Fundamental Algebraic Concepts in Concept-Enabled C++

Peter Gottschling

November 9, 2006

Contents

Part I
Description

Chapter 1

Introduction

Using generic programming concepts to define algebraic structures creates a symbiosis between generic programming and algebra. Mathematicians use the methodology of concepts to focus on essential properties of algebraic structures. All properties that are not relevant in a given context are ignored. In theoretical investigations, no specific objects or sets are examined but abstract objects holding certain properties. From these initially postulated properties, other properties are deduced without taking specific objects into consideration.

In the same manner, generic functions are not limited to a specific type but are applicable to every type with the appropriate properties. These properties are given as template function constraints in form of concepts. Concepts formally characterize the requirements of a data type to be applicable to a generic implementation of an algorithm. In this paper, we formalize the mathematical properties of algebraic structures as generic programming concepts in order to constrain C++ generic functions with mathematical stringency.

In some sense, these algebraic concepts establish a symbiosis between algebra and generic programming. At first, the idea of concepts was adapted from algebra to generic programming. Now, algebraic structures are specified with the generic programming concepts.

Historically, concepts in C++ played only the role of documentation and it was the user's responsibility to check whether his data type models a concept. Siek and Lumsdaine [13, 17] started to involve compilers in this checking process by introducing prototypes of concepts that access all associated types, operators, and functions of these concepts so that the instantiation of such a prototypes shows whether a data type fulfills all requirements of a concept.

Currently, research is going on to introduce concepts into the C++ standard [5, 14, 20]. Gregor started implementing concept checking in the gcc-based compiler ConceptGCC according the latest revision of the standard proposal on language extension for concepts. A first prototype is available for testing [6]. The concept implementations presented in this paper are compilable with the current version of ConceptGCC.

The intention behind integrating concepts into compilers was mainly to clarify error messages. However, we consider it even more important that the concept checking extends the error detection to semantic errors that are completely transparent to compilers without concept checking. As a consequence, concept checking not only helps to understand error messages better. It also provides a completely new ability to establish mathematical reliability in software because the compiler can check the semantic constraints when they are stated in the program. We consider this approach of compiler-supported semantic verification and of exposing semantic behavior in program sources as a new paradigm that we call *Property-Aware Programming*.

The definition of algebraic concepts is only the first step of establishing this new form of mathematical software reliability. However, it is really a fundamental step. The specific objectives of introducing algebraic concepts are to

1. Express algebraic concepts in the same fashion as STL concepts [18];
2. Implement these concepts in ConceptGCC;
3. Provide additive and multiplicative concepts that are based on operators $+$ and $*$;
4. Introduce algebraic concepts definable for an arbitrary binary operations;
5. Relate all these concepts to each other so that additive and multiplicative concepts are refinements of concepts with arbitrary operators;
6. Provide concepts with two connectors, like rings, that are refinements of additive and multiplicative concepts in an operator-based definition;
7. Establish alternative two-connector concepts that are defined on two arbitrary binary operations in the generic definition;
8. Discuss the behavior of intrinsic numeric C++ types;
9. Illustrate the applicability on examples;
10. Provide theoretical background in order to relate concepts with different numbers of template parameters to each other in a mathematically correct manner.

Chapter 2

Concepts and Models in Generic Programming

Generic software in terms of templated functions or templated function objects provides the calculation of an algorithm to the whole set of types that are admissible to the algorithm and its implementation. Types that do not match the requirements of a generic function syntactically cause compiler errors. Unfortunately, the error messages are often extremely long in heavily-templated libraries and not necessarily very helpful. The reason is that the messages are generated after template instantiation and possibly appear in a indirectly-called function (the one that could not be instantiated). This function can be unknown to the user and because many errors take place in internal helper functions, there might not even be a documentation for this function. Thus, it is often extremely difficult to relate the error message to the cause of the problem. Even more critical than syntactic mismatches are violations of semantic requirements. They are completely transparent to current compilers and executables are generated that compute incorrectly.

Because of the difficulties to interpret error messages caused by syntactic errors and of the necessity of probably time-consuming debugging in case of semantic errors, it is very important to assure that all requirements of a generic function are fulfilled when called in a program. For this reason, good generic libraries are equipped with very detailed specifications of syntactic and semantic demands of each function in form of concepts.

Definition 1 *A* concept *is a set of* syntactic *and* semantic *requirements on a type or a tuple of types. A type or tuple of types that fulfill all requirements is called a* model*. A concept that adds requirements to another concept is called* refinement*.*

The requirements defined in concepts consist of:

Valid Expressions: compilable C++ expressions that the type must provide;

Associated Types: other types than the modeling related to the concept;

Invariants: semantic properties, which can be required for all objects of the type(s) — like associativity — or required on function arguments as precondition — like symmetric values of a matrix — and;

Complexity Guarantees: maximum limits on compute time and memory needs often depending on complexity requirements on the parameters.

Valid expressions and associated types are syntactic requirements checkable by the compiler and their violation will be detected (unless implicit conversion cause accidental syntactic mismatching). Invariants cannot be verified by the compiler. The only way to detect violations of semantic requirements are run-time tests. This does not formally prove the correctness of programs but decreases the probability of erroneous executions.

For a more detailed description see for instance [15, chapter 2].

2.1 Concept Checking in C++

Concepts of generic C++ libraries are currently described as textual documentation and it is the users' responsibility to verify whether his types model the concepts of a generic function. An extension of C++ is proposed to the standard committee in order to use compilers for checking concepts. Integrating concepts into the C++ has several important advantages:

- Verification of type properties occurs at function call and so errors are not detected inside the implementation but at erroneous function calls;
- Error messages are generated in terms of concept requirements instead of obfuscated errors within an unknown context, and
- Verification is provided whether the concepts of a function cover its semantic and syntactic requirements including all functions called by it.

ConceptGCC is the first prototype of a C++ compiler that provides concept checking. It is based on gcc and implements the extensions proposed for the next standard. In our concept implementations, we use the syntax of the standard proposal. All programs are compiled with ConceptGCC. The details of ConceptGCC are given in [4]. In this chapter, we sketch the new language features used in this text on some examples (the examples are not part of algebraic concepts of the paper).

```
concept GroupExample<typename Operation, typename Element>
  : SemiGroupExample<Operation, Element>
{
    typename inverse_result_type;
    inverse_result_type inverse(Operation, Element);
    where std::Convertible<inverse_result_type, Element>;
    Element operator+=(Element&, Element);

    axiom Inversion(Operation op, Element x)
    {
        op( x, inverse(op, x) ) == identity(op, x);
        op( inverse(op, x), x ) == identity(op, x);
    }

    axiom SomeCondition(Operation op, Element x)
    {
        if ( x != identity(op, x) )
            op(x, x) != x;
    }

    typename magnitude_type = Element;
};
```

The syntax of concepts is similar to the syntax of classes and structs. The example above introduces a concept called GroupExample over the two types Operation and Element. The concept is a refinement of the concept SemiGroupExample. Both are defined over the same types.

The second line is the request for a free function or function object with the given signature. The result type of the function can be specified or, as in our example, automatically detected. It is then related to the associated type introduced in the first line. *Where* clauses are used to put more constraints on the modeling and the associated types, as in the above example to require the result type of the function inverse to be convertible into the Element type.

Furthermore, it is requested that the **operator**+= exist (i.e. x += y; is a valid expression). Its first parameter must be a non-const reference and the second one can be passed arbitrarily. In signatures within concept, value parameters and const reference parameters are equivalent, i.e. the signature

```
Element operator+=(Element&, const Element&);
```

is equivalent to the one in the concept.

Invariants are defined in *axioms*.[1] Axioms define properties that are held by all objects of a type, (e.g., that the result of a binary operation between some element and its inverse is the identity of this operation like in the sample concept above). Currently, axioms have no impact on the executable. The axioms are intended to be used for customizable optimization. Another possible utilization is to generate a set of tests to check whether the axioms are held for a set of random tuples. These would of course not prove the correctness of programs formally but decreases the probability of erroneous programs—to what extend would be strongly application dependent. Automatically generated concept-based run-time tests are subject to future research.

A precondition is a required property of a single object that is used as function argument and do not need to hold on all objects of a certain type; for

[1]The 'axioms' are actually no axioms in the mathematical sense. They are (mathematical) properties in general because these 'axioms' are not required to be a minimal set of theorems.

instance, $x \geq 0$ can be claimed for all unsigned integers but not for all signed integer values or all float values. Expressing the demand of non-negativeness as a type constraints would be overly restrictive and unnecessarily limit the applicability of a generic function. Therefor, this requirement is more appropriately specified as precondition.

The last statement in the example code introduces an associated type, which is by default the type Element unless it is declared by the user in a concept map. A **concept_map** is a declaration by the user that a certain type models a concept, for example

```
template <typename T>
    where Float<T>
concept_map GroupExample<math::add, T>
{
    typedef long double magnitude_type;
}
```

The concept map above is not a model declaration for a single type but is templated itself. For an arbitrary type T that models the concept Float the tuple (math::add, T) is a model of GroupExample. Furthermore the associated type 'magnitude_type' is defined differently from the default for these types. Declaring a type or a tuple of types a model of a concept causes the compiler to verify that all syntactic requirements are fulfilled. In the example concept it will be checked whether a plus operator exists and whether inverse is defined as a free function or as a function local to the concept to be defined in the concept map. All associated types must be defined in the concept map unless there is a default in the concept or they can be resolved by automatic type detection.

These concepts are used to constrain template functions, like in the following example:

```
template <typename Op, typename Element, typename Exponent>
    where math::GroupExample<Op, Element> && std::Integral<Exponent>
inline Element multiply_and_square(Element base, Exponent exp, Op op)
{ /* ... */ }
```

Calling this function with a tuple (base, exp, op) requires that the types of op and base model GroupExample and that Exponent models std::Integral which means that Exponent is a **signed** or **unsigned** integer. For types not modeling these concepts the compiler will not call the function but inform the user that the function is a candidate for his call and which requirement is not fulfilled. For a more detailed description, see for instance [4]

2.2 Portability to Compilers without Concepts

For practical reasons it is important that programs containing concepts and concept maps are backward compatible to compilers without concept checking. Concepts and concept maps are therefore conditionally defined only when concept checking is available. Template function constraints are replaced by the macro function LA_WHERE. This function evaluates to a where clause in compilations where concept checking is available and to an empty string otherwise. The only feature that needs to be supported in the concept-free software is the specification of associated types; these have to be defined as type traits. The access to associated types in code that is portable between concept-checking and concept-free compilers is much more convenient if the type traits have the same names as the concepts. Although this approach provides portability, the double definition raises a high risk of inconsistency between the concept-enabled and the concept-free version.

Chapter 3

Algebraic Concepts

The algebraic concepts introduced in this chapter can be distinguished between structures with one operation — like monoid or group — and structures with two operations — like ring or field. Another distinction between concepts is the genereicity of the definition: allowing the operation(s) being freely chooseable in terms of type parameter or being fixed as addition and/or multiplication. This different representation of operation is characterized by the following concept categories.

3.1 Concept Categories

Concepts with one operation are expressible in different two manners: with a given or with a freely chooseable operation. We will first introduce the general concepts where the operation is implemented with a functor realizing an arbitrary binary operation. This category of functor-based concepts is split into two sub-categories: one defining only mathematical properties in the concepts and the other adding very basic implementation requirements to facilitate their utilization in defining generic functions constraints.

Later we define concepts specialized to addition and multiplication as the most important binary operations. These concepts use operators instead of functors because this is the common practice in numerical libraries.

Thus, we have the following categories of concepts:

- Functor-based concept:
  - Purely algebraic concepts;
  - Augmented concepts, and
- Operator-based concepts.

These categories are used in the same manner for one-operation and two-operation concepts. For the sake of brevity, we will in the following omit the grouping into functor-based concepts and distinguish between three categories: purely algebraic, augmented, and operator-based concepts.

3.2 One-operation Concept Overview

In this section, we only sketch the concepts and discuss them later in more detail. The algebraically most general one-operation concept is a magma, which is a structure that requires only the closure of the considered operation. Semigroup adds associativity and monoid the identity element and related properties. Structures with inversion are called in this paper partially invertible monoid, short PIM, or group depending whether the inversion is defined on all elements or not. For all these concepts exist also a commutative version. Figure 3.1 depicts all one-operation concepts.

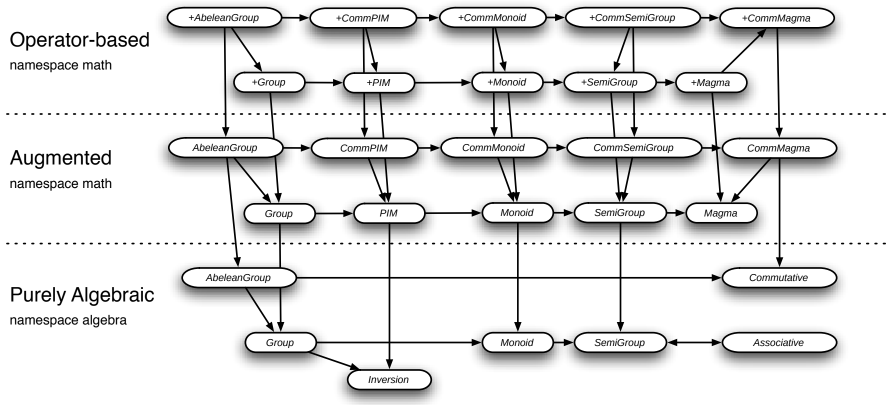

Figure 3.1: Overview of one-operation concepts.

3.3 Purely Algebraic and Augmented Concepts

In this paper we define an *Algebraic Structure* as a finite or infinite set of elements and one or more functions defined on elements of the set. In contrast to algebraic structures with an addition or multiplication as operation, we call algebraic structures with an arbitrary binary operation purely algebraic structures. 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, see. Section 8.1. The concepts in this section are strictly limited to mathematical properties. They are all defined in the namespace algebra.

Using these concepts to constrain template functions demands in most cases additional constraints regarding *assignability* and *convertibility* of the element type and results of operations. *Augmenting* the concepts with the requirements Assignable and Convertible enables clearly more concise definitions of function constraints as illustrated in Section 4.1.6. Except the two basic concepts Convertible and Assignable we do not use any other non-mathematical concept in any algebraic concept. As the augmented concepts have the same names as the purely algebraic ones, they are defined in the namespace math to avoid conflicts.

3.3.1 Magma

The concept *Magma* – also called groupoid[1] – introduces a binary operation on the set and its closure. The closure of a binary operation is defined so that for a set S of elements of type Element and an operation op of type Op, the operation must be defined on all $(a, b) \in S \times S$ and the result must be part of the set S

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

Unfortunately, this is not expressible in a clean mathematical style within the language. Due to this lack of correct definability, we exclude magma from the purely algebraic concepts in namespace algebra.

The definition of magma in namespace math represents the closure by means of the return type's convertibility into the element type. Requiring that the return type be the Element type would simplify the concept definition but limit the genereicity. As an example, expression templates could not be used when the return type of an operation was demanded to be equal to the element type.

For convenience, we also add the requirement of assignability. Mathematically, this is not necessary and even on the implementation side one could as well restrain from it and instead request the assignability in the template function where needed. However, in most cases the Magma concept will come together with Assignable and so that we included it into Magma.

3.3.2 Closure of Operations on Arithmetic Data Types

Since arithmetic operations are in almost all cases performed on hardware-supported floating point or integer types, we discuss in this section how these data types model the the most fundamental algebraic concept. First we consider the addition of **float**, **double**, or **long double** values that can result in overflow. In arithmetic compliant with the IEEE Standard 754 this results in special values '$+\infty$' and '$-\infty$'. Regarding these special numbers as legal values of type Element, all floating points are closed under addition and therefore are perfect models of Magma. However, it is safe to assume that in almost all cases floating point numbers are considered as approximation of real numbers $\mathsf{x} \approx x \in \mathbb{R}$. Taking this into account, the result $2 \cdot 10^{38} + 2 \cdot 10^{38} = \infty$, computed as 4-byte **float**, one can see that the addition of real numbers is not approximated correctly. The same is true for approximating the multiplication of real number. Besides overflow, the multiplication of floating point values is also falsified by the fact

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

that the product of very small non-zero values result in zero using computer arithmetic.

Thus, floating point numbers of different formats only approximate $\mathbb{R}$ as long as no overflow, underflow or rounding to zero occurs. Ignoring the approximation of real numbers, the closure of addition and multiplication can still be considered formally. In this case, the result ∞ as the sum of two large numbers is then perfectly legal as well as rounding to 0 are. The only cases where the set of legal values is left is $\infty + -\infty = \text{NaN}$, $-\infty + \infty = \text{NaN}$, and $\pm\infty * \pm 0 = \text{NaN}$, unless one regards the special IEEE 754 value NaN—Not a Number—as a legal result.

Accordingly, signed **int**s only behave like integer numbers $\mathbb{Z}$ and **unsigned int**s like natural numbers $\mathbb{N}$[2] if no overflow or underflow happen. Another prospective of **int**s is seeing them as cyclic groups $\mathbb{Z}_n$. An unsigned k-bit **int** is isomorphic to $\mathbb{Z}_{2^k}$ regarding addition and multiplication (e.g., an unsigned 32 bit **int** represents a cyclic set from 0 to 4,294,967,295). Furthermore, the set of signed k-bit **int** $\mathcal{S}_k$ is isomorphic to the set of unsigned k-bit **int** $\mathcal{U}_k$ whereby

$$s \in \mathcal{S}_k \equiv \begin{cases} 2^k - s & \text{for } -2^{k-1} \leq s < 0 \\ s & \text{for } 0 \leq s < 2^{k-1}. \end{cases} \tag{3.1}$$

For instance with 32 bit values, -2,147,483,647 is equivalent to 2,147,483,649, -1 to 4,294,967,295, ... The operations implemented in processor hardware and therefore realized in C++ operators holds the signed/unsigned isomorphism for addition, multiplication, and subtraction but not for division.

Regardless of whether the results of **int** operations are considered being correct when overflow and underflow occur, the set of values is never left and the requirements of Magma are thus perfectly fulfilled. Resuming, the arithmetic types model the formal concept definitions but the approximation of the corresponding sets of numbers is only valid in certain ranges of values.

3.3.3 CommutativeMagma

This concept adds the commutativity to a closed binary operation. We introduce it to provide a more precise characterization of floating point numbers, which are commutative but not associative, see. SemiGroup. The lack of associativity is caused by the finite precision, a property that is more investigated in computer science than in algebra. This might be the reason that CommutativeMagma is used as a concept in mathematics.

However, before considering to declare arithmetic on floating point numbers as CommutativeMagma, the programmer should read Section 4.3.2 and examine whether disavowing associativity improves the programs' numerical behavior.

3.3.4 SemiGroup and CommutativeSemiGroup

Semi-group is defined in two concepts: the pure mathematical version algebra::SemiGroup and the augmented version SemiGroup with basic implementa-

[2] Assuming $0 \in \mathbb{N}$

tion behavior in namespace math.

Commutative semi-group CommutativeSemiGroup has no definition in algebra because this structure is not common in mathematical textbooks but is convenient for numerical software.

3.3.5 Monoid and CommutativeMonoid

A monoid is a semi-group with an identity. The identity element depends both on the operation and on the element type. Earlier versions of the concept referred to the operation as template parameter assuming the operation to be state-less. In the current version, the operation is passed as function parameter, which not only allows the operation to have a state but also simplifies the syntax.

Referring to an element in the identity function is not necessary for scalar types. To be applicable to more complex types, like vectors and matrices, it is necessary to access run-time information (e.g., dimension). Another example, distributed data types, require information on the distribution in order to return a compatible identity element, for example string in Section 10.1.

The identity function is implemented using a functor named identity_t. From our experience, template class specialization provides a more comprehensible behavior than template function overloading. Therefore, we recommend the programmer when establishing new identity elements to specialize identity_t instead of overlapping identity.

Similar to semi-group, monoid is defined in two versions: algebra::Monoid and Monoid in math whereas CommutativeMonoid is only defined in math.

