

inline ostream& operator<< (ostream& stream, const age& a) {
  return stream << a.sayAge(); }
```

```
// User defined functor which build a commutative monoid with age
struct ageAdd_t {
  age operator() (const age& x, const age& y) {
    return age(x.sayAge() + y.sayAge()); }
  age identity() {
    return age(0); }
} ageAdd;

int main(int, char* []) {
  age a0(0.0), a2(2.0), a3(3.0), a4(4.0), a5(5.0);

  cout << "equalResults(a2,a5,  a3,a4, ageAdd) "
       << equalResults(a2,a5,  a3,a4, ageAdd)  << endl;
  cout << "equalResults(a2,a4,  a3,a4, ageAdd) "
       << equalResults(a2,a4,  a3,a4, ageAdd)  << endl;
  cout << "identityPair(a2,a4, ageAdd) "
       << identityPair(a2,a4, ageAdd)  << endl;
  cout << "identityPair(a0,a0, ageAdd) "
       << identityPair(a0,a0, ageAdd)  << endl;

  return 0;
}
```

The functor defined in the example fulfills the requirements that the module {`age`, `ageAdd_t`} is a CommutativeMonoid. So, the algebraic functions can be used. The results are as expected 1, 0, 0, and 1.

4.10 Age Example for an Alternative Functor

Alternatively to the standard addition, we introduce an operation that corresponds to the Euclidean distance $\sqrt{x^2 + y^2}$, which is associative and commutative but has no inverse in real numbers.

```
// age_example2f.cpp
#include <iostream.h>
#include <cmath>

#include "algebraic_functions.hpp"
 :
 :
struct pythagoras_t {
  age operator() (const age& x, const age& y) {
    return age(sqrt(x.sayAge()*x.sayAge() + y.sayAge()*y.sayAge())); }
} pythagoras;

struct pMonoid_t: public pythagoras_t {
  age identity() {
    return age(0); }
} pMonoid;
```

```
int main(int, char* []) {
  age a0(0.0), a2(2.0), a3(3.0), a4(4.0), a5(5.0);

  cout << "equalResults(a0,a5,  a3,a4, pythagoras) "
       << equalResults(a0,a5,  a3,a4, pythagoras)  << endl;
  cout << "equalResults(a2,a4,  a3,a3, pMonoid) "
       << equalResults(a2,a4,  a3,a3, pMonoid)  << endl;
  cout << "identityPair(a2,a4, pMonoid) "
       << identityPair(a2,a4, pMonoid)  << endl;
  cout << "identityPair(a0,a0, pMonoid) "
       << identityPair(a0,a0, pMonoid)  << endl;

  return 0;
}
```

We dropped the definition of the `age` class and the operators as they are the same as in section 4.9. The functor type `pythagoras_t` completes `age` to a Magma, whereas the types {`age, pMonoid_t`} fulfills the requirements of Commutative-Monoid. The execution returns 1, 0, 0, and 1.

Of course, objects of `pMonoid_t` can be used in `equalResults` but objects of `pythagoras_t` not in `identityPair` because it requires a Monoid. In the later case, the compiler gives an error message that 'identity' is not defined.

5 Define One – Get One Free

The usage of external functors allows an arbitrary number of operations. On the other hand, many programs require the existence of numeric operators. Thus, it is our goal to define only one type and derive the other one.

We decided to derive the functors from the operators because the other way has several disadvantages. Firstly, operators are globally accessible and cannot be put into a separate namespace so that it is preferable to not build them automatically. Secondly, the derivation must be defined several times for the involved operators whereas the derivation of a functor can be done in one type definition as we will show in the following. Thirdly, it is impossible in general to define an efficient implementation for `+=` based on a binary operation the returns the result on the stack.[4]

5.1 Default Functors derived from Operators

The default functors were put into a separate namespace for better control.

```
// default_functors_wo_markers.hpp, markers added later on
#ifndef glas_default_functors_include
#define glas_default_functors_include

namespace glas { namespace def {
```

[4] **Question:** More reasons? Counter-arguments?