3.3.6 PartiallyInvertibleMonoid

The concept PartiallyInvertibleMonoid, short PIM, introduces inversion and a check whether a given element is invertible. Even if it does not exist in mathematical literature, we propose it for implementation purposes in order to build a bridge between Monoid and Group. With PartiallyInvertibleMonoid, one can use inversion even when not all elements are invertible. The best-known example is division by zero but other monoids can have more than one non-invertible element, for instance cyclic sets of non-prime size with regard to multiplication. Another example is the inversion of square matrices where matrices other than the zero-matrix are not invertible. Therefor, all models of this concept must provide the function is_invertible.

Obviously, no **int** type, neither signed nor unsigned, provides inverse elements for the multiplication (with the exception of 1 and -1) consistent with the arithmetic of $\mathbb{Z}$ and $\mathbb{N}$. However, the picture is different if the integral types are considered as representations of cyclic groups $\{0, \ldots, 2^k - 1\}$ with k the size of the type in bits. Since all odd numbers are co-prime to the cycle, they have a reciprocal value, i.e. for every given odd number a exist a unique value b such that $a \cdot b = b \cdot a = 1$. For instance, the reciprocal of 3 with respect to the product of 32-bit **unsigned int** is 2,863,311,531. The isomorphism (3.1) between equally-sized signed and unsigned **int** also applies to multiplication, and the reciprocal of 3 with regard to 32-bit **signed int** is thus -1,431,655,765. Although **int** types formally model PartiallyInvertibleCommutativeMonoid regarding multiplication, the practical value is highly questionable. So, we refrain from including it in the standard model declarations. Besides, these modular reciprocals are inconsistent with the division operator. The reason we discussed it in this paragraph is the ironic fact that this accidental model provides a cleaner behavior than all models involving floating point types (which are the most commonly used arithmetic types).

3.3.7 Group and AbelianGroup

Groups and commutative groups, called Abelian groups, provide the same functions as PIMs. The difference is that all elements are invertible. The function is_invertible can therefor be defined by default to return always true so that invertibility tests can be removed during compilation.

3.4 Additive and Multiplicative Concepts

Operations used in pure algebraic and augmented concepts must be callable objects and are either functions or objects of a class with an application operator **operator**()(/*... */). A short term for such an object is *functor*.

We distinguish here between two types of functors: primary functors that define the semantic itself and derived functors that are based on the semantic of an operator (i.e. **operator**+ and **operator***). Whereby primary functors serve to provide as many operations as possible for a given element type, derived functors are used to relate additive and multiplicative concepts to pure algebraic concepts. Considering how inaccurately floating point operations model certain algebraic concepts, we will show how derived functors can control which concepts are modeled by floating point addition and multiplication.

3.4.1 Default Functors and Functions

Many types already provide operators for addition and/or multiplication. These operators, **operator**+ and **operator***, can be used to define the operation type needed for purely algebraic and augmented concepts. The semantic of these functors is of course identical with the semantic of the operators; only the syntax is different.

```
template <typename Element>
struct add : std::binary_function<Element, Element, Element>
{
    Element operator() (const Element& x, const Element& y)
    {
        return x + y;
    }
};
```

```
template <typename Element>
struct mult : std::binary_function<Element, Element, Element>
{
    Element operator() (const Element& x, const Element& y)
    {
        return x * y;
    }
};
```

The definition is similar to the functors add and times in the standard library. We chose to redefine them in namespace math for better adaption to the default return types. Especially the addition/multiplication of two **short int**/**char** returns an **int** not a **short int**/**char**. Therefor, the functors for these two types are specialized to return **int** instead of the arguments' type (other than in the standard library).

We also provide a functor to compute the minimum and maximum of two values; the maximum can be implemented in the following way:

```
template <typename Element>
struct max : public std::binary_function<Element, Element, Element>
{
    Element operator() (const Element& x, const Element& y)
    {
        return x >= y ? x : y;
    }
};
```

Notice that min and max in STL are template functions and not functors.

Similarly to the default functors for the binary operations, we introduce defaults to provide the identity element of certain functions. The additive identity element of most numeric types is typically an appropriate representation of zero. For this reason, we define the default additive identity as '0' converted to the corresponding element type. This approach covers all standard numeric types in C++. The identity of vectors and matrices is correctly computed if the '0' is converted into an object consisting completely of 0s. Nevertheless, there can be numeric types whose identity element cannot be calculated by converting 0 and the identity must be provided by the user.

The conversion of '1' into a standard numeric C++ types enables the correct multiplicative identity element for these types. Providing the multiplicative identity for vectors and matrices will most likely require a user-defined implementation. Notice that the identity is only used in the context of this paper when the types (Operation, Element) are declared to model the Monoid. Conversely, the declaration of a Monoid requires a user-defined computation of the identity element if the default implementation is not appropriate.

Objects returned by identity may depend on run-time information, especially in case of vectors and matrices. Therefore, an element is passed as reference to access information like vector dimension.

We omit the implementations for multiplication and minimum because they are equivalent to addition and maximum that are shown in the following listing:

```
template <typename Element>
struct identity_t< add<Element>, Element >
  : public std::binary_function<add<Element>, Element, Element>
{
    Element operator() (const add<Element>&, const Element& ref) const
    {
        Element tmp(ref);
        tmp= 0;
        return tmp;
    }
};

template <typename Element>
struct identity_t< max<Element>, Element >
  : public std::binary_function<max<Element>, Element, Element>
{
    Element operator() (const max<Element>&, const Element& ref) const
    {
        using std::numeric_limits;
        return numeric_limits<Element>::min();
    }
};

template <typename Operation, typename Element>
inline Element identity(const Operation& op, const Element& v)
{
    return identity_t<Operation, Element>() (op, v);
}

template <typename Element>
inline Element zero(const Element& v)
{
    return identity_t<math::add<Element>, Element>() (math::add<Element>(), v);
}
```

Alternatively to the implementation above, the operation can be defined as a template parameter instead of a function parameter (and an earlier version was in fact implemented in this manner). We favor a function parameter since this allows us to define operation types containing states and, more importantly, because it simplifies the syntax. The two-level implementation with a free function and a functor underneath is chosen because according to our experiences overloading highly templated functions bears a higher risk of accidently accessing a wrong version than when specializing templated types.[3] Thus, we recommend users to specialize identity_t instead of overlapping identity according to our experiences. As short cuts for multiplicative and additive identity elements, we provide the unary functions one and zero.

[3]Without going into detail, templated function overloading is a two-phase process and a more complex behavior needs to be considered.

The default functor inverse is based on identity and the operators for subtraction and division, respectively. Therefore, modifications on these operators or on the identity change the result of inverse. If the operators $-$ and / exist, and assuming that they are consistently defined with regard to + and $*$, specialization will not be needed in most cases. However, in absence of one these operators one can still define the inverse directly (e.g., matrices usually do not provide a division but one can implement an inverse function directly). Another reason for specializing inverse (i.e. the underlying type inverse_t) can be that the default using both identity and $+/*$might be significantly more expensive than a direct implementation.

```
template <typename Operation, typename Element>
struct inverse_t {} ;

template <typename Element>
struct inverse_t< add<Element>, Element >
  : public std::binary_function<add<Element>, Element, Element>
{
    Element operator()(const add<Element>&, const Element& v) const
    {
        return zero(v) − v;
    }
};

template <typename Operation, typename Element>
inline Element inverse(const Operation& op, const Element& v)
{
    return inverse_t<Operation, Element>() (op, v);
}
```

The default of the function is_invertible is defined with the same signature as identity. The default behavior for additive structures is to return always **true** and for multiplicative structures, the element is tested whether it is non-zero.

```
template <typename Operation, typename Element>
struct is_invertible_t {};

template <typename Element>
struct is_invertible_t< add<Element>, Element >
  : public std::binary_function<add<Element>, Element, Element>
{
    bool operator() (const add<Element>&, const Element&) const
    {
        return true;
    }
};

template <typename Element>
struct is_invertible_t< mult<Element>, Element >
  : public std::binary_function<mult<Element>, Element, Element>
```

```
{
    bool operator() (const mult<Element>&, const Element& v) const
    {
        return v == zero(v);
    }
};

template <typename Operation, typename Element>
inline bool is_invertible(const Operation& op, const Element& v)
{
    return is_invertible_t<Operation, Element>() (op, v);
}
```

This behavior can be changed for a certain type and operation by specializing the underlying functor is_invertible_t. The functions inverse and is_invertible are only used with the concept PartiallyInvertibleMonoid and its refinements.

3.4.2 Relating Operator and Functor-Based Concepts

The operator-based *additive* and *multiplicative* concepts are specializations of the functor-based algebraic concept and should, therefore, be programmed as refinements. To build the bridge between the former, multi-type concepts, and the latter, single-type concepts, the functors math::add and math::times are used. Thus, every type that models a corresponding additive/multiplicative concept implicitly models the corresponding purely algebraic concept with regard to the functor math::add/math::times. The consistency of the calculations is expressed in the axioms.

3.4.3 On Multiplicative Concepts

Floating point numbers provide a rather symmetric range of exponents so that most values are invertible. However, because of representation constraints this symmetry will never be perfect and there will be either some very large values whose reciprocals will evaluate to 0 or some very small values whose reciprocals will turn into infinity. The latter will be the case with floating points represented in the meanwhile common IEEE 754 standard, especially when inverting small denormalized numbers. To avoid critical rounding towards 0 or $\pm\infty$, one can specialize the function is_invertible in MultiplicativePartiallyInvertibleCommutativeMonoid accordingly. For performance reasons, we keep the simpler test of non-zeroness as default.

Most arithmetic data types will not model MultiplicativeGroup or MultiplicativeAbelianGroup because at least the identity of the corresponding addition will not be invertible. However, since **operator**$*$ can be arbitrary for user-defined data types, one can always define a type that model MultiplicativeGroup, especially when implementing examples from algebra text books like Kleinsche Vierergruppe (Klein's four-group), which are often specified with $*$ as operator. It is also imaginable to define data types that explicitly exclude 0 and other non-invertible elements. As a rather academic consideration: one could implement floating point or integer addition in user types with **operator**$*$; this is formally perfectly legal concerning the concepts but the practical impact would be limited to a fair amount of confusion.

3.5 Two-Operation Concepts

These concepts specify algebraic structures with an additive and a multiplicative operation—either defined in terms of an operator or as functor for more genereicity. There is a general agreement that the additive structure is always an ABELIAN group; different opinions exist regarding the multiplication.

In the definition of these concepts we oriented on definition from text books, like the one from VAN DER WAERDEN [21], and also on the formal definitions in Tecton [10]. In these documents, rings are defined as semi-groups with respect to multiplication. Other authors like BOURBAKI define rings as multiplicative monoids. We call such structures 'ring with identity'. Adding the notion of division, i.e., the inversion with regard to multiplication, yields a algebraic structure called division ring or synonymously skew-field. A commutative division ring is called field. Figure 3.2 summarizes these relationships.

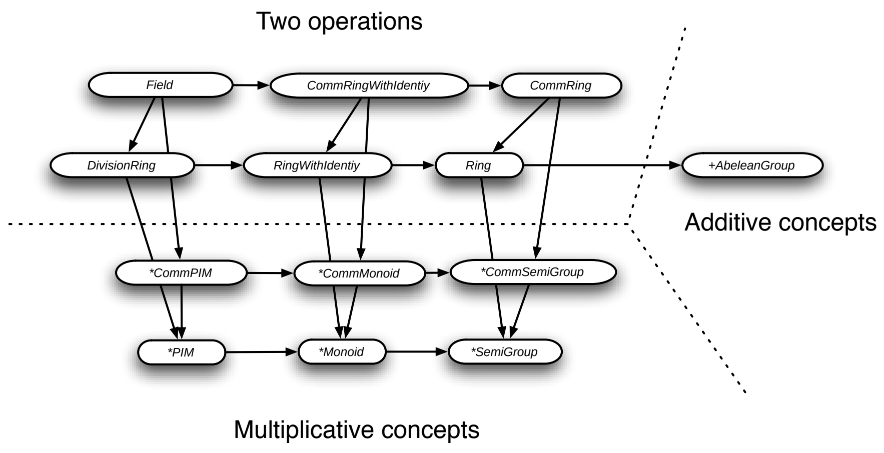

Figure 3.2: Overview of two-operation concepts.

The two-operation concepts are also organized in the following three categories:

- Purely algebraic concepts,
- Augmented concepts, and
- Operator-based concepts.

The first category is strongly oriented on mathematical textbooks and we defined there only concepts that are common for mathematicians. The concepts in the other categories are slightly more numerous. Figure 3.3 shows which concepts exist in the categories.

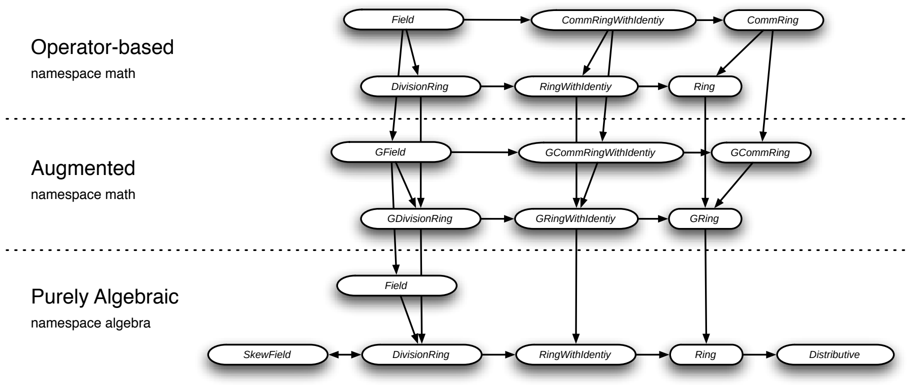

Figure 3.3: Organization of two-operation concepts.

3.6 Default Model Declarations

All algebraic concepts except Magma and its operator-based equivalents demand model declaration due to their semantic requirements (i.e. axioms which cannot verified by the compiler). Fortunately, it is not necessary to write a concept map for each combination of type and concept. When a type or a tuple of types is declared to be a model of a concept C then the mechanism of nested model declaration implicitly declares all models of concepts where C is a refinement. In this respect, **where** clauses in concepts differ from refined concepts, in both cases the same requirements are added to a concept but the model declaration of the refinement *implies* the model declaration of the concept refined from whereas a where clause *verifies* whether a model declaration of the referred concept is given. For instance,

```
concept B<typename T> : A<T> {}

concept C<typename T>
{
    where A<T>;
}
```

the model declaration of B for some type *implies* that this type also models A, whereby a concept map of C *requires* that this type models A. The 42 concepts in this paper all use refinement to extend the set of requirements. In particular, Field refines most other concepts and a model declaration as Field for some type implies modeling 37 other concepts. Currently, no model for MultiplicativeGroup exists but it is perfectly reasonable to define a type that models Multiplicative-Group, see also Section 3.4.3.

3.6.1 Default Declarations for Standard Arithmetic Types

Furthermore, we group the arithmetic types with the same behavior using concepts. We used concepts for signed and unsigned integers from the concept-enabled standard library.

```
template <typename T>
  where std::SignedIntegral<T>
concept_map CommutativeRingWithIdentity<T> {}


template <typename T>
  where std::UnsignedIntegral<T>
concept_map AdditiveCommutativeMonoid<T> {}

template <typename T>
  where std::UnsignedIntegral<T>
concept_map MultiplicativeCommutativeMonoid<T> {}
```

Signed integers are Abelian groups with regard to addition and commutative monoid for multiplication. In addition, the two operations are distributive so that they model CommutativeRingWithIdentity with the limitation of overflow problems. All other model declaration are implied by this declaration.

Unsigned integers are also distributive but the demand for an additive Abelian group in all two-operation concepts presented avoids a model declaration for any of them. Therefore, two independent concept maps are given for commutative monoid regarding both operations.

A concept for floating point types is introduced in the math namespace[4]; as well as a concept for complex types. Both are used in model declarations of the here-presented concepts

```
template <typename T>
  where Float<T>
concept_map Field<T> {}

template <typename T>
  where Complex<T>
concept_map Field<T> {}
```

Users can enable model declarations for their types in two ways, either writing them directly or writing a concept map for some of the classifying concepts used above. Which is more appropriate depends on how similarly the type behaves to the modeling types. As these classifying concepts are still under

[4]Possibly these concept will be later integrated into the standard namespace.

development and it is not yet foreseeable in which templated algebraic and numerical model declarations they will be used, it is recommended to not write concept maps of user defined types for these concepts now.

3.6.2 Restricting Model Declarations for Operations

In our concepts, we provide a model declaration for maximal applicability. For instance, we define floating point types as AbelianGroup regarding math::add and as PartiallyInvertibleCommutativeMonoid regarding math::mult despite the imperfect associativity caused by rounding errors.

As criterion for model declaration, we examined whether it is useful for a significant part of the elements. Theoretically, (mult<**int**>, **int**) build a model of PartiallyInvertibleCommutativeMonoid whereby only 1 and -1 are invertible. Assuming most algorithms stop with an error by lack of invertibility, the benefit of such model declaration is negligible. In contrast, floating point numbers are declared as Field despite the fact that some denormalized non-zero values have no correct inverse.

One way to restrict the model declarations is to disable all pre-defined concept maps by compiling with −DLA_NO_CONCEPT_MAPS and declare models as desired.

For two reasons, this approach may be inappropriate. This technique fails in programs where the model declaration shall be only changed for one type or a few types and all others are utilized in the common way. However, this is only a convenience problem. More critical is the situation when a pair (Operation, Element) shall be for instance a model of Monoid in one part of the program and an AbelianGroup in another part (within the same compilation unit). Since model declarations are globally valid in the whole program, this is not possible with the same type tuple. However, a different modeling behavior can be achieved by introducing a new type for the operation.[5] As an example, declaring addition of **float** as a commutative, non-associative operation can be implemented with the following code fragment:

```
template <typename Element>
struct non_associative_add : public math::add<Element> {};

concept_map CommutativeMagma<non_associative_add<float>, float> {}
```

[5]Introducing a new element type is theoretically also possible but usually less elegant, especially when the element type is not a class.

Chapter 4

Examples

Algebraic concepts are crucial as fundamental theoretical background in numerical software. As an example, we present a generic algebraic computation in the form of the generalized power calculation with an arbitrary binary operation instead of multiplication. We will show different algorithms to compute the power function generically and we will show how algebraic concepts can be used to select the best applicable algorithms. In addition to dispatching between algorithms, the concept determines the range of correct exponents. Later we will demonstrate how algebraic concepts guarantee that loop unrolling is arithmetically correctly applied.

Other applications, not presented in this paper, are the construction of factorization trees by Henckell and Pin [7] using Monoids and the study of languages' complexity with Monoids by Raymond et al. [12].

Even more important than the direct application of the fundamental concepts to specify algorithm requirements is the definition of high-level concepts, like *vector* and *Hilbert spaces*. These high-level concepts are the background of many numeric libraries and it is crucial for the software reliability to integrate these mathematical requirements into the compilation process. We developed a first prototype of vector space concepts and we applied it in collaboration with Bonderer and Troyer to a linear solver—conjugate gradient—and an eigensolver—power method [2]. These results will be published in future papers; for this document we limit ourselves to the fundamental algebraic concepts.

4.1 Generic Power Function on Various Algebraic Concepts

The power function computes the repeated multiplication of a value a.

$$\begin{aligned} a^1 &= a \\ a^n &= a^{n-1}a \quad \text{if } n > 1 \end{aligned}$$

The extension to arbitrary *binary* operations is straightforward

$$\begin{aligned}\text{power}(a, 1, \text{op}) &= a\\ \text{power}(a, n, \text{op}) &= \text{op}(\text{power}(a, n-1, \text{op}), a) \quad \text{if } n > 1\end{aligned}$$

4.1.1 Generic Power Function on Magmas

The simplest algorithm to compute the n^{th} power of a with respect to a given binary operation op is to apply the operation successively $n - 1$ times. This calculation does not require special algebraic properties except that the result is representable as the argument type. As mentioned before, POD only fulfill this requirement if no overflow occurs. Assuming this, the algorithm's need can be specified by the concept math::Magma. The exponent is required to be an integral type and its value must be larger than 0.

```
template <typename Op, typename Element, typename Exponent>
   where math::Magma<Op, Element> && std::Integral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
# ifdef MTL_TRACE_POWER_DISPATCHING
       std::cout << "[Magma] ";
# endif

    if (exp < 1) throw "In power: exponent must be greater than 0";

    Element value= a;
    for (; n > 1; --n)
        value= op(value, a);
    return value;
}
```

For documentation purposes, one can enable a log message showing which version of power was actually called using -DMTL_TRACE_POWER_DISPATCHING as compiler flag or defining the macro before including power.hpp. We will omit the parts of the power that enable this tracing in the following listings for the sake of brevity.

4.1.2 Generic Power Function on Monoids

If the operation is associative, the following exponent law

$$a^n = a^{n-m}a^m \quad \text{for all } 1 \le m < n \tag{4.1}$$

applies. STEPANOV [19] pointed out that concept-based dispatching allows the use of (4.1) to compute the generic power function with logarithmic logarithmic effort over the exponent instead of linear. This algorithm is known as *multiply-and-square* or Russian peasant algorithm [19]. The fastest and clearest version of this algorithms demands an identity element for the operation (i.e. the operation must be a Monoid).

```
template <typename Op, typename Element, typename Exponent>
    where math::Monoid<Op, Element> && std::Integral<Exponent>
inline Element multiply_and_square(const Element& a, Exponent n, Op op)
{
    if (n < 0) throw "In multiply_and_square: negative exponent";

    using math::identity;
    Element value= identity(op, a), square= a;

    if (n & 1)
        value= base;

    for (n>>= 1; n > 0; n>>= 1) {
        square= op(square, square);
        if (n & 1)
            value= op(value, square);
    }
    return value;
}
```

The idea of the algorithm is to square a successively and represent the power as a product of these squared values (e.g., $a^{21} = a^{16} \cdot a^4 \cdot a^1$). Starting with the identity as temporary value, the value must be therefore multiplied with a if n is odd, with a^2 if $\lfloor n/2 \rfloor$ is odd, with a^4 if $\lfloor n/4 \rfloor$ is odd, ...

The binary representation of the exponent makes it very easy to decompose it into a sum of powers of two, for instance to decompose $21 = 10101_2 = 10000_2 + 100_2 + 1_2$. Relying on this presentation—therefore the exponent type is required to be integral—the oddness of the exponent can be computed by testing whether the least-significant bit is 1 and the down-rounded division by 2 can be calculated by shifting the value to the right.

Note that including an identity e in the computation provides the 0^{th} power. The definition of the 0^{th} power as identity follows naturally from the beforementioned exponent law (4.1)

$$\forall a, n \colon e \cdot a^n = a^n = a^{0+n} = a^0 \cdot a^n \quad \Longleftrightarrow \quad a^0 = e.$$

The extension to the generic power function is straight forward.

There is one slight inefficiency left in the algorithm. The identity is assigned to value and immediately overwritten if the n is odd. Using the ternary operator in the initialization avoids this.

```
template <typename Op, typename Element, typename Exponent>
    where math::Monoid<Op, Element> && std::Integral<Exponent>
inline Element multiply_and_square(const Element& a, Exponent n, Op op)
{
    if (n < 0) throw "In multiply_and_square: negative exponent";

    using math::identity;
    Element value= bool(n & 1) ? Element(a) : Element(identity(op, a)), square= a;
    for (n>>= 1; n > 0; n>>= 1) {
        square= op(square, square);
        if (n & 1)
            value= op(value, square);
    }
    return value;
}
```

This minor change reveals the very subtle effect of concept checking. Firstly, the result of n & 1 must be converted into a boolean. Although integer values are considered true in C++ iff they are different from 0, this is not part of the concept Integral (yet). Probably, this will change soon.

Another subtle difference with checking concepts is the usage of '**if**' and '?:'. When using the if-statement, both the type of a and the result type of identity must be assignable to the type of value. This is covered by the concept Monoid.[1] Using ?: requires that a and identity(op, a) have the same type or that one type is convertible into the other but not both mutually into each other. This is necessary to determine the type of the expression without ambiguity. The used concept does not guarantee the uniqueness of this result type; therefore we explicitly convert both values into Element, which is permissible with the concepts. The uniqueness of the ternary operator could also be guaranteed by a SameType constraint to the functions requirement. However, this would restrict the set of permissible types and undermines the goals of generic programming.

Without concept-checking, the type of the expression is determined at instantiation time. Then the type of Element and the result type of identity are known and compiler error only happens when the result type of ?: is ambiguous for the actually instantiated types. As a consequence, type conflicts in generic functions can stay undiscovered for a long time if the function were never instantiated with type combinations causing the ambiguity. For instance, some subtle ambiguities using the ternary operator in the STL were only discovered after concept-checking the library.

In order to enable better code re-usability, we provide a separate function multiply_and_square that is used in different manners by several versions of power. The power function of Monoid is then trivial.

```
template <typename Op, typename Element, typename Exponent>
    where math::Monoid<Op, Element> && std::Integral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
    return multiply_and_square(a, n, op);
}
```

4.1.3 Generic Power Function on Semi-Groups

To lift the algorithm to SemiGroup the implementation must compute without using identity, which also excludes the 0^{th} power. One solution is to recursively

[1]in math:: not in algebra::

compute $a^{\lfloor n/2 \rfloor}$, square this result and multiply it with a if n is odd,

$$a^n = (a^{\lfloor n/2 \rfloor})^2 \cdot a^{n \bmod 2}.$$

The implementation of this recursive technique reads:

```
template <typename Op, typename Element, typename Exponent>
    where math::SemiGroup<Op, Element> && std::Integral<Exponent>
Element power(const Element& a, Exponent n, Op op)
{
    Exponent half= n >> 1;

    // If half is 0 then n must be 1 and the result is a
    if (half == 0)
        return a;
    Element value= power(a, half, op);
    value= op(value, value);
    if (n & 1)
        value= op(value, a);
    return value;
}
```

Recursive function calls are more expensive than iterative methods and unlikely to be inlined. To compute the generic power function iteratively in absence of identity, the computation must start with the most-significant bit. Intermediate values are continuously squared and, depending on whether the corresponding bit in the exponent is odd multiplied with a. The computation is very similar to HORNER's scheme [8] for evaluating polynomials

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = ((\cdots(a_n x + a_{n-1})x \cdots)x + a_1)x + a_0$$

In the same fashion, the power computation can be nested with the most-significant bit of the exponent inside all nested operations. Be $n = b_k 2^k + b_{k-1} 2^{k-1} + \cdot + b_1 2 + b_0 1$ the binary representation of n with $b_k = 1$. Then the n^{th} power of a can be written as

$$a^n = ((\cdots((((a^{b_k})^2 a^{b_{k-1}})^2 a^{b_{k-2}})^2 a_{n-2}) a \cdots)^2 a^{b_1})^2 a^{b_0}.$$

For instance, a^{20}, with $20 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$, reads

$$a^{20} = a^{10100_2} = ((((a^1)^2 a^0)^2 a^1)^2 a^0)^2 a^0.$$

The multiplication with a^0 is not realized as a multiplication with the identity, which is not available, but is only a notation that the actual value is not multiplied with a^0. The implementation of the algorithm starts with finding the highest 1-bit in the exponent. Special attention must be payed to signed **int** where the search of the highest one-bit of n must start in the second-most significant bit (otherwise negative values are introduced and the shifting usually behaves differently).

```
template <typename Op, typename Element, typename Exponent>
    where math::SemiGroup<Op, Element> && std::Integral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
    if (n <= 0) throw "In power [SemiGroup]: exponent must be greater than 0";

    Exponent mask= 1 << (8 * sizeof(mask) - 1);
    if (mask < 0)
        mask= 1 << (8 * sizeof(mask) - 2);
    while(!(bool)(mask & n)) mask>>= 1;

    Element value= a;
    for (mask>>= 1; mask; mask>>= 1) {
        value= op(value, value);
        if (n & mask)
            value= op(value, a);
    }
    return value;
}
```

Although this algorithm starts with the most-significant position in the exponent, its main loop has the same compute pattern of squaring and multiplying. The only difference in compute time is therefore the initial search for the highest bit in n.

Remark One can argue whether a separate implementation for semi-groups is needed. Each semi-group can be embedded into a monoid by adding a new element, say e, and defining $ex = x = xe$ for all members of the semi-group. Although this is mathematically perfectly feasible, we cannot easily add a new value to a type so that this embedding has no practical impact in computer science. On the other hand, all examples of semi-groups, we found so far provide an identity element. However, one can easily define non-monoidal semi-groups by appropriately limiting the element range, e.g. addition of **float** with $\forall x\colon x < -4$, multiplication of **int** with $\forall x\colon x > 47$, or concatenation of strings with at least three characters length. In all these cases the identity is a value of the type but excluded from the set by definition. As a consequence, one could use this identity, which comes from outside the set, internally in the algorithms but must require that the exponent is at least one so that the result always lies in the set.

4.1.4 Generic Power Function on Groups

Computing the power on a Group operation does not enable a more efficient algorithm. The difference is that the existence of an inverse function enables negative exponents by means of another *exponent law*

$$a^{-n} = \frac{1}{a^n}$$

or more generally

$$\text{power}(a, -n, \text{op}) = \text{power}(\text{inverse}(a), n, \text{op}). \tag{4.2}$$

The implementation of (4.2) is simple use of multiply-and-square

```
template <typename Op, typename Element, typename Exponent>
    where math::Group<Op, Element> && std::SignedIntegral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
    using math::inverse;
    return n >= 0 ? multiply_and_square(a, n, op)
                  : multiply_and_square(inverse(op, a), −n, op);
}
```

Please note that the implementation

```
template <typename Op, typename Element, typename Exponent>
    where math::Group<Op, Element> && std::SignedIntegral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
    using math::inverse;
    return n >= 0 ? power(a, n, op)
                  : power(inverse(op, a), −n, op);
}
```

would only call itself and end in an infinite loop. If it there were a possibility to call the power function of Monoid within the power function of Group, an implementation similar to the last listing were feasible. In order to use the multiply-and-square algorithm differently within the different power selections, it is implemented in a separate function.

4.1.5 Generic Power Function on Partially Invertible Monoids

For many types, the division is not defined for all values. More generically, there are operations with an inverse function but which is not guaranteed to work on all values—e.g., wrong results, exceptions, program crashes, infinite loops. The generic approach to handle this is the boolean function is_invertible that can be used to avoid illegal calls of inverse. For more details see PartiallyInvertibleMonoid.

```
template <typename Op, typename Element, typename Exponent>
    where math::PartiallyInvertibleMonoid<Op, Element>
          && std::SignedIntegral<Exponent>
inline Element power(const Element& a, Exponent n, Op op)
{
    using math::inverse; using math::is_invertible;

    if (n < 0 && !is_invertible(op, a))
        throw "In power: a must be invertible with negative exponent";
    return n >= 0 ? multiply_and_square(a, n, op)
                  : multiply_and_square(inverse(op, a), -n, op);
}
```

4.1.6 Test Case: Power Function on Non-Negative Reals

To test some types other then plain old data types (POD), we introduced a new class positive_real that explicitly excludes negative reals at run-time. The positive reals including the zero $\mathbb{R}_0^+$ form a CommutativeMonoid with respect to addition and a PartiallyInvertibleCommutativeMonoid w.r.t. multiplication. Excluding the zero $\mathbb{R}^+$ would build an AbelianGroup towards multiplication. We refrain from the last definition because slight asymmetries of floating point numbers small de-normalized numbers might also turn into infinity by inversion.

To enable different concept maps for the same binary operation, we introduce five new function classes, which are all publically derived from math::mult.

```
struct magma_mult : public math::mult<positive_real> {};
struct semigroup_mult : public math::mult<positive_real> {};
struct monoid_mult : public math::mult<positive_real> {};
struct pim_mult : public math::mult<positive_real> {}; // Partially invertible monoid
struct group_mult : public math::mult<positive_real> {};

namespace math {
    concept_map SemiGroup< semigroup_mult, positive_real > {};
    concept_map Monoid< monoid_mult, positive_real > {};
    concept_map PartiallyInvertibleMonoid< pim_mult, positive_real > {};
    concept_map Group< group_mult, positive_real > {};
}
```

The concept Magma is 'auto' and does not need a model declaration.

Alternatively, one could test the concept dispatching with the same operation functor and different element types. This demands typically more work when the base class has non-trivial constructors that has to be called from each derived class. In addition, functions like identity would need to be rewritten due to the argument change. Theoretically there is no objection either to vary both Element and Operation type (which causes even more implementation work). Resuming, from our experience it is the easiest way to provide multiple Operation functor types because they are normally state-free types. However, this are all practical technical considerations and do not demonstrate any limitation of concepts.

Computing the 777^{th} and -777^{th} power of 1.1 and 0 on with respect to different concepts returns the following results.

```
1.1^777 as Magma  [Magma] 1.4525e+32

1.1^777 as SemiGroup  [SemiGroup] 1.4525e+32

1.1^777 as Monoid  [Monoid] 1.4525e+32
1.1^-777 as Monoid  [Monoid]
== Exception: In multiply_and_square: negative exponent
```

```
1.1^777 as PIMonoid  [PIMonoid] 1.4525e+32
1.1^-777 as PIMonoid  [PIMonoid] 6.88468e-33
0^777 as PIMonoid  [PIMonoid] 0
0^-777 as PIMonoid  [PIMonoid]
== Exception: In power [PIMonoid]: a must be invertible with negative n

1.1^777 as Group  [Group] 1.4525e+32
1.1^-777 as Group  [Group] 6.88468e-33
0^777 as Group  [Group] 0
0^-777 as Group  [Group] inf
```

The modeled concept is printed for each operation, e.g., `1.1^777 as SemiGroup` means that 1.1^{777} is computed with a functor that is declared to model SemiGroup. To verify whether the correct version of power was called, the functions were enabled[2] to print out the concept dispatching in brackets.

As one can see, the result of 1.1^{777} is always the same. The efficiency is slightly lower in a SemiGroup and significantly lower in a Magma. Negative exponents cannot be used for Monoid and more general concepts. Raising zero to a negative power is an illegal operation and modeling the multiplication as PartiallyInvertibleMonoid allows us to provide a proper error behavior. Conversely, zero is incorrectly inverted to ∞ when the multiplication is modeled as Group. Note that the inversion of 0 and ∞ is to some extend consistently defined in IEEE 754 as $1/0 = \infty$ and $1/\infty = 0$. Nevertheless, it does not model Group since $0 \cdot \infty$ is not one but NaN.

4.2 On the Granularity of Template Function Constraints

In Section 4.1, we used the algebraic concepts from namespace math that impose constraints on the Element type in form of assignability and convertibility. In contrast to it, the concepts in namespace algebra specify exclusively mathematical properties. Therefore, if used to constrain a template function, all needs of this function—and of all functions called by it—towards assignability and convertibility must be defined in the **where** clause. For instance the power function for monoid and group from Section 4.1 would have the following constraint lists with the concepts from algebra

```
template <typename Op, typename Element, typename Exponent>
    where algebra::Monoid<Op, Element> && std::Integral<Exponent>
        && std::Callable2<Op, Element, Element>
        && std::Assignable<Element, std::Callable2<Op, Element,
                                                  Element>::result_type>
        && std::Assignable<Element, Element>
inline Element power(const Element& base, Exponent n, Op op)
{
    return multiply_and_square(base, n, op);
```

[2]Compiled with `-D MTL_TRACE_POWER_DISPATCHING`.

```
}

template <typename Op, typename Element, typename Exponent>
    where algebra::Group<Op, Element> && std::SignedIntegral<Exponent>
            && std::Callable2<Op, Element, Element>
            && std::Assignable<Element, std::Callable2<Op, Element,
                                                        Element>::result_type>
            && std::Assignable<Element, Element>
inline Element power(const Element& base, Exponent n, Op op)
{
    using math::inverse;
    return n >= 0 ? multiply_and_square(base, n, op)
                  : multiply_and_square(inverse(op, base), −n, op);
}
```

The advantage of explicitly enumerating the requirements is that they are absolutely minimal and the set of supported types is maximal. However, this explicitness comes at a high price.

One obvious disadvantage is that the **where** clause became significantly larger. The developer of concept-checked generic libraries will spent much time specifying constraints when using minimalist concepts. While libraries grow by defining more expressive and more powerful functions on top of existing ones, the situation even deteriorates because the requirements of all called functions and all used data types are agglomerated in the new function or in the new data type. In worst case, the lists of constraints grow exponentially with the depth of called functions and used data structures.

It is therefore inevitable that libraries with more expressive functionality also need to provide more complex concepts. For not being over-restrictive, the concepts must be kept as lean as possible while containing the essential requirements.

Another problem with minimalist function constraints is the sensitivity to changes of the called function. Adding, for instance, the statement `value= identity(op, base);` to `multiply_and_square` would demand the requirement

```
std::Assignable<Element, algebra::Monoid<Op, Element>::identity_result_type>
```

added to constraint list of `multiply_and_square`'s, and to the lists of the two `power` functions above. Especially, if a function is called by many others directly or indirectly, adding a requirement adds this requirement to *all functions that call it directly or indirectly* unless the calling function has the requirement already in the **where** clause. The impact is even much stronger than changing the interface because then only direct callers are concerned. To avoid this avalanche-like propagation of additional requirements, both function can be kept: (i) the original with less requirements and (ii) the more restrictive and more efficient version—we assume that the latter is more efficient in some sense otherwise one would not add a more restrictive function.

Using very fine-grained concepts like the ones above can be a source of highly increasing software development time. Many concepts introduce associated types that are only used as result type of a function and defined by automatic type detection. Often cases, their names do not need to be known in function constraints. In Section 4.1, the associated types of algebraic concepts were not exposed in the **where** clauses. This is different with the constraints in this section, the name of op's and identity's result types needs to be known to the author of the power functions. One might be lucky and find these names in the error messages of the compiler instead of reading all details of the documentation or the source code of the concept. However, this technique only works when functions fail to instantiate and it is by far more difficult to debug code that compiles but dispatches to non-optimal functions.

The reasonings above showed that the right granularity of concept design is uttermost importance. Too coarse-grained concepts eliminate too many types and undermine the whole idea of generic programming. On the other hand, expressing the constraints of complex functions with a long list of simple requirements will make the development of generic software too expensive. Enabling at the same time maximal applicability and efficient development can be realized with three methods. First, broadly used essential functions, like STL function, shall be implemented with maximal applicability regardless how many requirements need to be specified. More complex functionality shall involve more elaborated concepts; defining the requirements of a finite element assembly or a non-linear solver with dozens or hundreds involved std::Assignable and std::Convertible requirements would be fatally distractive. Second, concepts can be used more appropriately if they are defined hierarchically like the algebraic concepts in algebra and math.

Third, developing concepts according to functions and data types that are most likely used with this concepts so that more requirements can be included in the concepts without impact to the main applications. For instance, the difference between the easy-to-read constraints in Section 4.1 using more refined concepts and the long requirement lists in this section using less restrictive concepts is that the assignability of identity's result type was requested in the first case. However, this did not exclude any POD or user-defined type from the set of models. Even, if the first version is admittedly more restrictive, the restriction had no practical impact so far.

4.3 Generic Reduction Implementation

Another important generic function is the reduction of a set of n values to a single value using a binary operation. In the STL this function is called accumulate using a loop over each value. The function can be implemented with concepts in the following way:

```
template <typename Iter, typename Value, typename Op>
  where std::ForwardIterator<Iter>
    && std::Convertible<Value, std::ForwardIterator<Iter>::value_type>
    && math::Magma<Op, std::ForwardIterator<Iter>::value_type>
typename std::ForwardIterator<Iter>::value_type
inline accumulate(Iter first, Iter last, Value init, Op op)
{
    for (; first != last; ++first)
```

```
        init= op(init, *first);
    return init;
}
```

We will later refer to this implementation as sequential reduction.