```
template <class T>
class magmaAddOp {
public:
  T operator() (const T& x, const T& y) {
    return x + y; }
};

template <class T, class Base= magmaAddOp<T> >
class semiGroupAddOp: public Base {};

template <class T, class Base= semiGroupAddOp<T> >
class commSemiGroupAddOp: public Base {};

template <class T, class Base= semiGroupAddOp<T> >
class monoidAddOp: public Base {
public:
  T identity() {
    return T(0); }
};

template <class T, class Base= monoidAddOp<T> >
class commMonoidAddOp: public Base {};

template <class T, class Base= monoidAddOp<T> >
class groupAddOp: public Base {
public:
  T inverse(const T& x) {
    return identity() - x; }
};

template <class T, class Base= groupAddOp<T> >
class abeleanGroupAddOp: public Base {};

template <class T>
class magmaMultOp {
public:
  T operator() (const T& x, const T& y) {
    return x * y; }
};

template <class T, class Base= magmaMultOp<T> >
class semiGroupMultOp: public Base {};

template <class T, class Base= semiGroupMultOp<T> >
  class commSemiGroupMultOp: public Base {};

template <class T, class Base= semiGroupMultOp<T> >
class monoidMultOp: public Base {
public:
```

```
    T identity() {
      return T(1); }
  };

  template <class T, class Base= monoidMultOp<T> >
  class commMonoidMultOp: public Base {};

  template <class T, class Base= monoidMultOp<T> >
  class groupMultOp: public Base {
  public:
    T inverse(const T& x) {
      return identity() / x; }
  };

  template <class T, class Base= groupMultOp<T> >
  class abeleanGroupMultOp: public Base {};

} // namespace def
} // namespace glas

#endif // glas_default_functors_include
```

There are two types of functors: for addition and for multiplication. The naming conventions and the derivation structures are the same.

Using a functor is an explicit declaration of the user that the considered type models the corresponding concept for the respective operation. For instance, the utilization of `glas::def::commMonoidAddOp` for type `T` means that the user declares {`T, glas::def::commMonoidAddOp<T>`} as a model of CommutativeMonoid with respect to the addition.

The definition of the base class as template parameter allows more flexibility, which we will show in examples. Please notice that the derivation of classes is only a tool for easier implementation and that we do not rely on the class hierarchy in algorithms, which would restrict generality. The difference between commutative and non-commutative functors will be significant at a later stage.

For most numerical data types like `int`, `float`, `double`, and `complex<double>`, where all operators and the identity elements are defined in a consistent form, the functors can be used directly without further definition, cf. section 5.5.

5.2 Age Example with Default Functors

The age example from section 4.9 can also be implemented like this

```
inline age operator+(const age& x, const age& y) {
  return age(x.sayAge() + y.sayAge()); }

int main(int, char* []) {
  age a0(0.0), a2(2.0), a3(3.0), a4(4.0), a5(5.0);
  glas::def::magmaAddOp<age>    ageMagmaAdd;
  glas::def::monoidAddOp<age>   ageMonoidAdd;
```

```
  cout << "equalResults(a2,a5,  a3,a4, ageMagmaAdd) "
       << equalResults(a2,a5,  a3,a4, ageMagmaAdd)  << endl;
  cout << "equalResults(a2,a4,  a3,a4, ageMagmaAdd) "
       << equalResults(a2,a4,  a3,a4, ageMagmaAdd)  << endl;
  cout << "identityPair(a2,a4, ageMonoidAdd) "
       << identityPair(a2,a4, ageMonoidAdd)  << endl;
  cout << "identityPair(a0,a0, ageMonoidAdd) "
       << identityPair(a0,a0, ageMonoidAdd)  << endl;
  return 0;
```

The definition of `age` was omitted as well as the equality and output operators. The new parts in the code are the + operator and the inclusion of `glas.hpp` to provide the default functors. The advantage is that both the functor and the binary operator + are available to implement algorithms. As 0 can be converted into the `age` type, we do not need to define the identity explicitly.

The implementation of the Euclidean distance in section 4.10 can also be simplified with default functors. Suppose the functor for the Magma concept is defined in the same way, we define the CommutativeMonoid functor by default

```
  glas::def::commMonoidAddOp<age, pythagoras_t>  pMonoid;
```

We could go one step further and implement the Euclidean distance as `operator+`. As this would be counter-intuitive, we prefer to implement only operations as operators whose behaviors corresponds to the expectations of the corresponding operator.

5.3 More Algebraic Functions

In this section we introduce three new functions for repeated operations and to emulate a division with minus respectively emulating logarithms with division.

```
// {T, Op} must be a Monoid
template <class T, class Op>
inline T multiplyAndSquare(T base, int exp, Op op) {
  T value(op.identity()), square(base);
  for (; exp > 0; exp>>= 1) {
    if (exp & 1) value= op(value, square);
    square= op(square, square); }
  return value;
}

// {T, Op} must be a Group
// T must be LessThanComparable and Assignable
template <class T, class Op>
inline int poorMensDivision(const T& v1, const T& v2, Op op) {
  // copies to avoid redundant operations
  T id(op.identity()), iv2(op.inverse(v2)), tmp(v1);

  if (v1 <= op.identity()) return 0;
```

```
  int counter= 0;
  for (; tmp > id; counter++) tmp= op(tmp, iv2);
  if (tmp < id) counter--;
  return counter;
}