4.3.1 Arithmetically Correct Loop Unrolling with Concept Verification

In scientific computing loops are usually unrolled in order to accelerate the execution on super-scalar processors. Unfortunately, unrolling is often not sufficient if all statements access the same variable for accumulation. To provide independent operations it is necessary to introduce multiple temporaries. This in turn changes the order of operations and the result is only correct if the operation is associative and commutative (ignoring rounding errors in these examples). In addition, the identity of the operation is needed when more than one temporary is used. All these demands are specified by the concept CommutativeMonoid. In comparison, the STL function without unrolling only has the requirements of Magma. In addition, the requirement of ForwardIterator needs to be narrowed to RandomAccessIterator.

```
template <typename Iter, typename Value, typename Op>
  where std::RandomAccessIterator<Iter>
    && std::Convertible<Value, std::RandomAccessIterator<Iter>::value_type>
    && math::CommutativeMonoid<Op,
                               std::RandomAccessIterator<Iter>::value_type>
typename std::RandomAccessIterator<Iter>::value_type
inline accumulate(Iter first, Iter last, Value init, Op op)
{
    typedef std::RandomAccessIterator<Iter> trait;
    typedef typename trait::value_type value_type;
    typedef typename trait::difference_type difference_type;
    value_type t0= identity(op, init), t1= identity(op, init),
                     t2= identity(op, init), t3= init;
    difference_type size= last − first, bsize= size >> 2 << 2, i;

    for (i= 0; i < bsize; i+= 4) {
        t0= op(t0, first[i]);
        t1= op(t1, first[i+1]);
        t2= op(t2, first[i+2]);
        t3= op(t3, first[i+3]);
    }
    for (; i < size; i++)
        t0= op(t0, first[i]);
    return op(op(t0, t1), op(t2, t3));
}
```

In the context of this work, the change of the iterator concept is less important, also compilers without concepts will inform the programmer if the iterator has no random access (not necessarily in a readable form). What is completely unknown to the compiler is commutativity and associativity of the binary operation. This can only be provided by a concept map, either from the user or from a library, and only be verified by a concept-enabled compiler.

4.3.2 Handling Inaccurate Arithmetic

Floating point numbers are associative if one takes rounding errors into consideration. In most applications, rounding errors are tolerated unless the computation is numerically sensitive, like Gram-Schmidt orthogonalization. For this reason, we normally treat floating point numbers as associative data, both toward addition and multiplication. Handling **float** and **double** as non-associative disables the unrolling of reduction operations. However, that does not improve the computation at all.

The unrolled reduction is *Not Wronger* than the sequential reduction, it is only *Differently Wrong*. The inaccuracy originates from cancellations of low-significant bits and the problem already occurs in the sequential reduction. The unrolling of the reduction computation does not introduce a new source of error, it only shifts it slightly. As a matter of fact, the non-associativity itself is a consequence of the inaccuracy.

As mentioned above, it does not help to disable the unrolling in order to handle rounding errors. We rather need to disable both generic versions and need to call a numerically stabler reduction. Which algorithm is best in which context is beyond the scope of this document, we merely provide the framework to detect when the generic reduction can be used. Nevertheless, we like to mention that the selection between different stable algorithms can also be realized with the aid of concepts.

As a tool, we introduce at this point the concept RegularReduction and provide conditions for this regularity. The template function accumulate now dispatches between three different versions, which call different implementations.

```
concept RegularReduction<typename Operation, typename Element> {}

template <typename Iter, typename Value, typename Op>
   where std::ForwardIterator<Iter>
         && std::Convertible<Value, std::ForwardIterator<Iter>::value_type>
         && math::Magma<Op, std::ForwardIterator<Iter>::value_type>
         && RegularReduction<Op, std::ForwardIterator<Iter>::value_type>
typename std::ForwardIterator<Iter>::value_type
inline my_accumulate(Iter first, Iter last, Value init, Op op)
{
    return mtl::accumulate_simple(first, last, init, op);
}

template <typename Iter, typename Value, typename Op>
   where std::RandomAccessIterator<Iter>
         && std::Convertible<Value, std::RandomAccessIterator<Iter>::value_type>
         && math::CommutativeMonoid<Op,
                        std::RandomAccessIterator<Iter>::value_type>
```

```
            && RegularReduction<Op, std::RandomAccessIterator<Iter>::value_type>
typename std::RandomAccessIterator<Iter>::value_type
inline my_accumulate(Iter first, Iter last, Value init, Op op)
{
    return mtl::accumulate_unrolled(first, last, init, op);
}

template <typename Iter, typename Value, typename Op>
  where std::ForwardIterator<Iter>
        && std::Convertible<Value, std::ForwardIterator<Iter>::value_type>
        && math::Magma<Op, std::ForwardIterator<Iter>::value_type>
typename std::ForwardIterator<Iter>::value_type
inline my_accumulate(Iter first, Iter last, Value init, Op op)
{
    return mtl::accumulate_staple(first, last, init, op);
}
```

The question at hand is: What are sufficient conditions that the generic reduction can be used regularly? Firstly, data types that are not subject to rounding errors, like **int**, can always use the generic version. The notion of precise arithmetic for *every operation* is worthwhile to be formalized in a concept. All integral types model this concept. It is also clear that this is a sufficient condition to apply the generic reduction on every binary operation.

```
concept AccurateArithmetic<typename T> {}

template <typename T>
    where std::Integral<T>
concept_map AccurateArithmetic<T> {}
```

The generic reduction is also applicable if rounding errors for a certain type are tolerated, which is specified by the following concept

```
concept TolerateRoundingErrors<typename T> {}
```

It is the common practice to accept rounding errors for all standard floating point types so that this should be the default behavior. On the other hand, we need the opportunity to disable this default. A possible implementation is the following,

```
# ifndef CONSIDER_FLOAT_ROUNDING_ERRORS
    template <typename T>
        where math::Float<T>
    concept_map TolerateRoundingErrors<T> {}
# endif
```

where for all Float types rounding errors are supposed to be tolerated unless the macro CONSIDER_FLOAT_ROUNDING_ERRORS is defined in the source code or command line. This approach allows a finer grained type treatment. It is possible to disable the tolerance in general but enable it for some type, e.g., types with higher precision like quad-double qdouble.

```
concept_map TolerateRoundingErrors<qdouble> {}
```

Even for types where rounding errors are an issue, some operations can be non-critical, like minimum and maximum. The reason why these operations are no subject to rounding errors is that no arithmetic operations are executed but only *selective* operation. We characterize this type of operations in the concept SelectiveOperation.

```
concept SelectiveOperation<typename Operation, typename Element> {}

template <typename Element>
concept_map SelectiveOperation<math::min<Element>, Element> {}

template <typename Element>
concept_map SelectiveOperation<math::max<Element>, Element> {}
```

We declare the functor types math::min and math::max as models of this concept for all types. The practice will show whether it is necessary to declare the model only for known types or if it is safe to declare it for all types with these operations. Note that math::min and math::max are defined as functors, opposed to std::min and std::max, which are functions.

Modeling one of the concepts above is a sufficient condition that the generic reduction can be used. Since it is not clear at this moment whether logical disjunction will be supported, we implement the model declaration with three concept maps.

```
template <typename Operation, typename Element>
  where AccurateArithmetic<Element>
concept_map RegularReduction<Operation, Element> {}

template <typename Operation, typename Element>
  where TolerateRoundingErrors<Element>
        && !AccurateArithmetic<Element>
concept_map RegularReduction<Operation, Element> {}

template <typename Operation, typename Element>
  where SelectiveOperation<Operation, Element>
        && !AccurateArithmetic<Element>
        && !TolerateRoundingErrors<Element>
concept_map RegularReduction<Operation, Element> {}
```

For internal implementation reasons of concept-enabled compilers, the constraints in the **concept_map** must exclude each other. Therefor, we add extra negative constraints to the model declarations. It is possible that more concepts can help to characterize the regularity of reduction.

Although we investigated the treatment of rounding errors in the context of a reduction operation, we expect that similar approaches can be taken in other contexts and that this discussion and the introduced concepts can contribute to address rounding errors in whole area of scientific computing as part of scientific libraries (instead of being optimization details of compilers).

Chapter 5

Value-Based Model Declarations

This chapter discusses concept modeling that depends on values. An important example of value-based model distinction are cyclic groups that are fields if the cycle length is prime and rings otherwise. In order to declare whether a cyclic group is a field or a ring for a given cycle length, the concept checking has to determine whether this length is prime.

Another aspect that makes cyclic sets so interesting is that all hardware-supported arithmetic types have flaws in modeling algebraic concepts properly The reason is that infinite mathematical structures are approximated with finite data types. On the other hand, finite algebraic structures can be perfectly modeled.

5.1 Modular Arithmetic

Cyclic groups can be implemented with modular arithmetic on integral data types. Using machine arithmetic for modular arithmetic, the size of the set and all intermediate results must not be larger than the maximum value of the underlying data type. In fact, this means that the cycle size must be equal to or smaller than the square root of the largest representable value of the underlying arithmetic type. Furthermore, the division has to implemented with an extended Euclidean algorithm instead of the native division. A cyclic group with the underlying arithmetic type and the modulus as template parameter is sketched here

```
template<typename T, T N>
class mod_n_t
{
    T value;
  public:
    static T const modulo= N;
    typedef mod_n_t self;

    explicit mod_n_t(T const& v) : value( v % modulo ) {}

    self operator+ (self const& x, self const& y);
    self operator− (self const& x, self const& y);
    self operator∗ (self const& x, self const& y);
    self operator/ (self const& x, self const& y);
    /∗ ... ∗/
};
```

The name ‘cyclic group’ is somehow confusing since the cyclic sets are only groups with respect to addition but not with regard to multiplication.

5.2 Algebraic Structure of Cyclic Groups

The type models the concept CommutativeRingWithIdentity for all possible instantiation of T and N.

```
template <typename T, T N>
concept_map CommutativeRingWithIdentity< mod_n_t<T, N> > {}
```

The really interesting fact is that in case N is a prime number the corresponding type also models Field, which in turn can be tested with the concept Prime.

```
template <typename T, T N>
    where meta_math::Prime<N>
concept_map Field< mod_n_t<T, N> > {}
```

The concept Prime relies on a meta-function that calculates at compile-time whether a number is prime.

```
concept Prime<long int N> {}

template <long int N>
    where std::True<is_prime<N>::value>
concept_map Prime<N> {}
```

More precisely the meta-function is used to express a conditional model declarations. To not digress too far, we refrain from giving the source code of is_prime and only mention that it tests whether N is divisible by 2 or any odd number smaller than the square root of N. For small N the compile time is not affected, for values around 1,000,000 the compiler needed about 18 seconds on a PowerPC G4 1.3GHz, and about one minute to test prime number around 10,000,000. In the latter case, intermediate types reached slightly more than 1,500 levels of template nesting.

5.3 Concept-Based Invertibility Test

The multiplicative structure of cyclic sets is never a group because at least the 0-element — or more precisely the equivalence class with additive identity element — is never invertible. The multiplicative structure of a cyclic set always models MultiplicativePartiallyInvertibleMonoid regardless the cycle length.

The computation of the invertibility can be optimized regarding the cycle length. In the general case, the invertibility of a certain element has to be computed with EUCLIDEAN's algorithm to check whether the element in question is co-prime with the cycle length.

```
template<typename T, T N>
struct is_invertible_t< mult< mod_n_t<T, N> >, mod_n_t<T, N> >
{
    typedef mod_n_t<T, N> mod_t;
    bool operator() (mult<mod_t> const&, mod_t const& v) const
    {
        return v.get() != 0 && mtl::gcd(N, v.get()) == 1;
    }
};
```

This computation is relatively expensive. For cyclic groups where the cycle length is prime — i.e. N models Prime — the invertibility test can be implemented more efficiently. In this case, every element is invertible if it is different from 0.

```
template<typename T, T N>
    where meta_math::Prime<N>
struct is_invertible_t< mult< mod_n_t<T, N> >, mod_n_t<T, N> >
{
    typedef mod_n_t<T, N> mod_t;
    bool operator() (mult<mod_t> const&, mod_t const& v) const
    {
        return v.get() != 0;
    }
};
```

This is only one example where concept modeling can be used to accelerate computation.

Chapter 6

Theory of Conceptual Restriction

In this chapter we discuss the questions:

- While using the requirements of concept A in concept B, when is a refinement preferable to a **where** clause?
- What types of requirements can be expressed by refinements?
- Which requirements should not be specified by refinements, even if this is technically feasible?
- When should a concept definition be **auto**?
- Which models are implicitly declared for different styles of concept definitions?

Some of these questions cannot be answered in a formal way. Instead, they are design decisions and we will provide guidelines for them. Before we discuss the answers to the questions raised above, we introduce definitions needed to express the theory of conceptual restriction.

6.1 Definitions

Concept refinements and **where** clauses are equal in the sense that refining concept B from concept A or requesting concept A in a **where** clause of concept B both signify that a type or a tuple of types must model A in order to model B. This commonality is represented in the following:

Definition 2 *For a concept B requesting concept A in a* **where** *clause we say B* requests *A and depict it by* $\mathsf{B} \dashv \mathsf{A}$. *The refinement of concept A is symbolized by* $\mathsf{B} : \mathsf{A}$ *as in programs. Concept B* restricts *concept A if it refines A or requests A. Restriction is represented by* $\mathsf{B} \prec \mathsf{A}$

$$\mathsf{B} \prec \mathsf{A} \overset{\text{def}}{=} \mathsf{B} \dashv \mathsf{A} \ \vee \ \mathsf{B} : \mathsf{A} \tag{6.1}$$

To simplify notation we often write a tuple of types $(t_1, t_2, \ldots, t_n)$ as one symbol $t = (t_1, t_2, \ldots, t_n)$. The set of all definable types in C++ is denotes as $\mathbb{T}$. Equivalently, the set of all tuples of types with arity n is given by the n^{th} direct product of $\mathbb{T}$

$$\mathbb{T}^n = \underbrace{\mathbb{T} \times \cdots \times \mathbb{T}}_{n \text{ times}} = \{(t_1, \ldots, t_n) \colon t_1 \in \mathbb{T} \wedge \cdots \wedge t_n \in \mathbb{T}\} \tag{6.2}$$

Definition 3 *The set of tuples that model concept* C *is for brevity called* model set of concept C *and is denoted by* $\mathbb{T}_\mathsf{C}$*. The arity of the concept* n_C *is the defined as the arity of these tuples, which is identical with the concept's number of template parameters.*

The tuples modeling a certain concept are a subset of all tuples with the concepts' arity

$$\mathbb{T}_\mathsf{C} \subseteq \mathbb{T}^{n_\mathsf{C}}.$$

Note that this is not a strict subset relationship. However, concepts with the equality $\mathbb{T}_\mathsf{C} = \mathbb{T}^{n_\mathsf{C}}$ have no practical value and can be omitted.

Definition 4 *A type or a tuple of types* t *modeling a concept* C *is notated by* $t \models \mathsf{C}$*. Furthermore, model declaration explicitly written by the programmer are symbolized* $t \models_p \mathsf{C}$ *and implicit declarations inserted by the compiler* $t \models_i \mathsf{C}$*.*

The fact that a certain type tuple models a concept can be written different ways

$$t \models \mathsf{C} \equiv t \in \mathbb{T}_\mathsf{C}.$$

For brevity, we also introduce a notation for modeling multiple concepts

$$t \models \mathsf{C}_1, \ldots, \mathsf{C}_n \overset{\text{def}}{=} t \models \mathsf{C}_1 \wedge \cdots \wedge t \models \mathsf{C}_n.$$

6.2 Model Set Inclusion of Concepts with Different Arities

The set of types that model a concept—short model set—is a set of type tuples where the arity of the tuples is the number of type parameters in the concept (the concept's arity). If concept C is refined from concept A and the two concepts have different arities then the two model sets consists of tuples of different arities. Nevertheless, we will show in this section that the model set of C is included in the model set of A by adapting the representation of A's model set. This model set inclusion is a crucial property of concepts and it is therefor important to prove its validity for concept refinement with arity change.

Although the model set inclusion is not in general expected regarding a concept B requested in a **where** clause of concept C, we can show that the model set of C is included in the model set of B in an appropriate representation. At first, we consider refinements of multi-type concepts on examples.

Refining a concept adds requirements and as a consequence the set of modeling types is potentially reduced

$$\mathsf{C} : \mathsf{A} \quad \Longrightarrow \quad \mathbb{T}_{\mathsf{C}} \subseteq \mathbb{T}_{\mathsf{A}}.$$

The model set of a refinement from multiple concepts is a subset of the intersection of the concepts refined from

$$\mathsf{C} : \mathsf{A}_1, \ldots, \mathsf{A}_n \quad \Longrightarrow \quad \mathbb{T}_{\mathsf{C}} \subseteq \bigcap_{i\in[1,n]} \mathbb{T}_{\mathsf{A}_i} \tag{6.3}$$

This formulae cannot be applied directly when the refined concepts have different arity. However, we will show that a very similar formulae, (6.9), applies in general.

6.2.1 Refining from Concepts with Lower Arity

First, we consider the case that the refined concept(s) has/have lower arity. How does the type set inclusion work if the concepts have different arities? For instance, the concept GenericRing in Section 9.2.1 is defined on three types and refines two concepts on two types each.

```
concept GenericRing<typename AddOp, typename MultOp, typename Element>
  : AbelianGroup<AddOp, Element>,
    SemiGroup<MultOp, Element>
{ /* ... */ };
```

To introduce the principle of model set extension, we start by considering the example above. Let $t = (t_1, t_2, t_3)$ model GenericRing. This requires that (t_1, t_3) models AbelianGroup and that (t_2, t_3) models SemiGroup. Be $\mathbb{T}_{\mathsf{A}_1}$, $\mathbb{T}_{\mathsf{A}_2}$, and $\mathbb{T}_{\mathsf{C}}$ the model sets of AbelianGroup, SemiGroup, and GenericRing respectively, then we define the extension of $\mathbb{T}_{\mathsf{A}_1}$ and $\mathbb{T}_{\mathsf{A}_2}$ with respect to $\mathbb{T}_{\mathsf{C}}$ as

$$\hat{\mathbb{T}}_{\mathsf{A}_1}(\mathsf{C}) \stackrel{\text{def}}{=} \{(t_1, t_2, t_3) \colon (t_1, t_3) \in \mathbb{T}_{\mathsf{A}_1} \wedge t_2 \in \mathbb{T}\}$$
$$\hat{\mathbb{T}}_{\mathsf{A}_2}(\mathsf{C}) \stackrel{\text{def}}{=} \{(t_1, t_2, t_3) \colon (t_2, t_3) \in \mathbb{T}_{\mathsf{A}_2} \wedge t_1 \in \mathbb{T}\}.$$

To determine which type parameters are added to the tuple, the type extension must refer to a reference concept (here $\mathbb{T}_{\mathsf{C}}$). However, if the reference is clear from the context we will omit it. In other words, tuples with lower arities are extended with free parameters. Thus, the extended model sets are tuples of the same arity as concept C and the added parameters can be of any type. For this extended type sets, the inclusion (6.3) of the intersection also holds

$$\mathbb{T}_{\mathsf{C}} \subseteq \hat{\mathbb{T}}_{\mathsf{A}_1} \cap \hat{\mathbb{T}}_{\mathsf{A}_2}. \tag{6.4}$$

Please note that although in this example the arities of the refined concepts are lower than the arity of the defined concept, all type parameters of the defined concept are used at least once in the refined concepts. This is neither a necessary requirement for mathematical correctness nor is it needed in order to fulfill the inclusion requirement. The type set extension is also correct if not all template parameters are referred to in the refined concepts. Nevertheless, for the sake of maintainability it is advisable to not over-stress refinement from concepts with lower arity; read also the discussion in Sections 6.3.2 and 6.3.3.

6.2.2 Refining from Concepts with Higher Arity

Conversely, the refinement from the functor-based to the operator-based single-operation concepts reduces the concept arity. For instance, the refinement of AdditiveMagma from Magma

```
concept AdditiveMagma<typename Element>
  : Magma< math::add<Element>, Element >
{ /* ... */ };
```

omits the first type parameter. This is only possible because the additional type is completely dependent on the parameters of the refined concept.

The refinement from a concept with more *free* parameters violates the consistent inclusion of model sets. All parameters must be either *constant* or *uniquely determined* by the type parameters of the concept. Fortunately, the language specification in C++ forces this by declaring the free type parameters as parameters of the concept. For instance, a syntax like

```
template <typename T, typename U>
concept C<T> : A<T, U> {}
```

would tolerate more free parameters in the refined concepts than in the refining one.

Be $\mathbb{T}_\mathsf{A}$ and $\mathbb{T}_\mathsf{C}$ the model sets of Magma and AdditiveMagma respectively, then $\mathbb{T}_\mathsf{C}$ is extended with regard to $\mathbb{T}_\mathsf{A}$ by adding the dependent type math::add.

$$\hat{\mathbb{T}}_\mathsf{C}(A) \stackrel{\text{def}}{=} \{(\mathsf{math::add}\langle t\rangle, t)\colon t \in \mathbb{T}_\mathsf{C}\}$$

Now, the inclusion relation (6.3) is fulfilled

$$\hat{\mathbb{T}}_\mathsf{C} \subseteq \mathbb{T}_\mathsf{A}.$$

Taking the type parameter dependency in the concept definition into account, the models of Magma that are relevant for the considered refinement

$$\mathbb{T}'_\mathsf{A} = \{(t_1, t_2)\colon t_1 = \mathsf{math::add}\langle t_2\rangle\} \cap \mathbb{T}_\mathsf{A} \subseteq \mathbb{T}_\mathsf{A}.$$

Let $\hat{\mathbb{T}}_\mathsf{A}$ be a set of single elements with

$$\hat{\mathbb{T}}_\mathsf{A} = \{t\colon (\mathsf{math::add}\langle t\rangle, t) \in \mathbb{T}_\mathsf{A}\}.$$

A bijective mapping between $\mathbb{T}'_\mathsf{A}$ and $\hat{\mathbb{T}}_\mathsf{A}$ can be easily defined

$$(\mathsf{math::add}\langle t\rangle, t) \equiv t.$$

In other words, the relevant model set of Magma is equivalent to a the single-type set $\hat{\mathbb{T}}_{\mathsf{A}}$. As $\hat{\mathbb{T}}_{\mathsf{A}}$ has the same parameters as concept C it can be considered as type parameter adaption towards C and we can write $\hat{\mathbb{T}}_{\mathsf{A}}(C)$. Furthermore, it is a superset of C's model set

$$\mathbb{T}_{\mathsf{C}} \subseteq \hat{\mathbb{T}}_{\mathsf{A}}(C) \equiv \mathbb{T}'_{\mathsf{A}} \subseteq \mathbb{T}_{\mathsf{A}}.$$

Resuming, the dependency of the type parameter reduces the model set $\mathbb{T}_{\mathsf{A}}$ to $\mathbb{T}'_{\mathsf{A}}$, which in turn is equivalent to a set of tuples with fewer parameters $\hat{\mathbb{T}}_{\mathsf{A}}$. Finally, the model set $\mathbb{T}_{\mathsf{C}}$ is a subset of $\hat{\mathbb{T}}_{\mathsf{A}}$.

This relatively complicated construction allows us to express the inclusion relationship on fewer parameters.

Another reason why the arity of the general concept can be higher is parameter replication, for instance

```
concept B<typename T, typename U> : A<U, T, U, T, U> {};
```

Note that the order of type parameters can be arbitrary; the only restriction is that A cannot have more free parameters than B.

As in the previous example, we can extend the model set of B or adapting the model set of A to this context. The latter option provides the advantage that we can treat all examples, including the first one, equally.

$$\begin{aligned} \hat{\mathbb{T}}_{\mathsf{A}}(B) &\overset{\text{def}}{=} \{(t_1, t_2)\colon (t_2, t_1, t_2, t_1, t_2) \in \mathbb{T}_{\mathsf{A}}\} \\ \mathbb{T}_{\mathsf{B}} &\subseteq \hat{\mathbb{T}}_{\mathsf{A}}. \end{aligned}$$

As one can see, the parameters in the tuple must be reordered appropriately.

The examples show that the arity adaption is possible in both ways: increasing and decreasing the number of parameters. The last example also shows that the order of parameters can change.

6.2.3 General Type Inclusion for Concept Refinement

In order to handle the general case of concept refinement, we introduce the notion of *type parameter mapping*. The language specification requires that refined concepts depend on at least one template parameter of the defined concept. The simplest case is that the parameter or some of the parameters are used directly, possibly permuted

$$\mathsf{C}\langle T_1, \ldots, T_n\rangle : \mathsf{A}\langle T_k, T_j\rangle \quad \text{with } 1 \leq k, j \leq n.$$

Notice that A's parameters depend on C's template parameters. One can therefore introduce mapping functions ψ_i that map from C's parameters to each of A's parameters. In the case at hand, the mappings to A's first and second parameters are defined as

$$\begin{aligned} \psi_1^{\mathsf{A}}\colon (T_1, \ldots, T_n) &\mapsto T_k \\ \psi_2^{\mathsf{A}}\colon (T_1, \ldots, T_n) &\mapsto T_j. \end{aligned}$$

To shorten the notation, we also use a vectorized mapping

$$\underline{\psi}^{\mathsf{A}} \colon (T_1, \ldots, T_n) \mapsto (T_k, T_j).$$

Part of the A's parameters can be constants, but not all, e.g.:

$$\mathsf{C}\langle T_1, \ldots, T_n\rangle : \mathsf{A}\langle T_k, \textbf{bool}\rangle$$
$$\underline{\psi}^{\mathsf{A}} \colon (T_1, \ldots, T_n) \mapsto (T_k, \textbf{bool}).$$

Refined concepts can also be defined on classes completely or partly templated on C's parameters.

$$\mathsf{C}\langle T_1, \ldots, T_n\rangle : \mathsf{A}\langle \text{std::set<T3, less<T3>, my\_alloc>}, \text{std::vector<T2>}\rangle$$
$$\underline{\psi}^{\mathsf{A}} \colon (T_1, \ldots, T_n) \mapsto (\text{std::set<T3, less<T3>, my\_alloc>}, \text{std::vector<T2>}).$$

Nested classes are also feasible if they depend at least partly on the template parameters of the defined concept, for instance:

$$\mathsf{C}\langle T_1, \ldots, T_n\rangle : \mathsf{A}\langle \text{std::vector<std::vector<T2> >}\rangle$$
$$\underline{\psi}^{\mathsf{A}} \colon (T_1, \ldots, T_n) \mapsto (\text{std::vector<std::vector<T2> >}).$$

Any combination of the parameters above is feasible. Nevertheless, associated types are not allowed as parameters for refinements.

We summarize the parameter mapping with the following:

Property 1 *All parameters of refined concepts depend on the template parameters of the defined concept or are constant. At least one parameter must be non-constant.*

Not all of C's template parameter $T = (T_1, \ldots, T_n)$ are used in each refinement. This leads to

Definition 5 *The template parameters referred to in refined concept A are depicted as $\underline{T/\!\!/_A}$. These parameters have the same relative order as they have in the parameter list of concept C. The remaining parameters are symbolized by $\overline{T/\!\!/_A}$—also preserving the order. The number of referred parameters is denoted as μ_{A}, the number of template parameters not referred to in concept A is depicted as η_{A}. Their sum is the concept arity $n_{\mathsf{C}} = \mu_{\mathsf{A}} + \eta_{\mathsf{A}}$.*

For instance, the following concept

```
concept C<typename T1, typename T2, typename T3, typename T4, typename T5>
  : A1<T4, T1, vector<T3>, class_alpha<T3, T1> > /* , ... */
{}
```

has the template parameters $T = (T_1, T_2, T_3, T_4, T_5)$. Only three of them are referred in A_1: $\underline{T/\!\!/_{\mathsf{A}_1}} = (T_1, T_3, T_4)$ so that $\overline{T/\!\!/_{\mathsf{A}_1}} = (T_2, T_5)$. Thus, $\mu_{\mathsf{A}_1} = 3$ and $\eta_{\mathsf{A}_1} = 2$. The parameter limitation can also be applied to constant types, for instance

$$\underline{(\textbf{bool}, \textbf{int}, \textbf{short}, \textbf{char}, \textbf{float})/\!\!/_{\mathsf{A}_1}} = (\textbf{bool}, \textbf{short}, \textbf{char})$$

In the same way as the mapping from T to a A's parameter list is specified, we define the mapping from $T/\!\!/_{\mathsf{A}}$ to A's parameter list. Since the template parameters not in $T/\!\!/_{\mathsf{A}}$ are not used by $\underline{\psi}^{\mathsf{A}}$, we can define a mapping from $T/\!\!/_{\mathsf{A}}$ to the image of $\underline{\psi}^{\mathsf{A}}$

$$\underline{\psi}^{\mathsf{A}} \colon T \mapsto U \quad \Longleftrightarrow \quad \underline{\varphi}^{\mathsf{A}} \colon T/\!\!/_{\mathsf{A}} \mapsto U \tag{6.5}$$

where U are the template parameters of A (as they are used in the refinement clause). For instance,

$$\begin{aligned}\underline{\psi}^{\mathsf{A}_1}(\textbf{bool}, \textbf{int}, \textbf{short}, \textbf{char}, \textbf{float}) &= \underline{\varphi}^{\mathsf{A}_1}(\textbf{bool}, \textbf{short}, \textbf{char}) \\ &= (\textbf{char}, \textbf{bool}, \mathsf{vector}{<}\textbf{short}{>}, \mathsf{class\_alpha}{<}\textbf{short}, \textbf{char}{>}) \end{aligned}\tag{6.6}$$

The examination of necessary conditions to model refined concepts reveals the usefulness of the parameter limitation and the corresponding parameter mapping. As an example, in order to model the refined concept A_1 from the last example, the tuple must provide a certain structure

$$(t_1, t_2, t_3, t_4) \models \mathsf{A}_1 \quad \Longrightarrow \quad \exists t' \colon t_3 = \mathsf{vector}{<}t'{>} \wedge t_4 = \mathsf{class\_alpha}{<}t', t_2{>}.$$

A new notation is introduced by

Definition 6 *A tuple t that matches the parameter pattern of a concept A's refinement clause is denoted as $t \ltimes \Pi_{\mathsf{A}}$. The set of all these pattern matching tuples is depicted as $\tilde{\mathbb{T}}_{\mathsf{A}} \overset{\text{def}}{=} \{t \colon t \ltimes \Pi_{\mathsf{A}}\}$*

For instance, (**int**, **short**, vector<**float**>, class_alpha<**float**, **short**>) matches the parameter pattern of A_1.

Applying the vectorized parameter mapping $\underline{\varphi}^{\mathsf{A}_1}$ to an arbitrary triplet of types yields a quadruple of types that always matches A_1's parameter pattern

$$\forall t \in \mathbb{T}^3 \colon \underline{\varphi}^{\mathsf{A}_1}(t) \ltimes \Pi_{\mathsf{A}_1}.$$

More generally, for an arbitrary refinement A_i of concept C, a tuple of arity μ_{A_i} can be mapped to a tuple that matches the parameter pattern of A_i's clause

$$\mathsf{C} : \mathsf{A}_i, \ldots \quad \Longrightarrow \quad \forall t \in \mathbb{T}^{\mu_{\mathsf{A}_i}} : \underline{\varphi}^{\mathsf{A}_i}(t) \ltimes \Pi_{\mathsf{A}_i}. \tag{6.7}$$

In order to be relevant for modeling concept C, a tuple of types must at the same time model the refined concept A and match the parameter pattern in refinement clause Π_{A}. We define the set of these types

$$\mathbb{T}'_{\mathsf{A}_i} \overset{\text{def}}{=} \tilde{\mathbb{T}}_{\mathsf{A}_i} \cap \mathbb{T}_{\mathsf{A}_i} \subseteq \mathbb{T}_{\mathsf{A}_i}.$$

We also define the set of tuples of arity μ_{A_i} that can be projected to a model of A_i

$$\mathbb{T}''_{\mathsf{A}_i} \overset{\text{def}}{=} \{t \in \mathbb{T}^{\mu_{\mathsf{A}_i}} : \underline{\varphi}^{\mathsf{A}_i}(t) \models \mathsf{A}_i\}.$$

Despite the potentially different arities of their elements, each element in $\mathbb{T}''_{\mathsf{A}_i}$ can be associated with exactly one element in $\mathbb{T}'_{\mathsf{A}_i}$

$$\mathbb{T}'_{\mathsf{A}_i} \equiv \mathbb{T}''_{\mathsf{A}_i}.$$

Furthermore, when a tuple t models concept C, then its limitation to the referred parameters $t /\!\!/_{\mathsf{A}_i}$ can be mapped to a tuple that:

- Matches the clause pattern due to the mapping's definition and
- Models the refined concept due to refinement properties.

That is

$$t \models \mathsf{C} \quad \Longrightarrow \quad \underline{\varphi}^{\mathsf{A}_i}(t /\!\!/_{\mathsf{A}_i}) \ltimes \Pi_{\mathsf{A}_i} \wedge \underline{\varphi}^{\mathsf{A}_i}(t /\!\!/_{\mathsf{A}_i}) \models \mathsf{A}_i.$$

As only the referred parameters are interesting for modeling A_i we introduce another set that separates the parameters referred and unreferred in A_i. This set $\ddot{\mathbb{T}}_{\mathsf{C}}(\mathsf{A}_i)$ is a superset of $\mathbb{T}_{\mathsf{C}}$

$$\ddot{\mathbb{T}}_{\mathsf{C}}(\mathsf{A}_i) \overset{\text{def}}{=} \{t \in \mathbb{T}^{n_{\mathsf{C}}} : \underline{\varphi}^{\mathsf{A}_i}(t /\!\!/_{\mathsf{A}_i}) \models \mathsf{A}_i \wedge \overline{t /\!\!/_{\mathsf{A}_i}} \in \mathbb{T}^{\eta_{\mathsf{A}_i}}\} \supseteq \mathbb{T}_{\mathsf{C}}$$

This set can be represented as the following direct product:

$$\ddot{\mathbb{T}}_{\mathsf{C}}(\mathsf{A}_i) = \mathbb{T}''_{\mathsf{A}_i} \times \mathbb{T}^{\eta_{\mathsf{A}_i}}.$$

Together with the inclusions and equivalences above, this yields

$$\mathbb{T}_{\mathsf{C}} \subseteq \ddot{\mathbb{T}}_{\mathsf{C}}(\mathsf{A}_i) = \mathbb{T}''_{\mathsf{A}_i} \times \mathbb{T}^{\eta_{\mathsf{A}_i}} \equiv \mathbb{T}'_{\mathsf{A}_i} \times \mathbb{T}^{\eta_{\mathsf{A}_i}} \subseteq \mathbb{T}_{\mathsf{A}_i} \times \mathbb{T}^{\eta_{\mathsf{A}_i}}. \tag{6.8}$$

The last term we call

$$\hat{\mathbb{T}}_{\mathsf{A}_i} \overset{\text{def}}{=} \mathbb{T}_{\mathsf{A}_i} \times \mathbb{T}^{\eta_{\mathsf{A}_i}}.$$

Since this inclusion holds for every refinement, the general form of Equation (6.3) extends to

$$\mathsf{C} : \mathsf{A}_1, \ldots, \mathsf{A}_n \quad \Longrightarrow \quad \mathbb{T}_{\mathsf{C}} \subseteq \bigcap_{i \in [1,n]} \hat{\mathbb{T}}_{\mathsf{A}_i}. \tag{6.9}$$

6.2.4 General Type Inclusion in Concept Restriction

The inclusion of model sets applies in the same way to **where** clauses as it does to refinements. The difference is that requests can be defined on associated types so that more parameter mappings need to be taken into consideration. An associated type of the defined concept

```
concept C<typename T1, ..., typename TN>
{
  typename assoc_type;
}
```

is considered dependent on all template parameters because it may be different for each combination of types

$$\psi^{\mathsf{a}}\colon (T_1,\ldots,T_n) \mapsto \text{C<T1, ..., TN>::assoc\_type},$$

using a as abbreviation for assoc_type. The same argument applies to associated types that are automatically detected from result types of functions since the arguments of these functions depend directly or indirectly on the concept parameters or type constants. Associated types of other concepts or classes are mapped similarly as templated classes, for instance

$$\mathsf{C}\langle T_1,\ldots,T_n\rangle \dashv \mathsf{B}\langle \text{std::set<T3>::reference}, \text{std::vector<T2>::pointer}\rangle$$
$$\underline{\psi}^{\mathsf{B}}\colon (T_1,\ldots,T_n) \mapsto (\text{std::set<T3>::reference}, \text{std::vector<T2>::pointer}).$$

By back-substituting all associated types of the defined concepts, every request is expressed depending on the concept parameter or type constants. For instance,

```
concept C<typename T1, typename T2, typename T3>
{
  typename assoc_type;
  typename type2 = D1<T3>::result_type;
  typename type3 = D2<vector<type2>::pointer, T2>::reference;

  where B<type3, D3<T3>::pointer>;
}
```

can be transformed into

```
concept C<typename T1, typename T2, typename T3>
{
  typename assoc_type;
  typename type2 = D1<T3>::result_type;
  typename type3 = D2<vector<type2>::pointer, T2>::reference;

  where B<D2<vector<D1<T3>::result_type>::pointer, T2>::reference,
          D3<T3>::pointer>;
}
```

where all type parameters of B depend on the concept parameters.

With these additional mappings, the intermediate sets from Section 6.2.3 can be used in the same manner for requests. In addition, $\hat{\mathbb{T}}_{\mathsf{B}_j}$ is defined correspondingly to $\hat{\mathbb{T}}_{\mathsf{A}_i}$. Therefore, the model set of concept C is a subset of the *adapted* model sets of all requested concepts

$$\mathsf{C} \dashv \mathsf{B}_1,\ldots,\mathsf{B}_m \quad\Longrightarrow\quad \mathbb{T}_{\mathsf{C}} \subseteq \bigcap_{j\in[1,m]} \hat{\mathbb{T}}_{\mathsf{B}_j}. \tag{6.10}$$

Furthermore, each type tuple that models concept C must model all refined and all requested concepts. Thus, Equations (6.9) and (6.10) can be combined to equation

$$\mathsf{C} : \mathsf{A}_1, \ldots, \mathsf{A}_n \dashv \mathsf{B}_1, \ldots, \mathsf{B}_m \quad \Longrightarrow \quad \mathbb{T}_{\mathsf{C}} \subseteq \bigcap_{i \in [1,n]} \hat{\mathbb{T}}_{\mathsf{A}_i} \cap \bigcap_{j \in [1,m]} \hat{\mathbb{T}}_{\mathsf{B}_j} \tag{6.11}$$

6.3 Implicit Model Declarations

This section discusses which model declarations are implicitly inserted by the compiler for different definitions of a concept. Conversely, one can start designing concepts by initially deciding which model declarations imply each other and then define the concepts accordingly.

6.3.1 Model Implication Rules

Nominally confirmed refinements only imply that a type modeling the defined concept implicitly models all refined concepts. The reverse is not true.

$$\frac{\mathsf{C} : \mathsf{A}_1, \ldots, \mathsf{A}_n \dashv \mathsf{B}_1, \ldots, \mathsf{B}_m}{t \models \mathsf{C} \quad \Longrightarrow \quad t \models_i \mathsf{A}_n \wedge \cdots \wedge t \models_i \mathsf{A}_1} \tag{6.12}$$

Automatic refinement implies model declaration in two directions. The model declaration of the concepts refined from is again implied. In addition, types that model all requested and all refined concepts automatically model the defined concept.

$$\frac{\mathsf{auto\ C} : \mathsf{A}_1, \ldots, \mathsf{A}_n \dashv \mathsf{B}_1, \ldots, \mathsf{B}_m}{\begin{array}{c} t \models \mathsf{C} \quad \Longrightarrow \quad t \models_i \mathsf{A}_1 \wedge \cdots \wedge t \models_i \mathsf{A}_n \\ t \models \mathsf{A}_1 \wedge \cdots \wedge t \models \mathsf{A}_n \wedge t \models \mathsf{B}_1 \wedge \cdots \wedge t \models \mathsf{B}_m \quad \Longrightarrow \quad t \models_i \mathsf{C} \end{array}} \tag{6.13}$$

Equations (6.12) and (6.13) can also be expressed for multi-type concepts. If the arities are different the adaption from Section 6.2 must be applied. For the sake of syntactic clarity, we refrain from it here. However, the example in the following section illustrates multi-type model implication.