// {T, Op} must be a Group
// T must be LessThanComparable and Assignable
template <class T, class Op>
inline int poorMensAbsDivision(const T& v1, const T& v2, Op op) {
  // copies to avoid redundant operations
  T id(op.identity()), iv2(v2 < op.identity() ? v2 : op.inverse(v2)),
    va1(v1 < op.identity() ? op.inverse(v1) : v1), tmp(va1);
  if (va1 <= op.identity()) return 0;
  int counter= 0;
  for (; tmp > id; counter++) tmp= op(tmp, iv2);
  if (tmp < id) counter--;
  return counter;
}
```

Multiply-and-Square is a fast method to compute $\mathrm{op}(a, \mathrm{op}(a, \ldots \mathrm{op}(a, a) \ldots))$ where a appears n[5] times. The straight-forward computation requires $\mathcal{O}(n)$ operations. Computing squares and squares of squares and so on reduces the computational effort to $\mathcal{O}(\log(n)$.

The function `poorMensDivision` computes a division in terms of repeated subtractions, in case the functor is based on addition. The result corresponds to logarithms for multiplication-based operators. `poorMensAbsDivision` calculates at first the magnitude of the values, which is the reciprocal value for multiplication-based operators in case it is smaller than the identity.

Both functions could be programmed in a simpler way comparing `tmp` with `v2`, which works correctly for most data types but can fail for cyclic sets like in section 5.6. In the same section, an example for a very expensive inverse function can be found, which was an additional reason to store inverse values instead of recomputing them. However, divisions are in general more expensive than multiplications – taking more processor cycles and often breaking pipelining – so that it is better to store inverse values, anyway. For addition-based functors the difference between saving and recomputing is usually negligible but as authors of generic functions we should avoid such assumptions.

5.4 Example with STL Strings

Another style of derivation is used in the following example. Here, the user-defined functor is derived from default functor because the + operator behaves as required but the identity cannot be converted from 0.

```
using std::string;
```

[5] **Question:** Which is the best type for it?

```
struct stringMonoid_t: glas::def::semiGroupAddOp<string> {
  string identity() {
    return string(); }
} stringMonoid;

int main(int, char* []) {
  string sa("a"), sab("ab"), sbc("bc"), sc("c"), s;
  glas::def::semiGroupAddOp<string>     stringSemiGroup;

  cout << "equalResults(sa, sbc,  sab, sc, stringMonoid) "
       << equalResults(sa, sbc,  sab, sc, stringMonoid)  << endl;
  cout << "equalResults(sab, sbc,  sab, sc, stringSemiGroup) "
       << equalResults(sab, sbc,  sab, sc, stringSemiGroup) << endl;
  cout << "identityPair(s, s, stringMonoid) "
       << identityPair(s, s, stringMonoid)  << endl;
  cout << "multiplyAndSquare(sab, 13, stringMonoid) "
       << multiplyAndSquare(sab, 13, stringMonoid) << endl;
  return 0;
}
```

Strings are Monoids with respect to the concatenation, using a functor like in the example program. The concatenation is associative and the empty string behaves neutrally. Of course, the concatenation is not commutative.

5.5 Multiplication of Floating Point Numbers as Functor

The algebraic functions can be applied directly to numerical data without further definitions.

```
int main(int, char* []) {
  using namespace glas::def;
  cout << "equalResults(2.,5.,  3.,4., magmaMultOp<float>()) "
       << equalResults(2.,5.,  3.,4., magmaMultOp<float>())  << endl;
  cout << "identityPair(0.5,2., monoidMultOp<float>()) "
       << identityPair(0.5,2., monoidMultOp<float>())  << endl << endl;
  cout << "poorMensDivision(33.,2., groupAddOp<float>()) "
       << poorMensDivision(33.,2., groupAddOp<float>())  << endl;
  cout << "poorMensDivision(33.,2., groupMultOp<float>()) "
       << poorMensDivision(33.,2., groupMultOp<float>())  << endl;
  cout << "poorMensAbsDivision(0.125,2, groupMultOp<float>()) "
       << poorMensAbsDivision(0.125,2., groupMultOp<float>())  << endl;
  return 0;
}
```

The emulated division calculated down-rounded quotients with an addition-based functors and down-rounded logarithms with multiplication-based functors. The ‘abs’ in the multiplicative context means the reciprocal value if it is smaller then one.[6] The results are therefore 0, 1, 5, 16, and 3.

[6] I know that this example is strange and that abs(-2) is -0.5.

5.6 Finite Sets

Cyclic sets are the only algebraic structures whose computer representation does not encounter the problems described in section 3 if the largest sum or product can be represented in the underlying `int` format.[7] These finite sets are Abelian groups with respect to the addition. Concerning the multiplication, the sets are commutative monoids and in the case that the cycle is a prime number, an Abelian group. Given the many references to this data type and being the only real model of the concepts, we print here the complete code of this example.

```
#include <iostream.h>

#include "glas.hpp"
#include "algebraic_functions.hpp"

template<unsigned n>
class modN_t {
  unsigned value;
 public:
  // const unsigned myN= n;
  modN_t(int v) {
    value= v >= 0 ? v%n : n - -v%n; } // no modulo of negative numbers
  modN_t(const modN_t<n>& m): value(m.get()) {}
  modN_t<n>& operator= (const modN_t<n>& m) {
    value= m.value; return *this; }
  template<unsigned OtherN>
  modN_t<n>& convert(const modN_t<OtherN>& m) {
    value= m.value >= 0 ? m.value%n : n - -m.value%n; return *this; }
  unsigned get() const {
    return value; }
};

template<unsigned n>
inline ostream& operator<< (ostream& stream, const modN_t<n>& a) {
  return stream << a.get(); }

template<unsigned n>
inline bool operator==(const modN_t<n>& x, const modN_t<n>& y) {
  return x.get() == y.get(); }

template<unsigned n>
inline bool operator>=(const modN_t<n>& x, const modN_t<n>& y) {
  return x.get() >= y.get(); }
// other comparison operators are defined respectively

template<unsigned n>
inline modN_t<n> operator+ (const modN_t<n>& x, const modN_t<n>& y) {
```

[7] **Question:** Are there tricks for n close to the maximum? Shall we consider this at some point? Certainly, not the most important.

```
  return modN_t<n>(x.get() + y.get()); }

template<unsigned n>
inline modN_t<n> operator- (const modN_t<n>& x, const modN_t<n>& y) {
  // add n to avoid negative numbers
  return modN_t<n>(n + x.get() - y.get()); }

template<unsigned n>
inline modN_t<n> operator* (const modN_t<n>& x, const modN_t<n>& y) {
  return modN_t<n>(x.get() * y.get()); }

template<unsigned n>
inline modN_t<n> operator/ (const modN_t<n>& x, const modN_t<n>& y) {
  if (y.get() == 0) throw "Division by 0";
  int u= y.get(), v= n, x1= 1, x2= 0, q, r, x0;
  while (u != 1) {
    q= v/u; r= v%u; x0= x2 - q*x1;
    v=u; u= r; x2= x1; x1= x0; }
  return modN_t<n>(x.get() * x1);
}

// definitions of +=, -= etc should be added here

int main(int, char* []) {
  typedef modN_t<127>             mytype;
  glas::def::groupMultOp<mytype>  mymult;
  mytype   v78(78), v113(113), v90(90), v80(80);

  cout << "equalResults(v78, v113,  v90, v80, mymult) "
       << equalResults(v78, v113,  v90, v80, mymult) << endl;
  cout << "equalResults(v78, v113,  v90, mytype(81), mymult) "
       << equalResults(v78, v113,  v90, mytype(81), mymult) << endl;
  cout << "identityPair(v78, mytype(-78), mymult) "
       << identityPair(v78, mytype(-78), mymult) << endl;
  cout << "identityPair(v78, mytype(57), mymult) "
       << identityPair(v78, mytype(57), mymult) << endl;
  cout << "poorMensDivision(mytype(8), mytype(2), mymult) = log_2 (8) "
       << poorMensDivision(mytype(8), mytype(2), mymult) << endl;
  cout << "poorMensDivision(mytype(35), v78, mymult) = log_78 (35) "
       << poorMensDivision(mytype(35), v78, mymult) << endl;
  return 0;
}
```

As 127 is a prime number, the set (excluding 0) is an AbelianGroup with respect to the multiplication. The results are 0, 1, 0, 1, 3, and 8.

The implementation of the division with an optimized extended Euclidean algorithm has logarithmic time complexity over the size of the set. This was the motivation to store inverse values in pure algebraic algorithms as stated in section 5.3

6 Handling Associativity and Commutativity

Associativity and commutativity cannot be verified by any compiler. We therefore introduce type traits for it, which were already mentioned in sections 4.2 and 4.3. Which operation is regarded as associative or commutative is the user's decision.

The associativity of an operation can be checked with the boolean value `glas::is_op_associative<Op>::value`, which is `false` unless explicitly defined as `true` for this type or some group of types it belongs to.

6.1 Type Traits and Markers

To simplify the implementation, we introduce markers that can be used to mark a functor as associative or commutative. The default functors, in particular, are derived from the markers, so that the code in section 5.1 is slightly extended, where only the beginning is shown here

```
namespace glas { namespace def {

  struct associativity_marker {};

  struct commutativity_marker {};

  template <class T>
  class magmaAddOp {
  public:
    T operator() (const T& x, const T& y) {
      return x + y; }
  };

  template <class T, class Base= magmaAddOp<T> >
  class semiGroupAddOp: public Base, associativity_marker {};

  template <class T, class Base= semiGroupAddOp<T> >
  class commSemiGroupAddOp: public Base, commutativity_marker {};
```

The non-specialized implementation of the type traits utilizes this to determine the properties of the default functors.