Concept maps are similar to **auto** concepts concerning the implication from the requested concepts towards the defined concept. The difference is that the opposite implication is not given in general

$$\frac{T \models_p \mathsf{C} \dashv \mathsf{B}_1\langle T \rangle, \ldots, \mathsf{B}_m\langle T \rangle}{t \models \mathsf{B}_1 \wedge \cdots \wedge t \models \mathsf{B}_m \quad \Longrightarrow \quad t \models_i \mathsf{C}} \tag{6.14}$$

Or in terms of model sets

$$\bigcap_{j \in [1,m]} \mathbb{T}_{\mathsf{B}_j} \subseteq \mathbb{T}_{\mathsf{C}} \tag{6.15}$$

Note that in contrast to Section 6.2 the intersection of $\mathbb{T}_{\mathsf{B}_j}$ is included in $\mathbb{T}_{\mathsf{C}}$, not vice versa.

Multiple concept maps are treated as disjunction of the model declarations

$$\frac{T \models_p \mathsf{C} \dashv \mathsf{B}_1\langle T \rangle \wedge \cdots \wedge T \models_p \mathsf{C} \dashv \mathsf{B}_m\langle T \rangle}{t \models \mathsf{B}_1 \vee \cdots \vee t \models \mathsf{B}_m \quad \Longrightarrow \quad t \models_i \mathsf{C}} \tag{6.16}$$

Expressed as model sets it reads

$$\bigcup_{j\in[1,m]} \mathbb{T}_{\mathsf{B}_j} \subseteq \mathbb{T}_{\mathsf{C}} \tag{6.17}$$

Multiple maps of one concept is the only situation where unions of sets are involved.

6.3.2 Case Study: Vector Space

Vector spaces are mathematical structures of the following two element types:

- The vector type, which must model an AdditiveAbelianGroup and
- The scalar type, which must model a Field.

In most cases, the scalar type is the value_type of the vector type. Thus, at least three operations are defined over the element types individually: vector addition and scalar addition and multiplication. In addition, the two types must be distributive with regard to scalar-vector multiplication and both additions. To simplify notation, we consider the case that all operations are denoted as operators and we only sketch the syntactic requirements.

```
concept VectorSpace<typename Vector,
                    typename Scalar = typename Vector::value_type>
  : AdditiveAbelianGroup<Vector>,
    Field<Scalar>
{
    // Require operators of mixed multiplication and respective assignabilities
    axiom Distributivity(Vector v, Vector w, Scalar a, Scalar b)
    {
        a * (v + w) == a * v + a * w;
        (a + b) * v == a * v + b * v;
    }
}
```

As VectorSpace is not an auto-concept, all models must be declared explicitly. (We assume in this section that VectorSpace is not refined by other concepts.) Declaring a type v (or the pair (v, s)) to be a model of VectorSpace implies the model declaration of v as AdditiveAbelianGroup and v::value_type (or s) will be implicitly declared to model Field

$$(v, s) \models_p \mathsf{VectorSpace} \quad \Longrightarrow \quad v \models_i \mathsf{AdditiveAbelianGroup} \wedge s \models_i \mathsf{Field}.$$

If all model declarations of VectorSpace are correct then the the implied declarations for AdditiveAbelianGroup and Field are also correct. Incorrect declarations of VectorSpace can imply incorrect declarations of AdditiveAbelianGroup and Field. If for instance somebody accidently declares a vector of integer as VectorSpace then integer will be incorrectly declared as Field. The wrong declaration affects *the whole program*!

In turn, erroneous implicit model declarations can only be corrected by removing the causing model declaration from the program. Currently, no tool exist to trace the inference of models. Thus, in order to avoid an accidental declaration of type t modeling concept C, the programmer must inspect the whole source code to find all concepts directly or indirectly refining C and examining those concepts' model declarations.

The wrong model implication of Field can be avoided by changing this requirement from a refinement into a **where** clause:

```
concept VectorSpace<typename Vector,
                    typename Scalar = typename Vector::value_type>
  : AdditiveAbelianGroup<Vector>
{
    where Field<Scalar>;

    // Require operators of mixed multiplication and respective assignabilities
    axiom Distributivity(Vector v, Vector w, Scalar a, Scalar b) { /* ... */ }
}
```

Declaring an integer vector to model VectorSpace will now cause the compiler to *verify* whether integer models Field. The programmer must now explicitly declare that integer models Field (assuming that this not already done by another wrong declaration).

The incorrect VectorSpace model still can be forced by incorrectly declaring integer modeling Field. However, it is more likely that the programmer—before writing the wrong declaration—verifies the requirements of Field. Then he or she will realize that—despite the existence of integer division—the notion of reciprocal elements is not fulfilled.

In other words, changing a refinement into a request does not avoid wrong model declarations. Nevertheless, it introduces a higher level of security by forcing the programmer to *declare the wrong model directly* instead of indirectly.

Another criterion for the concept design is the role a type plays in a concept. The VectorSpace concept is primarily a description of the Vector type and the Scalar type plays a more secondary role. From this prospective, most people might not be aware that statement on the VectorSpace impacts the behavior of the Scalar type in other contexts.

The refinement from AdditiveAbelianGroup is less critical because the demand for additivity of the Vector type with all its properties is more obvious for VectorSpaces. Nevertheless, this is a design decision and other programmers might prefer to request AdditiveAbelianGroup in a **where** clause instead of a refinement.

Rule 1 *Restriction with more* **where** *clauses and less refinements reduces the risk of accidental model declarations but comes at the price of demanding more model declarations from the programmer. Conversely, replacing requests by refinements does not falsify the concepts but complicates the detection of wrong model declarations.*

What are sufficient conditions that a pair of types models VectorSpace? Obviously, the two types must model AdditiveAbelianGroup and Field. In addition, the distributivity—defined in the concept body—must be also fulfilled. These three conditions are sufficient for modeling VectorSpace. In order to declare distributivity independent on VectorSpace, it must be defined in a separate concept Distributive: VectorSpace is **auto**-matically modeled.

```
concept Distributive<typename Vector,
                     typename Scalar = typename Vector::value_type>
{
    // Require operators of mixed multiplication and respective assignabilities
    axiom Distributivity(Vector v, Vector w, Scalar a, Scalar b) { /* ... */ }
}

auto concept VectorSpace<typename Vector,
                         typename Scalar = typename Vector::value_type>
  : AdditiveAbelianGroup<Vector>,
    Distributive<Vector, Scalar>
{
    where Field<Scalar>;
}
```

Given that for a pair of types all three prerequisite concepts are modeled, Conversely, declaring a VectorSpace model implies modeling AdditiveAbelianGroup and Distributive for the corresponding types. With the abbreviation VS, A, D, and F for VectorSpace, AdditiveAbelianGroup, Distributive, and Field respectively, we can represent the model set implication as

$$\frac{\mathsf{auto\ VS}\langle v,s\rangle : \mathsf{A}\langle v\rangle, \mathsf{D}\langle v,s\rangle \dashv \mathsf{F}\langle s\rangle}{\begin{array}{c}(v,s) \models_p \mathsf{VS} \quad\Longrightarrow\quad v \models_i \mathsf{A} \wedge (v,s) \models_i \mathsf{D} \\ v \models_p \mathsf{A} \wedge (v,s) \models_p \mathsf{D} \wedge s \models_p \mathsf{F} \quad\Longrightarrow\quad (v,s) \models_i \mathsf{VS}\end{array}}$$

6.3.3 Design of Restricting Concepts based on Model Implication

In this section, we investigate how concepts can be designed on the base of given model set implications. We use in all following examples the notation that concept Z restricts concepts X and Y. Concept Z will be referred to as 'defined concept' or the 'special concept' and concepts X and Y as 'restricting concepts' or 'general concepts'.

As first example, we search a concept definition so that for all types modeling both concepts X and Y, the modeling of Z is implied

$$t \models \mathsf{X}, \mathsf{Y} \quad\Longrightarrow\quad t \models \mathsf{Z}.$$

This implication can be realized with an *auto* concept *requesting* concepts X and Y.

```
auto concept Z<typename T>
{
  where X<T>;
  where Y<T>;
};
```

The opposite case that modeling Z implies modeling both X and Y

$$t \models \mathsf{X}, \mathsf{Y} \quad \Longleftarrow \quad t \models \mathsf{Z}$$

can be implemented by refinement without request.

```
concept Z<typename T>
  : X<T>, Y<T>
{};
```

Note that the concept must not be **auto**. Defining the concept **auto** (without **where** clause)

```
auto concept Z<typename T>
  : X<T>, Y<T>
{};
```

establishes an equivalence between Z and the conjugation of concepts X and Y

$$t \models \mathsf{X}, \mathsf{Y} \quad \Longleftrightarrow \quad t \models \mathsf{Z}$$

In other words, their model sets are equal

$$\mathbb{T}_{\mathsf{X}} \cap \mathbb{T}_{\mathsf{Y}} = \mathbb{T}_{\mathsf{Z}}.$$

An **auto** refinement without **where** clauses from a single concept introduces an equivalence of the concepts. For instance, the synonym SkewField for DivisionRing can be established with:

```
auto concept SkewField<typename T> : DivisionRing<typename T> {}
```

Auto-refining DivisionRing from SkewField would have the same effect.

It is also possible to define the concept in a manner that modeling Z implies only modeling X but not Y (or vice versa).

$$t \models \mathsf{X} \quad \Longleftarrow \quad t \models \mathsf{Z}$$

by refining from X and requesting Y

```
concept Z<typename T>
  : X<T>
{
  where Y<T>;
};
```

The opposite case that modeling only X but not Y implies modeling Z is not possible to declare. This case makes no sense mathematically: either modeling X and Y is a sufficient condition for modeling Z or it is not because more conditions must be fulfilled. Thus, there is no possibility that only modeling X implies modeling Z.

Of course it is also possible to establish unrelated model sets in both directions using requests without auto.

```
concept Z<typename T>
{
  where X<T>;
  where Y<T>;
  /* ... */
};
```

The added comment stands for additional conditions since—as stated in the last paragraph—the fact that modeling X and Y does not imply modeling Z means that more conditions exist. Then these conditions should be stated in the concept as **where** clause or **axiom**. This leads to

Rule 2 *Non-auto concepts without axioms are incomplete.*

If the concept has semantic requirements, they must be stated in an axiom. Without semantic requirements, models can be automatically deduced from modeling all restricting concepts and fulfilling the syntactic requirements. The converse results in

Rule 3 *Auto concepts with axioms are wrong.*

Semantic requirements cannot be verified by the compiler; thus, their existence prohibits model set implication from the restricted concepts.

Is it possible that the disjunction of modeling two restrictions implies modeling the defined concept

$$t \models \mathsf{X} \vee t \models \mathsf{Y} \quad \Longrightarrow \quad t \models \mathsf{Z}?$$

Expressed in terms of model sets, the union of X's and Y's model sets is included in Z's model set

$$\mathbb{T}_\mathsf{X} \cup \mathbb{T}_\mathsf{Y} \subseteq \mathbb{T}_\mathsf{Z}.$$

The restriction requires the inclusion of Z's model set in the union of X's and Y's model sets, see Section 6.2

$$\mathbb{T}_\mathsf{Z} \subseteq \mathbb{T}_\mathsf{X} \cap \mathbb{T}_\mathsf{Y}.$$

It follows that the concepts X and Y have the same model sets

$$\mathbb{T}_\mathsf{X} \cup \mathbb{T}_\mathsf{Y} \subseteq \mathbb{T}_\mathsf{Z} \subseteq \mathbb{T}_\mathsf{X} \cap \mathbb{T}_\mathsf{Y} \quad \Longrightarrow \quad \mathbb{T}_\mathsf{X} = \mathbb{T}_\mathsf{Y} = \mathbb{T}_\mathsf{Z}. \tag{6.18}$$

This means that the concepts X and Y are equivalent; they might be formulated with different refinements and different requests but they are perfectly interchangeable with each other. Furthermore, concept Z is also equivalent. For the sake of clarity, it is advisable to verify at this point whether really three names for the same concept are needed. To answer the introducing question: the disjunction is possible but the equivalence (6.18) deprives its usefulness. Thus, the fact that the disjunction is not expressible with the current language specification does not constitute a restriction.

The implementations of the expressible implications are summarized in

Table 6.1: Design guide for concept restriction

Model implication	*Implementation*
$t \models \mathsf{X}, \mathsf{Y} \rightarrow t \models \mathsf{Z}$	auto Z ⊣ X, Y
$t \models \mathsf{X}, \mathsf{Y} \leftarrow t \models \mathsf{Z}$	Z : X, Y
$t \models \mathsf{X} \leftarrow t \models \mathsf{Z}$	Z : X ⊣ Y
$t \models \mathsf{X}, \mathsf{Y} \leftrightarrow t \models \mathsf{Z}$	auto Z : X, Y

6.3.4 Model Implication of Non-Restricting Concepts

In contrast to the last section, we assume that concept Z does not restrict concepts X and Y. This lack of restriction turns the model implications into *sufficient* conditions, which can be realized by means of

Rule 4 *Concept maps implement sufficient conditions.*

Automatic concepts also realize sufficient conditions. The differences are that

- Automatic concepts also require the types to model the restricted concepts and
- Concept maps enables disjunction.

Thus, automatic concept definitions state *sufficient and necessary* conditions.

Firstly, we consider the objective that for every type holds that modeling both X and Y implies modeling Z

$$t \models \mathsf{X}, \mathsf{Y} \quad \Longrightarrow \quad t \models \mathsf{Z}.$$

This situation is given by concept maps in Equation (6.14). Thus, the implication above can be implemented with

```
template <typename T>
  where X<T> && Y<T>
concept_map Z<T> {}
```

The opposite case that modeling Z implies modeling both X and Y

$$t \models \mathsf{Z} \quad \Longrightarrow \quad t \models \mathsf{X}, \mathsf{Y}$$

is realized by two model declarations

```
template <typename T>
  where Z<T>
concept_map X<T> {}

template <typename T>
  where Z<T>
concept_map Y<T> {}
```

Using logical 'or' in the **where** clause is not recommended since it is currently not clear whether this will be supported by compilers. Likewise, the behavior that modeling X *or* Y implies modeling Z

$$t \models \mathsf{X} \lor t \models \mathsf{Y} \quad \Longrightarrow \quad t \models \mathsf{Z}.$$

is implemented with two model declarations

```
template <typename T>
  where X<T>
concept_map Z<T> {}

template <typename T>
  where Y<T>
concept_map Z<T> {}
```

The opposite case—that modeling Z implies modeling either X or Y—is mathematically valid but not specific enough be useful in programming, at least imperative programming. [1]

Summarizing the previous model implications of non-restricting concepts leads to

Table 6.2: Design guide for non-restricting concepts

Model implication	*Implementation*
$t \models \mathsf{X}, \mathsf{Y} \to t \models \mathsf{Z}$	$T \models_p \mathsf{Z} \dashv \mathsf{X}\langle T\rangle, \mathsf{Y}\langle T\rangle$
$t \models \mathsf{Z} \to t \models \mathsf{X}, \mathsf{Y}$	$T \models_p \mathsf{X} \dashv \mathsf{Z}\langle T\rangle \land T \models_p \mathsf{Y} \dashv \mathsf{Z}\langle T\rangle$
$t \models \mathsf{X}, \mathsf{Y} \to t \models \mathsf{Z}$	$T \models_p \mathsf{Z} \dashv \mathsf{X}\langle T\rangle \land T \models_p \mathsf{Z} \dashv \mathsf{X}\langle T\rangle$

6.4 Comparison between Refinement and Requests

Resuming the results from this chapter, the differences between refinements and requests are the following:

- Refinements imply model declarations from the defined concept to all concepts refined from and requests have no impact on the restricted concepts.
- Requests can be defined on associated types of:
  - The defined concept,
  - Other concepts, and
  - Classes.

  Refinements cannot refer to associated types.

[1] **Question:** What happens with cyclic model declarations?

Thus, all requirements on associated types must be defined as request. On the other hand, declaring concept refinements on associated types would introduce many hard to maintain side effects in terms of implied model declaration.

Implicit model declaration is a very important design criterion since:

- Too many refinements result in strong interference between concepts and it is more difficult to locate where a model declaration in question is implied from.
- Too few refinements demand the programmer to declare too many declaration by hand

There is no absolute rule when a certain restriction should be a refinement or a request. However, we can adapt a rule of thumb from object oriented programming to choose between membership and inheritance. If one can say for two classes A and B: "A is-a B" then B should be a base class of A. If one said rather: "A has-a B" then B should be a member of A. Nevertheless, this rule is not sharp and leaves space for subjective interpretation.

Correspondingly, we can suggest the following guideline for concepts:

Rule 5 *Let C and A be two concepts where the set of C's requirements is a super-set of A's requirements. In case one would say "C is-a A" then C should be a refinement of A. In case that is more appropriate to say "C need-a A", then A should be a* **where** *clause of C.*

For the sake of conciseness, we repeat this in

Table 6.3: Choice between refinement and request

Description	*Implementation*
C is-a A	C : A
C need-a A	C ⊣ A

Chapter 7

Conclusions

The algebraic concepts presented in this document specify, as all concepts do, syntactic and semantic requirements on types. Syntactic demands serve amongst others to type check template function calls before instantiation and provide the programmer in case of an erroneous call with a message that is clearly more concise and often also more meaningful than current error messages.

Despite the distinct importance of explicating error messages for the program development, we consider the ability to characterize semantic requirements—which is novel to C++—by far more important. Semantic errors were completely invisible to compilers before the introduction of concepts and resulted in unobstructed compilations generating incorrect executables. On-going research addresses the question how to efficiently declare and verify semantic properties that only hold on single values of a type.

A remarkable advantage of concepts is that programmers are now able to oppose the semantic properties required by a generic function to the semantic properties hold by a type. In non-generic programs, the comparison between demanded and fulfilled properties is mentally executed by the programmer before or while writing a function for a certain type. The lack of exposing semantic properties in programs makes it difficult or impossible for others to understand and verify the correctness of the containing computations, unless massive documentation on the whole mathematical background exists. Is this the reason why non-trivial numeric software is scarcely extended by somebody else than the authors?

The importance of exposing mathematical properties within program sources and then verifying them in the compilation cannot be overstated. We call this new paradigm of embedding semantic behavior

Property-Aware Programming

The design of concepts is discussed in this paper based on our experience. The question whether refinement or requesting is preferable in a certain concept is (similarly to the question in OO whether inheritance or membership is better suited for a certain class) not answerable with mathematical stringency but one can provide good guidelines. If one would say "concept X 'is-a' Y" then X should be a refinement of Y and if one would say "concept X 'needs-a' Y" then X should request Y.

An important behavior of concepts is that refinement adds requirements and removes types from the model set. This behavior is less obvious for refinement of multi-type concepts, especially if the refined concepts have different arities. It could be shown that the inclusion of model sets holds for refinement from concepts with fewer parameters as well as from concepts with more parameters. The same inclusive behavior holds for concepts requested in **where** clause. Furthermore, we give the complete set of rules that define implied model sets.

Returning to the algebraic character of this paper's concepts, they only represent the core of algebraic structures and build a base for other algebraic concepts. We distinguish the following three categories of concepts:

- The purely algebraic concepts that only contain mathematical properties on arbitrary operations;
- The augmented concepts that add basic implementation requirements; and
- The operator-based concepts.

The first category of concepts should be used if the function constraints need to be specified with very fine granularity in order to enable the absolute maximum set of model types. The second category represents a slight restriction over the first one by adding requirements like assignability and convertibility to the concepts. We expect that these additional requirements do not reduce the model sets significantly. Furthermore, we assume that many generic functions need assignability and convertibility and their lists of conceptual prerequisites can be shortened by using the augmented concepts. For numeric functions expressed in terms of standard arithmetic operators, the operator based concepts are the most suitable.

The example of the power function illustrates how the computation is defined on different ranges and implemented with different algorithms depending on the algebraic concept modeled. The calculation of a generic reduction of n values w.r.t. an arbitrary binary operation showed that concepts allow the programmer to specify the admissibility of performance optimization techniques. How to address floating point rounding errors in a general manner was also illustrated on the example of generic reduction.

The application of these algebraic concepts within concepts in vector spaces is not shown in this document and will be published in a future paper. Nevertheless, we like to mention here that the concepts have proven to work on non-scalar data types like vectors.

The flexible definition of the concepts, especially the fact that the return type of an operation is not required to be identical with the arguments, enabled the concept checking of expression templates. At the moment, only a fixed set of expressions is supported, which is sufficient to express all common linear solvers with natural operator representation, and research is in progress to support arbitrary expressions.