```
// File property_traits.hpp
#ifndef property_traits_include
#define property_traits_include

#include <boost/type_traits.hpp>
#include "default_functors.hpp"

namespace glas {

  template <class T>
```

```
  struct is_op_associative {
    static const bool value= boost::is_base_and_derived<
                              def::associativity_marker, T>::value;
  };

  template <class T>
  struct is_op_commutative {
    static const bool value= boost::is_base_and_derived<
                              def::commutativity_marker, T>::value;
  };

} // namespace glas

#endif // property_traits_include
```

Please notice that inheritance is only used to simplify the implementation and does not affect the generality of the utilization.

6.2 Associativity and Commutativity of User-defined Functors

To declare a user-defined functor – say `myFunctor` – as associative can be done in two ways. The first one is specialization:

```
namespace glas {
  template <>
  struct is_op_associative<myFunctor> {
    static const bool value= true; };
}
```

The second and shorter method is to benefit from the definition of the type trait and to derive the user functor from the marker:

```
struct myFunctor: glas::def::associativity_marker, ... {  .... }
```

6.3 Algebraic Functions based on Associativity and Commutativity

Algebraic functions can check if functors are associative or commutative. In case a required property is not fulfilled, an exception could be thrown, for instance:

```
template <class T, class Op>
int foo(T v1, T v2, Op op) {
  if ( ! glas::is_op_commutative<Op>::value)
    throw ``Operation must commutative in foo!\n'';   ...
```

The property can be already checked at compile time using 'Enable_if'. As an example an accumulate that sorts the values first may verify that the operation is associative and commutative.

```
// {Iter*, Op} must be a CommutativeMonoid
struct sortedAccumulate_t {
  template <class Iter, class Op, class Comp>
  typename enable_if<glas::is_op_associative<Op>::value
                     && glas::is_op_commutative<Op>::value,
                     typename iterator_traits<Iter>::value_type>::type
  operator() (Iter first, Iter last, Op op, Comp comp) {
    typedef typename std::iterator_traits<Iter>::value_type value_type;
    std::vector<value_type> tmp(first, last);
    std::sort(tmp.begin(), tmp.end(), comp);
    return std::accumulate(tmp.begin(), tmp.end(), op.identity(), op); }
} sortedAccumulate;
```

In the example, an iterator range is taken to accumulate the values w.r.t. `op` after being sorted w.r.t. `comp`. The return type of the function operator is the `value_type` of the iterator type but only in the case that the operation is associative and commutative. Otherwise the template `enable_if` has no `type` and the compilation will stop at this point.

A more robust version could call an alternative function if the properties are not fulfilled. To this purpose, we introduce another meta-template "`if_type<bool B, class T1, class T2>`", whose type is `T1` if `B` is `true` and `T2` otherwise. The function `trySortedAccumulate` tests the properties of the operation and calls `sortedAccumulate` if allowed and `unSortedAccumulate` otherwise.

```
// {Iter*, Op} must be a Monoid
struct unsortedAccumulate_t {
  template <class Iter, class Op>
  typename std::iterator_traits<Iter>::value_type
  operator() (Iter first, Iter last, Op op) {
    return std::accumulate(first, last, op.identity(), op); }
  // Only for Compability
  template <class Iter, class Op, class Comp>
  typename std::iterator_traits<Iter>::value_type
  operator() (Iter first, Iter last, Op op, Comp) {
    return operator() (first, last, op); }
} unsortedAccumulate;

// {Iter*, Op} must be a Monoid
template <class Iter, class Op, class Comp>
inline typename std::iterator_traits<Iter>::value_type
trySortedAccumulate(Iter first, Iter last, Op op, Comp comp) {
  typename if_type<glas::is_op_associative<Op>::value
                   && glas::is_op_commutative<Op>::value,
                   sortedAccumulate_t, unsortedAccumulate_t>::type
                                                              accumulate;
  return accumulate(first, last, op, comp);
}
```

As all distinctions are made at compile time and all functions are implicitly

inlined, only the sorting, the accumulation and the construction of the identity consume compute time, in as far as needed.

The user can call `trySortedAccumulate` and will always get a result as in the following example