This excursion into vector spaces and linear solvers gives a fore-taste of the potential of mathematical concepts to emerge into all domains of scientific computing and to initiate a new age of scientific computing that can be strongly impacted by property-aware programming enabling a new quality of mathematical software reliability.

Acknowledgment: I wish to thank Laura Hopkins for her intensive support on the linguistic details of the whole text. I also thank Torsten Höfler for his constructive discussions.

Part II

Concept Specifications

Chapter 8

Single-Operation Concepts

Concepts with one operation are expressible in different manners: with a given or with a freely chooseable operation. We will first introduce the general concepts where the operation is implemented with a functor realizing an arbitrary binary operation. This part will be separated into two sub-parts: one defining only mathematical properties in the concepts and the other adding very basic implementation requirements to ease their utilization.

Later we define concepts specialized to addition and multiplication as the most important binary operations. These concepts use operators instead of functors because this is the common practice in numerical libraries.

Thus, we have the following three categories of concepts:

- Purely algebraic concepts,
- Augmented concepts, and
- Operator-based concepts.

The same categories are used for concepts with two operations.

8.1 Purely Algebraic Concepts

All concepts in this section characterize tuples of two types: a functor that represents an operation and a type the functor operates on, which are the elements of the set. They are all defined in the namespace `algebra`. Purely algebraic concepts are meant to specify only mathematical properties. Unfortunately, there is no proper way to characterize the closure of an operation

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

Therefore we omit this property and start with commutativity.

8.1.1 Commutative

The concept commutative formalizes that the order of arguments in a binary operation can change.

Notation

{Operation, Element}	are types that build a model of Commutative.
op	is an object of type Operation.
x, y	are objects of type Element.

Valid Expressions

- Commutativity
  op(x, y)
  Return Type: Arbitrary
  Semantics: Arbitrary

Invariants

- Commutativity
  op(x, y) = op(y, x)

Implementation in ConceptGCC

```
concept Commutative<typename Operation, typename Element>
{
    axiom Commutativity(Operation op, Element x, Element y)
    {
        op(x, y) == op(y, x);
    }
};
```

8.1.2 Associative

Associative operations can be executed in different orders.

Notation

{Operation, Element}	are types that build a model of Associative.
op	is an object of type Operation.
x, y	are objects of type Element.

Valid Expressions

- Associativity
  op(x, y)
  Return Type: Arbitrary
  Semantics: Arbitrary

Invariants

- Associativity
  op(op(x, y), z) = op(x, op(y, z))

Implementation in ConceptGCC

```
concept Associative<typename Operation, typename Element>
{
    axiom Associativity(Operation op, Element x, Element y, Element z)
    {
        op(x, op(y, z)) == op(op(x, y), z);
    }
};
```

8.1.3 SemiGroup

A *Semi-Group* is an algebraic structure closed under its binary operation and associative. As we cannot specify the closure, algebra::SemiGroup and Associative are equivalent, confer Section 6.3 on concept equivalence.

Auto-Refinement of

Associative

Implementation in ConceptGCC

```
auto concept SemiGroup<typename Operation, typename Element>
  : Associative<Operation, Element>
{};
```

8.1.4 Monoid

Refinement of

algebra::SemiGroup

Associated Types

identity_result_type	The result type of identity, automatically detected, normally the same as Element.

Notation

{Operation, Element}	are types that build a model of Commutative.
op	is an object of type Operation.
x	is an object of type Element.

Valid Expressions

In addition to the expressions defined in algebra::SemiGroup, the following expressions must be valid:

- Identity element
  identity(op, x)
  Return Type: identity_result_type
  Semantics: See invariants

Invariants

- Neutrality from left
  op(identity(op, x), x) = x

- Neutrality from right
  op(x, identity(op, x)) = x

Implementation in ConceptGCC

```
concept Monoid<typename Operation, typename Element>
  : SemiGroup<Operation, Element>
{
    typename identity_result_type;
    identity_result_type identity(Operation, Element);

    axiom Neutrality(Operation op, Element x)
    {
        op( x, identity(op, x) ) == x;
        op( identity(op, x), x ) == x;
    }
};
```

8.1.5 Inversion

This concept introduces the notion of generic inversion, which is not necessarily defined on all elements.

Associated Types

inverse_result_type	The result type of inverse, automatically detected, normally the same as Element.

Notation

{Operation, Element}	are types that build a model of Commutative.
op	is an object of type Operation.
x	is an object of type Element.

Valid Expressions

- Inversion
  inverse(op, x)
  Return Type: inverse_result_type
  Semantics:

Invariants

- Cancellation from left
  op(inverse(op, x), x) = identity(op, x) if inversion is defined for x
- Cancellation from right
  op(x, inverse(op, x)) = identity(op, x) if inversion is defined for x

Implementation in ConceptGCC

```
auto concept Inversion<typename Operation, typename Element>
{
    typename inverse_result_type;
    inverse_result_type inverse(Operation, Element);

};
```

Notes

Using the inversion in its pure form is very dangerous. It should be either used with an explicit invertibility check, see PartiallyInvertibleMonoid, or with concepts where invertibility is guaranteed for all elements, see algebra::Group.

8.1.6 Group

A *Group* is a monoid with inversion on all elements.

Refinement of

algebra::Monoid and Inversion.

Notation

{Operation, Element}	are types that build a model of Commutative.
op	is an object of type Operation.
x	is an object of type Element.

Valid Expressions

Same as algebra::Monoid and Inversion.

Invariants

- Cancellation from left
  op(inverse(op, x), x) = identity(op, x)
- Cancellation from right
  op(x, inverse(op, x)) = identity(op, x)

Implementation in ConceptGCC

```
concept Group<typename Operation, typename Element>
  : Monoid<Operation, Element>, Inversion<Operation, Element>
{
    axiom Inversion(Operation op, Element x)
    {
        op( x, inverse(op, x) ) == identity(op, x);
        op( inverse(op, x), x ) == identity(op, x);
    }
};
```

Notes

The cancellation is defined in the manner as in concept Inversion but in this concept the inversion is required to be executable on *every* element.

8.1.7 AbelianGroup

Auto-Refinement of

algebra::Group and Commutative.

Valid Expressions

Same as algebra::Group and Commutative.

Implementation in ConceptGCC

```
auto concept AbelianGroup<typename Operation, typename Element>
  : Group<Operation, Element>, Commutative<Operation, Element>
{};
```

8.2 Augmented Algebraic Concepts

8.2.1 Magma

The concept *Magma* introduces a binary operation on the set and its closure. The closure of a binary operation is defined so that for a set S of elements of type Element and an operation op from type Op, the operation must be defined on all $(a, b) \in S \times S$ and the result must be of type Element.

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

It is augmented with Convertible and Assignable, see. Section 3.3.1 The convertibility represents in some measure the closure, see. Section 3.3.2.

Refinement of

Callable2, Convertible, and Assignable.

Notation

{Operation, Element}	are types that build a model of Magma.
op	is an object of type Operation.
x, y	are objects of type Element.

Valid Expressions

- Operation
  op(x, y)
  Return Type: Element or a type convertible to Element.

Implementation in ConceptGCC

```
namespace math {

auto concept BinaryIsoFunction<typename Operation, typename Element>
{
    where std::Callable2<Operation, Element, Element>;
    where std::Convertible<std::Callable2<Operation, Element, Element>::result_type, Element>;

    typename result_type = std::Callable2<Operation, Element, Element>::result_type;
};

auto concept Magma<typename Operation, typename Element>
    : BinaryIsoFunction<Operation, Element>
{
    where std::Assignable<Element>;
    where std::Assignable<Element, BinaryIsoFunction<Operation, Element>::result_type>;
};

}
```

Remark: the closure is not directly expressed in the concept. Instead it is characterized by the request that the return type is convertible into the element type. Whether the result of the computation and the possibly following conversion is mathematically correct cannot be expressed by the concept and the answer also depends on interpretation as we will explain in section 3.3.2.

We will omit the namespace declaration in later programs. All following mathematical concepts and type traits are defined in the namespace math.

Models

- modN_t<n> with functor implementing + or $*$, see Section 5.1.
- Contingent: (**int**, math::add<**int**>), for limitations see Section 3.3.2.
- Contingent: (**float**, math::add<**float**>) and (**float**, math::mult<**float**>), for limitations see Section 3.3.2.
- Contingent: (complex<**double**>, math::add<complex<**double**> >) and (complex<**double**>, math::mult<complex<**double**> >), same as **float**.

8.2.2 CommutativeMagma

This concept adds the commutativity to a closed binary operation. We introduce it to provide a more precise characterization of floating point numbers, which are commutative but not associative; see also Section 3.3.3.

Refinement of

Magma and Commutative.

Invariants

The same as Commutative.

Implementation in ConceptGCC

```
auto concept CommutativeMagma<typename Operation, typename Element>
  : Magma<Operation, Element>,
    algebra::Commutative<Operation, Element>
{};
```

Models

- modN_t<n> with functor implementing $+$ or $*$, see Section 5.1.
- Contingent: (**int**, math::add<**int**>), for limitations see Section 3.3.2.
- Contingent: (**float**, math::add<**float**>) and (**float**, math::mult<**float**>), for limitations see Section 3.3.2.
- Contingent: (complex<**double**>, math::add<complex<**double**> >) and (complex<**double**>, math::mult<complex<**double**> >), same as **float**.

8.2.3 SemiGroup

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

Refinement of

Magma and algebra::SemiGroup

Invariants

The same as algebra::SemiGroup.

Implementation in ConceptGCC

```
auto concept SemiGroup<typename Operation, typename Element>
  : Magma<Operation, Element>,
    algebra::SemiGroup<Operation, Element>
{};
```

8.2.4 CommutativeSemiGroup

A *Commutative Semi-Group* combines associativity and commutativity.

Refinement of

SemiGroup and CommutativeMagma

Implementation in ConceptGCC

```
auto concept CommutativeSemiGroup<typename Operation, typename Element>
  : SemiGroup<Operation, Element>,
    CommutativeMagma<Operation, Element>
{};
```

8.2.5 Monoid

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

Refinement of

SemiGroup and algebra::Monoid.

Notation

{Operation, Element}	are types that build a model of Monoid.
op	is an object of type Operation.
x	is an object of type Element.

Valid Expressions

- Identity
  identity(op, x)
  Return Type: Element or a type convertible to Element.

Invariants

The same as SemiGroup and algebra::Monoid.

Implementation in ConceptGCC

```
auto concept Monoid<typename Operation, typename Element>
  : SemiGroup<Operation, Element>,
    algebra::Monoid<Operation, Element>
{
    where std::Convertible<identity_result_type, Element>;
};
```

Remark: Refinement is currently not **auto** due to internal problems with ConceptGCC (revision 338).

Models

- STL strings with concatenation-based functor and empty string as identity, see Section 10.1.

Note

The difference to algebra::Monoid is only the convertibility of identity_result_type.

8.2.6 CommutativeMonoid

A *Commutative Monoid* can be considered as monoid that is commutative or alternatively as a commutative semi-group with an identity.

Refinement of

Monoid and CommutativeSemiGroup.

Invariants

The same as

Implementation in ConceptGCC

```
auto concept CommutativeMonoid<typename Operation, typename Element>
  : CommutativeSemiGroup<Operation, Element>,
    Monoid<Operation, Element>
{};
```

Models

- Non-negative reals as positive_real with an addition-like functor type like in Section 10.2.

8.2.7 PartiallyInvertibleMonoid

This concept introduces inversion and a check whether a given element is invertible, see Section 3.3.6.

Refinement of

Monoid and Inversion.

Notation

{Operation, Element}	are types that build a model of SemiGroup.
op	is an object of type Operation.
x	is an object of type Element.

Valid Expressions

- Identity
  is_invertible(op, x)
  Return Type: **bool** or a type convertible to **bool**.
  inverse(op, x)
  Return Type: Element or a type convertible to Element.

Remark: The contrast to Inversion, we request here that the result of inverse must be convertible to Element.

Invariants

- Cancellation from left
  if (is_invertible(op, x)) $\longrightarrow$ op(inverse(op, x), x) = identity(op, x)
- Cancellation from right
  if (is_invertible(op, x)) $\longrightarrow$ op(x, inverse(op, x)) = identity(op, x)

Implementation in ConceptGCC

```
concept PartiallyInvertibleMonoid<typename Operation, typename Element>
  : Monoid<Operation, Element>,
    algebra::Inversion<Operation, Element>
{
    typename is_invertible_result_type;
    is_invertible_result_type is_invertible(Operation, Element);
    where std::Convertible<is_invertible_result_type, bool>;

    where std::Convertible<inverse_result_type, Element>;

    // Does it overwrites the axiom from algebra::Inversion
    axiom Inversion(Operation op, Element x)
    {
        // Only for invertible elements:
        if (is_invertible(op, x))
            op( x, inverse(op, x) ) == identity(op, x);
        if ( is_invertible(op, x) )
            op( inverse(op, x), x ) == identity(op, x);
    }
};
```

8.2.8 PartiallyInvertibleCommutativeMonoid

Refinement of

PartiallyInvertibleMonoid and CommutativeMonoid

Implementation in ConceptGCC

```
concept PartiallyInvertibleCommutativeMonoid<typename Operation, typename Element>
  : PartiallyInvertibleMonoid<Operation, Element>,
    CommutativeMonoid<Operation, Element>
{};
```

Models

- modN_t<n> with functor implementing $+$ or $*$, see Section 5.1.
- Contingent: (**int**, math::add<**int**>), for limitations see Section 3.3.2.
- Contingent: (**float**, math::add<**float**>) and (**float**, math::mult<**float**>), for limitations see Section 3.3.2.
- Contingent: (complex<**double**>, math::add<complex<**double**> >) and (complex<**double**>, math::mult<complex<**double**> >), same as **float**.

Notes

An interesting behavior provide integer values towards the invertibility of multiplication. The division of one by any value with magnitude larger than one returns zero but the inversion could be implement by means of a EUCLIDEAN algorithm. Then for every odd value exists a unique reciprocal, i.e. their product is one, see. Section 3.3.6.

8.2.9 Group

A *Group* provides the same operations and properties as PartiallyInvertible-Monoid except that each element must be invertible.

Refinement of

PartiallyInvertibleMonoid

Notation

{Operation, Element}	are types that build a model of SemiGroup.
op	is an object of type Operation.
x	is an object of type Element.

Invariants

- Cancellation from left
  op(inverse(op, x), x) = identity(op, x)
- Cancellation from right
  x, op(inverse(op, x)) = identity(op, x)

Implementation in ConceptGCC

```
concept Group<typename Operation, typename Element>
  : PartiallyInvertibleMonoid<Operation, Element>
{
    axiom Inversion(Operation op, Element x)
    {
        // In contrast to PartiallyInvertibleMonoid all elements must be invertible
        op( x, inverse(op, x) ) == identity(op, x);
        op( inverse(op, x), x ) == identity(op, x);
    }
};
```

8.2.10 AbelianGroup

An *Abelian Group* adds the commutativity to a Group

Refinement of

Group and PartiallyInvertibleCommutativeMonoid

Implementation in ConceptGCC

```
concept AbelianGroup<typename Operation, typename Element>
  : Group<Operation, Element>,
    PartiallyInvertibleCommutativeMonoid<Operation, Element>
{};
```

Models

- modN_t<n> with functor implementing $+$ or $*$, see Section 5.1.
- Contingent: (**int**, math::add<**int**>), for limitations see Section 3.3.2.
- Contingent: (**float**, math::add<**float**>) and (**float**, math::mult<**float**>), for limitations see Section 3.3.2.
- Contingent: (complex<**double**>, math::add<complex<**double**> >) and (complex<**double**>, math::mult<complex<**double**> >), same as **float**.

8.3 Concepts for Additive Algebraic Structures

Although algebraic structures with freely chooseable operations are applicable in a broader sense, almost all arithmetic algorithms are expressed in terms of operators. For this reason, we introduce the additive and multiplicative concepts, which are also more convenient regarding the syntax.

One of our most important design goals was the consistency between the pure algebraic and the operator-based concepts. As the later are special cases of the former, we tried to express them as concept refinements. This raised the problem how to refine a concept on one template type from a concept with two types. The default functors can be used for the second template type and also to specify the consistency between the concepts in the axioms.

8.3.1 AdditiveMagma

An *Additive Magma* is a set S of elements of type Element with an addition. The set must be closed under the addition

$$x, y \in S \quad \rightarrow \quad x + y \in S.$$

Refinement of

Magma<math::add<Element>, Element>

Associated Types

- Corresponding functor
  math::add

Notation

Element	is a type that models AdditiveMagma.
x, y	are objects of type Element.

Valid Expressions

- Addition
  x + y
  Return Type: Element or a type convertible to Element
  Semantics: Can be arbitrary if for all pairs of values results are returned

- Addition Assignment
  x += y
  Return Type: Element&
  Semantics: Equivalent to x = x + y.

Invariants

- Consistency with Magma
  math::add<Element>()(x, y) = x + y.
  Unless add is specialized for Element the default implementation implies this consistency.

Implementation in ConceptGCC

```
concept AdditiveMagma<typename Element>
  : Magma< math::add<Element>, Element >
{
    typename assign_result_type;
    assign_result_type operator+=(Element& x, Element y);

    // Operator + is by default defined with +=
    typename result_type;
    result_type operator+(Element& x, Element y);
    {
        Element tmp(x);
        return tmp += y;
    }

    // Type consistency with Magma
    where std::SameType< result_type,
                         Magma< math::add<Element>, Element >::result_type >;

    axiom Consistency(math::add<Element> op, Element x, Element y)
    {
        op(x, y) == x + y;
        // Might change later
        x + y == x += y;
    }
}
```

The equality in axioms is not the equivalence of two consecutive computations but of two *separate* computations. Therefore side effects in one evaluation does not influence the other one, which we used to characterize the consistency between + and +=. However, this axiom is subject to modification. In case that it will be allowed to use temporaries in axioms the declaration will be written more intuitively.

The concept cannot be defined as an auto refinement because the consistency of the results must be confirmed by the programmer.

8.3.2 AdditiveCommutativeMagma

This concept combines the notion of additive closure and commutativity. The motivation is the same as for CommutativeMagma that floating point numbers are strictly spoken not associative but commutative, see notes.

Auto-Refinement of

CommutativeMagma<math::add<Element>, Element> and AdditiveMagma.

Implementation in ConceptGCC

```
auto concept AdditiveCommutativeMagma<typename Element>
  : AdditiveMagma<Element>,
    CommutativeMagma< math::add<Element>, Element >
{};
```

Models

- **float** and all other standard floating point types, see notes.

Notes

The existence of this concept is related to the non-associativity of floating point numbers. However, this topic is more complex and we discuss it in more detail in Section 4.3.2.

8.3.3 AdditiveSemiGroup

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

Auto-Refinement of

SemiGroup<math::add<Element>, Element> and AdditiveMagma.

Implementation in ConceptGCC

```
auto concept AdditiveSemiGroup<typename Element>
  : AdditiveMagma<Element>,
    SemiGroup< math::add<Element>, Element >
{};
```

Models

- STL strings as concatenation is defined with +, see Section 10.1.

8.3.4 AdditiveCommutativeSemiGroup

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

Auto-Refinement of

CommutativeSemiGroup<math::add<Element>, Element>, AdditiveSemiGroup, and AdditiveCommutativeMagma.

Implementation in ConceptGCC

```
auto concept AdditiveCommutativeSemiGroup<typename Element>
  : AdditiveSemiGroup<Element>,
    AdditiveCommutativeMagma<Element>,
    CommutativeSemiGroup< math::add<Element>, Element >
{};
```

8.3.5 AdditiveMonoid

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

Auto-Refinement of

Monoid<math::add<Element>, Element> and AdditiveSemiGroup.

Implementation in ConceptGCC

```
auto concept AdditiveMonoid<typename Element>
  : AdditiveSemiGroup<Element>,
    Monoid< math::add<Element>, Element >
{};
```

8.3.6 AdditiveCommutativeMonoid

An *Additive Commutative Monoid* is a commutative monoid w.r.t. addition.