```
bool greaterAbs(float x, float y) {
  return fabs(x) > fabs(y); }

int main(int, char* []) {
  glas::def::groupAddOp<float>                groupAdd;
  glas::def::abelianGroupAddOp<float>         abelAdd;
  float array[7] = {1.0, 8e6, -3e-4, 1e7, -8e6, 3e-4, -1e7};

  cout << "Sum with trySortedAccumulate as group (unsorted): "
       << trySortedAccumulate(array, array+7, groupAdd, greaterAbs) << endl;
  cout << "Sum with trySortedAccumulate as Abelian group (sorted): "
       << trySortedAccumulate(array, array+7, abelAdd, greaterAbs) << endl;
  return 0;
}
```

Notice that the declaration of commutativity is implicit by declaring two different functors modeling Group and AbelianGroup, respectively. Both functors have the same functionality whereas the latter is defined as being commutative. The unsorted accumulation returns 0 and the sorted version the correct value 1.[8]

7 Concepts for Additive and Multiplicative Algebraic Structures

In order to allow convenient programming for additive and multiplicative algebraic structures we focus the utilization on the operators. On the other hand, we want to define them as refinements of the corresponding pure algebraic concepts. This is realized with the default functors so that additive and multiplicative concepts are only indirectly defined on a second type.

7.1 AdditiveMagma

An *Additive Magma* is a set of elements (T) with an addition. The set must be closed under the addition

$$a, b \in T \quad \rightarrow \quad a + b \in T.$$

Refinement of

Magma

[8]Sorting in descending order works well to reduce last-bit-cancellations. Under other circumstances, sorting in ascending order is preferable. Especially for sums of floating point, there are more sophisticated techniques to calculate always the correct result in spite of rounding errors. These algorithms will be considered eventually at a later stage. For the moment, we only want to illustrate the usage of attributes.

Associated Types

- Corresponding functor
  `glas::def::magmaAddOp<T>`

Notation

`T`	is a type that models AdditiveMagma.
`a, b`	are objects of type `T`.

Definition of Refinement

The module {`T, glas::def::magmaAddOp<T>`} must be a model of Magma.

Valid Expressions

- Addition
  `a + b`
  Return Type: `T` or a type convertible to `T`
  Semantics: Can be arbitrary as long as all results are valid, which can depend on the perspective, confer section 3. By definition of the functor `glas::def::magmaAddOp<T>()(a, b)` = `a + b`.

- Addition Assignment
  `a += b`
  Return Type: `X&`
  Semantics: Equivalent to `a = a + b`.

7.2 AdditiveSemiGroup

An *Additive Semi-Group* is an additive magma where the addition is associative.

Refinement of

AdditiveMagma and SemiGroup

Associated Types

- Corresponding functor
  `glas::def::semiGroupAddOp<T>`

Definition of the Refinement

For a type `T` that models AdditiveSemiGroup the module {`T, glas::def::semiGroupAddOp<T>`} must be a model of SemiGroup. This implies that the + operator is associative.

Models

- STL strings because concatenation is defined with $+$, see section 5.4.

7.3 AdditiveCommutativeSemiGroup

An *Additive Commutative Semi-Group* is an additive commutative semi-group, obviously.

Refinement of

AdditiveSemiGroup and CommutativeSemiGroup

Associated Types

- Corresponding functor
  `glas::def::commSemiGroupAddOp<T>`

Definition of the Refinement

For a type `T` that models AdditiveCommutativeSemiGroup the module {`T, glas::def::commSemiGroupAddOp<T>`} must be a model of CommutativeSemiGroup. This implies that the $+$ operator is associative and commutative.

7.4 AdditiveMonoid

An *Additive Monoid* is an additive semi-group with an identity.

Refinement of

AdditiveSemiGroup and Monoid

Notation

`T`	is a type that models AdditiveMonoid.

Associated Types

- Corresponding functor
  `glas::def::monoidAddOp<T>`

Valid Expressions

- Identity
  `T(0)`
  Return Type: `T` or a type convertible to `T`.

Definition of the Refinement

For a type `T` that models AdditiveMonoid the module {`T, glas::def::monoidAddOp<T>`} must be a model of Monoid. This implies that `T(0)`[9] is the identity element of the addition.

7.5 AdditiveCommutativeMonoid

An *Additive Commutative Monoid* is of course an additive commutative monoid or alternatively an additive commutative semi-group with an identity.

Refinement of

AdditiveMonoid, AdditiveCommutativeSemiGroup and CommutativeMonoid

Associated Types

- Corresponding functor
  `glas::def::commMonoidAddOp<T>`

Definition of the Refinement

For a type `T` that models AdditiveCommutativeMonoid the module {`T, glas::def::commMonoidAddOp<T>`} must be a model of CommutativeMonoid.

Models

- Contingent: `unsigned int`, see section 3.
- Contingent: the `age` example in section 5.2 with operator.

7.6 AdditiveGroup

A *Additive Group* is an additive monoid with an inverse function, which is the unary minus. The binary minus is only a shortcut of the addition with an inverted value.

Refinement of

AdditiveMonoid and Group

Notation

`T`	is a type that models AdditiveGroup.
`a, b`	are objects of type `T`.

[9] **Question:** Should be checked if this always works (besides for strings, which are not crucial for an algebra library). Otherwise using type traits (with T(0) as default).

Definition of Refinement

The module {`T, glas::def::groupAddOp<T>`} must be a model of Group. This implies that `a - a` is `T(0)` for all a.

Valid Expressions

- Inverse
  `-a`
  Return Type: `T` or a type convertible to `T`.
  Semantics: Implied by the definition of Group.
- Subtraction
  `a - b`
  Return Type: `T` or a type convertible to `T`
  Semantics: Equivalent to `a + -b`.
- Subtraction Assignment
  `a -= b`
  Return Type: `T&`
  Semantics: Equivalent to `a = a + -b`.

7.7 AdditiveAbelianGroup

An *Additive Abelian Group* is an additive commutative group or alternatively an additive commutative monoid with an inverse function.

Refinement of

AdditiveGroup, AdditiveCommutativeMonoid and AbelianGroup

Associated Types

- Corresponding functor
  `glas::def::abelianGroupAddOp<T>`

Definition of the Refinement

For a type `T` that models AdditiveAbelianGroup, the module {`T, glas::def::abelianGroupAddOp<T>`} must be a model of AbelianGroup.

Models

- `modN_t<n>`, see section 5.6.
- Contingent: `int`, see section 3. In addition, the smallest value in any signed `int` format is its own inverse as a result of the 2-complement definition. The program fragment, assuming `int` is 32 bit,

```
int a= -2147483648;
cout << sizeof (a) << ' ' << a << ' ' << -a << endl;
```

returns "32 -2147483648 -2147483648".

- Contingent: `float`, see section 3.
- Contingent: `complex<double>`, same as `float`.
- Matrices of dimension $n \times m$, if the underlying element type models AdditiveAbelianGroup. In general, the matrix always models the same additive concept as the element type.

7.8 Multiplicative Algebraic Structures

The definitions of these concepts are analogical. 'Multiplicative' replaces 'Additive' in all names, respectively 'Mult' replaces 'Add'. In all definitions and programs 0 is substituted by 1, + by *, and - by /.

Models

- `modN_t<n>` is a MultiplicativeAbelianGroup if `n` is a prime number and a MultiplicativeCommutativeMonoid otherwise, see section 5.6
- Contingent: `int` is a MultiplicativeCommutativeMonoid because reciprocals are not defined (except for 1 and -1), see section 3.
- Contingent: `float` is a MultiplicativeAbelianGroup ignoring problems with extreme values, see section 3 for restrictions and section 5.5 for an example.
- Contingent: `complex<double>` is a MultiplicativeAbelianGroup, same as `float`.
- Matrices of dimension $n \times m$ does not model MultiplicativeMagma if $n \neq m$.
- Matrices of dimension $n \times n$ model MultiplicativeMonoid, given the element type models too.

8 Algebraic Structures with Two Operations

The concept definitions for these algebraic structures – like ring or field – are straight-forward refinements of additive and multiplicative concepts from section 7.

8.1 Ring

A *Ring* is a set of elements with an associative multiplication[10] and an addition that fulfills the requirements of an Abelian group.

[10] In the literature exist different definitions of rings differing in the inclusion of the multiplicative identity element. We prefer the definition of ring without an identity because it allows a finer grained conception.

Refinement of

AdditiveAbelianGroup and MultiplicativeSemiGroup.

Invariants

- Pre-Distributivity
  `a * (b + c) = a*b + a*c`

- Post-Distributivity
  `(b + c) * a = b*a + c*a`

8.2 RingWithUnity

Adding a multiplicative identity to a ring defines a *Ring With Unity*.

Refinement of

Ring and MultiplicativeMonoid.

8.3 CommutativeRing

Adding commutativity w.r.t. the multiplication completes a *Commutative Ring*.

Refinement of

Ring and MultiplicativeCommutativeSemiGroup.

8.4 Field

A *Field* is a ring with a multiplicative Abelian group.

Refinement of

CommutativeRing, RingWithUnity and MultiplicativeAbelianGroup.

9 Future Work

The algebraic structures here are only a part of linear algebra concepts we plan to consider, confer also figure 1. To reduce the length of names, some of them are abbreviated in an obvious manner, e.g., Commutative to Comm or Multiplicative to *.

The concepts described so far, cover the part of scalar-like types. Whether a type is scalar-like is relative to the context. `complex<double>` can be considered as a space over `double` or as scalar-like with, for instance, `vector<complex<double> >` as a space over it. In the same way, R-modules can be defined over $n \times n$

matrices so that the $n \times n$ matrices are regarded as scalars whereby in other contexts these matrices can be considered as operators or as spaces.

- Associativity and commutativity of addition and multiplication in additive and multiplicative structures.
- Checks whether a certain type models a certain concept, which might simpler for some algorithms than checking attributes.
- Spaces: vector spaces, Banach spaces, ...
- Operators: linear operator, finite linear operator, ...
- Concepts for distributed data types or types enabling multi-tasking access.
- Concepts to represent imprecision or explicitly committed exactness.
- More meaningful examples, hopefully.

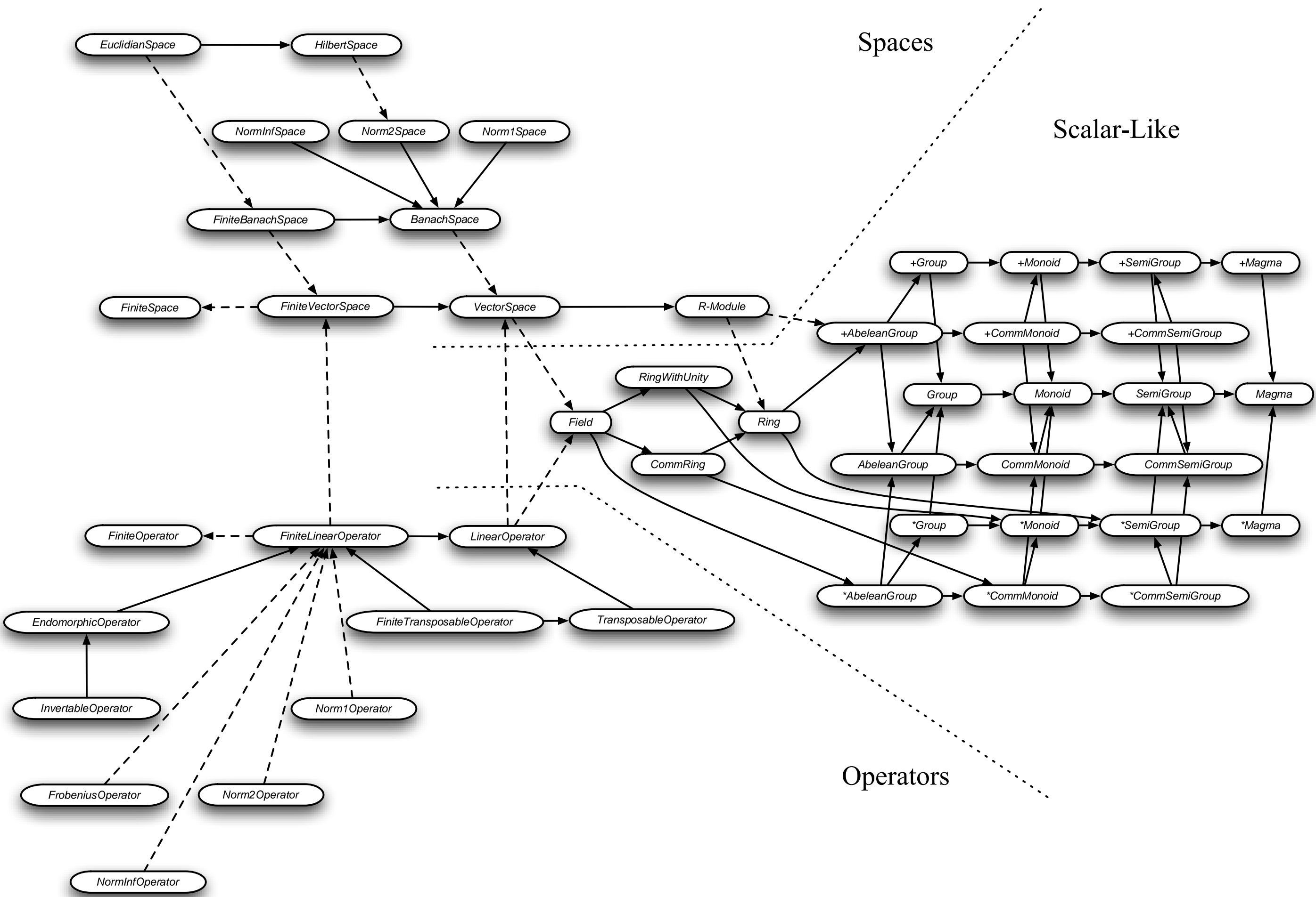

Figure 1: Linear algebra concepts