Auto-Refinement of

CommutativeMonoid<math::add<Element>, Element> AdditiveMonoid, and AdditiveCommutativeSemiGroup

Implementation in ConceptGCC

```
auto concept AdditiveCommutativeMonoid<typename Element>
  : AdditiveMonoid<Element>,
    AdditiveCommutativeSemiGroup<Element>,
    CommutativeMonoid< math::add<Element>, Element >
{};
```

Models

- **unsigned int** and all other unsigned integral types.

8.3.7 AdditivePartiallyInvertibleMonoid

A *Additive Partially Invertible Monoid* is a monoid with an inverse function written as an unary minus. The binary minus is formally only an abbreviation for the addition with an inverted value. In contrast to AdditiveGroup, not all elements need to be invertible.

Refinement of

PartiallyInvertibleMonoid<math::add<Element>, Element> and AdditiveMonoid

Notation

Element	is a type that models AdditiveMagma.
x, y	are objects of type Element.
op	an object of type math::add<Element>.

Valid Expressions

- Inverse
  −x
  Return Type: Element or a type convertible to Element
  Semantics: Inverse of x. Behavior is undefined if x is not invertible.

- Subtraction
  x − y
  Return Type: Element or a type convertible to Element
  Semantics: Equivalent to x + −y. Behavior is undefined if y is not invertible.

- Subtraction Assignment
  x −= y
  Return Type: Element or a type convertible to Element
  Semantics: Equivalent to x = x + −y.

Invariants

- Consistency with PartiallyInvertibleMonoid
  inverse(op, x) = −x if x is invertible.
  If defaults for add and inverse are used, consistency is implied for default implementation of unary minus in concept as well as for unary minus predefined for C++ arithmetic data types.

Implementation in ConceptGCC

```
concept AdditivePartiallyInvertibleMonoid<typename Element>
  : AdditiveMonoid<Element>,
    PartiallyInvertibleMonoid< math::add<Element>, Element >
```

```
{
    // Operator −, binary and unary
    where std::Subtractable<Element>;
    where Negatable<Element>;

    typename assign_result_type;
    assign_result_type operator−=(Element& x, Element y);

    // Operator − by default defined with −=
    typename result_type;
    result_type operator−(Element& x, Element y);
    {
        Element tmp(x);
        return tmp −= y;
    }

    typename unary_result_type;
    unary_result_type operator−(Element x);
    {
        return zero(x) − x;
    }

    axiom Consistency(math::add<Element> op, Element x, Element y);
    {
        // consistency between additive and pure algebraic concept
        if ( is_invertible(op, y) )
            op(x, inverse(op, y)) == x − y;
        if ( is_invertible(op, y) )
            inverse(op, y) == −y;

        // consistency between unary and binary −
        if ( is_invertible(op, x) )
            identity(op, x) − x == −x;

        // Might change later
        if ( is_invertible(op, y) )
            x − y == x −= y;
    }
};
```

Notes

This concept is mainly defined for completeness and for consistency with pure algebraic concepts. Most additive concepts that provide inversion are invertible for all elements. However, the invertibility check can be used to deal with boundary cases like the minimal value of a signed integer type, which is formally invertible but in a numerically inadequate way (it is the value itself).

8.3.8 AdditivePartiallyInvertibleCommutativeMonoid

Auto-Refinement of

PartiallyInvertibleCommutativeMonoid<math::add<Element>, Element>, AdditivePartiallyInvertibleMonoid, and AdditiveCommutativeMonoid

Implementation in ConceptGCC

```
auto concept AdditivePartiallyInvertibleCommutativeMonoid<typename Element>
  : AdditivePartiallyInvertibleMonoid<Element>,
    AdditiveCommutativeMonoid<Element>,
    PartiallyInvertibleCommutativeMonoid< math::add<Element>, Element >
{};
```

Notes

8.3.9 AdditiveGroup

An *Additive Group* is identical with an AdditivePartiallyInvertibleMonoid except that all elements must be invertible.

Auto-Refinement of

Group<math::add<Element>, Element> and AdditivePartiallyInvertibleMonoid

Invariants

- Consistency with Group
  math::inverse(op, x) = −x
  In contrast to AdditivePartiallyInvertibleMonoid, this must hold for all elements.

Implementation in ConceptGCC

```
auto concept AdditiveGroup<typename Element>
  : AdditivePartiallyInvertibleMonoid<Element>,
    Group< math::add<Element>, Element >
{};
```

8.3.10 AdditiveAbelianGroup

An *Additive Abelian Group* refines AdditiveGroup with commutativity.

Auto-Refinement of

AbelianGroup<math::add<Element>, Element>, AdditiveGroup, and AdditivePartiallyInvertibleCommutativeMonoid

Implementation in ConceptGCC

```
auto concept AdditiveAbelianGroup<typename Element>
  : AdditiveGroup<Element>,
    AdditiveCommutativeMonoid<Element>,
    AbelianGroup< math::add<Element>, Element >
{};
```

Models

- **int** and all other signed integral types.
- **double** and all other standard floating point types (rounding errors a side).
- std::complex<T> where T is an additive Abelian group.

Notes

Appropriately defined vector and matrix types providing operators must be models.

8.4 Concepts for Multiplicative Algebraic Structures

The multiplicative concepts are very similar to the additive concepts and can be mostly derived from the latter by appropriate replacement of names and operators. The main difference in the definitions is that no unary operator representing inversion exist. For convenience, we provide the complete set of multiplicative concepts. Although the families concepts are very similar, they differ noticeably in how arithmetic data types model them.

8.4.1 MultiplicativeMagma

A *Multiplicative Magma* is a set S of elements of type Element with a multiplication. The set must be closed under the multiplication

$$x, y \in S \quad \rightarrow \quad x * y \in S.$$

Refinement of

Magma<math::mult<Element>, Element>.

Associated Types

- Corresponding functor
  math::mult

Notation

Element	is a type that models MultiplicativeMagma.
x, y	are objects of type Element.

Valid Expressions

- Multiplication
  x ∗ y

Return Type:	Element or a type convertible to Element
Semantics:	Can be arbitrary if for all pairs of values results are returned

- Multiplication Assignment
  x ∗= y

Return Type:	Element&
Semantics:	Equivalent to x = x ∗ y.

Invariants

- Consistency with Magma
  math::mult<Element>()(x, y) = x * y.
  Unless mult is specialized for Element the default implementation implies this consistency.

Implementation in ConceptGCC

```
concept MultiplicativeMagma<typename Element>
  : Magma< math::mult<Element>, Element >
{
    typename assign_result_type;
    assign_result_type operator*=(Element& x, Element y);

    // Operator * is by default defined with *=
    typename result_type;
    result_type operator*(Element& x, Element y);
    {
        Element tmp(x);
        return tmp *= y;
    }

    // Type consistency with Magma
    where std::SameType< result_type,
                         Magma< math::mult<Element>, Element >::result_type >;

    axiom Consistency(math::mult<Element> op, Element x, Element y)
    {
        op(x, y) == x * y;
        // Might change later
        x * y == x *= y;
    }
}
```

8.4.2 MultiplicativeCommutativeMagma

This concept combines the notion of multiplicative closure and commutativity. The motivation is the same as for CommutativeMagma that floating point numbers are strictly spoken not associative but commutative, but see Section 4.3.2.

Auto-Refinement of

CommutativeMagma<math::mult<Element>, Element> and MultiplicativeMagma.

Implementation in ConceptGCC

```
auto concept MultiplicativeCommutativeMagma<typename Element>
  : MultiplicativeMagma<Element>,
    CommutativeMagma< math::mult<Element>, Element >
{};
```

Models

- **float** and all other standard floating point types, see notes of AdditiveCommutativeMagma.

8.4.3 MultiplicativeSemiGroup

A *Multiplicative Semi-Group* is an multiplicative magma where the multiplication is associative.

Auto-Refinement of

SemiGroup<math::mult<Element>, Element> and MultiplicativeMagma.

Implementation in ConceptGCC

```
auto concept MultiplicativeSemiGroup<typename Element>
  : MultiplicativeMagma<Element>,
    SemiGroup< math::mult<Element>, Element >
{};
```

8.4.4 MultiplicativeCommutativeSemiGroup

A *Multiplicative Semi-Group* is an multiplicative magma where the multiplication is associative.

Auto-Refinement of

CommutativeSemiGroup<math::mult<Element>, Element>, MultiplicativeSemiGroup, and MultiplicativeCommutativeMagma.

Implementation in ConceptGCC

```
auto concept MultiplicativeCommutativeSemiGroup<typename Element>
  : MultiplicativeSemiGroup<Element>,
    MultiplicativeCommutativeMagma<Element>,
    CommutativeSemiGroup< math::mult<Element>, Element >
{};
```

8.4.5 MultiplicativeMonoid

A *Multiplicative Monoid* is an multiplicative semi-group with an identity.

Auto-Refinement of

Monoid<math::mult<Element>, Element> and MultiplicativeSemiGroup.

Implementation in ConceptGCC

```
auto concept MultiplicativeMonoid<typename Element>
  : MultiplicativeSemiGroup<Element>,
    Monoid< math::mult<Element>, Element >
{};
```

8.4.6 MultiplicativeCommutativeMonoid

A *Multiplicative Commutative Monoid* is a commutative monoid w.r.t. multiplication.

Auto-Refinement of

CommutativeMonoid<math::mult<Element>, Element> MultiplicativeMonoid, and MultiplicativeCommutativeSemiGroup.

Implementation in ConceptGCC

```
auto concept MultiplicativeCommutativeMonoid<typename Element>
  : MultiplicativeMonoid<Element>,
    MultiplicativeCommutativeSemiGroup<Element>,
    CommutativeMonoid< math::mult<Element>, Element >
{};
```

Models

- **int** and all other integral types.
- Cyclic groups, see Section 5.

8.4.7 MultiplicativePartiallyInvertibleMonoid

A *Multiplicative Partially Invertible Monoid* is a monoid with an inverse function. In contrast to MultiplicativeGroup, not all elements need to be invertible.

Refinement of

PartiallyInvertibleMonoid<math::mult<Element>, Element> and MultiplicativeMonoid.

Notation

Element	is a type that models MultiplicativeMagma.
x, y	are objects of type Element.
op	an object of type math::mult<Element>.

Valid Expressions

- Division
  x / y

Return Type:	Element or a type convertible to Element
Semantics:	Equivalent to x ∗ inverse(op, y). Behavior is undefined if y is not invertible.

- Division Assignment
  x /= y

Return Type:	Element or a type convertible to Element
Semantics:	Equivalent to x = x / y.

Implementation in ConceptGCC

```
concept MultiplicativePartiallyInvertibleMonoid<typename Element>
  : MultiplicativeMonoid<Element>,
    PartiallyInvertibleMonoid< math::mult<Element>, Element >
{
    typename assign_result_type;
    assign_result_type operator/=(Element& x, Element y);

    // Operator / by default defined with /=
    typename result_type;
    result_type operator/(Element& x, Element y);
    {
        Element tmp(x);
        return tmp /= y;
    }

    axiom Consistency(math::mult<Element> op, Element x, Element y);
    {
        // consistency between multiplicative and pure algebraic concept
        if ( is_invertible(op, y) )
            op(x, inverse(op, y)) == x / y;

        // Consistency between / and /=, might change later
        if ( is_invertible(op, y) )
            x / y == x /= y;
    }
};
```

Models

- Cyclic groups, see. Section 5.
- Square matrices where **operator**$*$ realizes matrix product.
- Rectangular matrices where **operator**$*$ provides element-wise product.[1]

Notes

Typically, all elements except one will be invertible and the test is_invertible will check whether the value is non-zero, i.e. the identity of the corresponding addition. This is therefore the default implementation which should be specialized if the behavior of some type differs from that, e.g. for matrices.

[1]To avoid confusion with the canonical matrix product, we suggest to not implement this product in terms of **operator**$*$. Mathematical tools like Mathematica provide two multiplication operators to deal with this ambiguity. Whether the matrix product should use an operator is a decision to be made on the context. Furthermore, operators can be defined either way within the context of a specific toolbox.

8.4.8 MultiplicativePartiallyInvertibleCommutativeMonoid

Auto-Refinement of

PartiallyInvertibleCommutativeMonoid<math::mult<Element>, Element>, MultiplicativePartiallyInvertibleMonoid, and MultiplicativeCommutativeMonoid.

Implementation in ConceptGCC

```
auto concept MultiplicativePartiallyInvertibleCommutativeMonoid<typename Element>
  : MultiplicativePartiallyInvertibleMonoid<Element>,
    MultiplicativeCommutativeMonoid<Element>,
    PartiallyInvertibleCommutativeMonoid< math::mult<Element>, Element >
{};
```

Notes

Floating point numbers provide a rather symmetric range of exponents so that most values are invertible. Nevertheless, some very large values are not invertible, see. Section 3.4.3.

8.4.9 MultiplicativeGroup

An *Multiplicative Group* is identical with an MultiplicativePartiallyInvertibleMonoid except that all elements must be invertible.

Auto-Refinement of

Group<math::mult<Element>, Element> and MultiplicativePartiallyInvertibleMonoid.

Implementation in ConceptGCC

```
auto concept MultiplicativeGroup<typename Element>
  : MultiplicativePartiallyInvertibleMonoid<Element>,
    Group< math::mult<Element>, Element >
{};
```

Models

Most arithmetic data types will not model MultiplicativeGroup because at least the identity of the corresponding addition will not be invertible. However, one can provide user-defined data types with an **operator**∗, see. Section 3.4.3.

8.4.10 MultiplicativeAbelianGroup

An *Multiplicative Abelian Group* refines MultiplicativeGroup with commutativity.

Auto-Refinement of

AbelianGroup<math::mult<Element>, Element>, MultiplicativeGroup, and MultiplicativePartiallyInvertibleCommutativeMonoid.

Implementation in ConceptGCC

```
auto concept MultiplicativeAbelianGroup<typename Element>
  : MultiplicativeGroup<Element>,
    MultiplicativeCommutativeMonoid<Element>,
    AbelianGroup< math::mult<Element>, Element >
{};
```

Notes

No standard arithmetic type models MultiplicativeAbelianGroup because at least the identity of the corresponding addition will not be invertible. However, one can provide user-defined data types with an **operator**∗, see. Section 3.4.3.

Chapter 9

Algebraic Structures with Two Operations

The concepts introduced in this section characterize set with two operations defined upon. In most numerical software this will be addition and multiplication. However, other pairs of operations perfectly model these concepts and algorithms exist that can be expressed using ring concepts over minimum and maximum function as we will show later in this section.

Given the need for arbitrary operations we provide generic concepts which we later refine to concepts defined on addition and multiplication. As we consider the latter by far more important we use shorter names — like Ring or Field — and longer names — as GenericRing — otherwise.

9.1 Purely Algebraic Concepts

In namespace algebra.

9.1.1 Distributive

Notation

{AddOp, MultOp, Element}	are types that build a model of GenericRing.
add	is an object of type AddOp.
mult	is an object of type MultOp.
x, y, z	are objects of type Element.

Valid Expressions

None in addition to the expressions defined in AbelianGroup<math::add<Element>, Element> and SemiGroup<math::mult<Element>, Element>.

Invariants

- Distributivity from left
  mult(x, add(y, z)) = add(mult(x, y), mult(x, z))
- Distributivity from right
  mult(add(x, y), z) = add(mult(x, z), mult(y, z))

Implementation in ConceptGCC

```
concept Distributive<typename AddOp, typename MultOp, typename Element>
{
    axiom Distributivity(AddOp add, MultOp mult, Element x, Element y, Element z)
    {
        // From left
        mult(x, add(y, z)) == add(mult(x, y), mult(x, z));
        // z right
        mult(add(x, y), z) == add(mult(x, z), mult(y, z));
    }
};
```

9.1.2 algebra::Ring

Auto-Refinement of

AbelianGroup<AddOp, Element>, SemiGroup<MultOp, Element>, and Distributive.

Implementation in ConceptGCC

```
auto concept Ring<typename AddOp, typename MultOp, typename Element>
  : AbelianGroup<AddOp, Element>,
    SemiGroup<MultOp, Element>,
    Distributive<AddOp, MultOp, Element>
{};
```

9.1.3 algebra::RingWithIdentity

Auto-Refinement of

algebra::Ring and Monoid<MultOp, Element>.

Implementation in ConceptGCC

```
auto concept RingWithIdentity<typename AddOp, typename MultOp, typename Element>
  : Ring<AddOp, MultOp, Element>,
    Monoid<MultOp, Element>
{};
```

9.1.4 algebra::DivisionRing

Refinement of

algebra::RingWithIdentity and Inversion<MultOp, Element>.

Notation

{AddOp, MultOp, Element}	are types that build a model of GenericDivisionRing.
add	is an object of type AddOp.
mult	is an object of type MultOp.
x	is an objects of type Element.

Invariants

- Zero is different from one
  identity(add, x) $\neq$ identity(mult, x)
- Non-zero divisibility from left
  if (x $\neq$ identity(add, x))
  mult(inverse(mult, x), x) = identity(mult, x)
- Non-zero divisibility from right
  if (x $\neq$ identity(add, x))
  mult(x, inverse(mult, x)) = identity(mult, x)

Implementation in ConceptGCC

```
concept DivisionRing<typename AddOp, typename MultOp, typename Element>
  : RingWithIdentity<AddOp, MultOp, Element>,
    Inversion<MultOp, Element>
{
    // 0 != 1, otherwise trivial
    axiom ZeroIsDifferentFromOne(AddOp add, MultOp mult, Element x)
    {
        identity(add, x) != identity(mult, x);
    }

    // Non-zero divisibility from left and from right
    axiom NonZeroDivisibility(AddOp add, MultOp mult, Element x)
    {
        if (x != identity(add, x))
            mult(inverse(mult, x), x) == identity(mult, x);
        if (x != identity(add, x))
            mult(x, inverse(mult, x)) == identity(mult, x);
    }
};
```

9.1.5 algebra::SkewField

SkewField is defined as synonym for algebra::DivisionRing.

Auto-Refinement of

algebra::DivisionRing.

Implementation in ConceptGCC

```
auto concept SkewField<typename AddOp, typename MultOp, typename Element>
  : DivisionRing<AddOp, MultOp, Element>
{};
```

9.1.6 algebra::Field

Auto-Refinement of

algebra::DivisionRing and Commutative<MultOp, Element>.

Implementation in ConceptGCC

```
auto concept Field<typename AddOp, typename MultOp, typename Element>
  : DivisionRing<AddOp, MultOp, Element>,
    Commutative<MultOp, Element>
{};
```

9.2 Augmented Concepts

9.2.1 GenericRing

A *Ring* is a set upon which two operations are defined, one of them building an *Abelian group* and the other one forming a *semi-group*, see notes. Furthermore the second operation is distributive over the first one.

Refinement of

AbelianGroup<math::add<Element>, Element> and SemiGroup<math::mult<Element>, Element>

Notation

{AddOp, MultOp, Element}	are types that build a model of GenericRing.
add	is an object of type AddOp.
mult	is an object of type MultOp.
x, y, z	are objects of type Element.

Valid Expressions

None in addition to the expressions defined in AbelianGroup<math::add<Element>, Element> and SemiGroup<math::mult<Element>, Element>.

Invariants

- Distributivity from left
  mult(x, add(y, z)) = add(mult(x, y), mult(x, z))
- Distributivity from right
  mult(add(x, y), z) = add(mult(x, z), mult(y, z))

Implementation in ConceptGCC

```
concept GenericRing<typename AddOp, typename MultOp, typename Element>
  : AbelianGroup<AddOp, Element>,
    SemiGroup<MultOp, Element>
{
    axiom Distributivity(AddOp add, MultOp mult, Element x, Element y, Element z)
    {
        // From left
        mult(x, add(y, z)) == add(mult(x, y), mult(x, z));
        // From right
        mult(add(x, y), z) == add(mult(x, z), mult(y, z));
    }
};
```

Notes

Another definition of ring requires the second operation to be a monoid whereby calling a structure with a semi-group pseudo-ring. We choose this terminology amongst other reasons to be consistent with already existing formal concept definitions like Tecton [10].

9.2.2 GenericCommutativeRing

A *Commutative Ring* adds the notion of commutativity to the concept of a ring.

Refinement of

GenericRing<math::add<Element>, math::mult<Element>, Element> and CommutativeSemiGroup<math::mult<Element>, Element>

Implementation in ConceptGCC

```
concept GenericCommutativeRing<typename AddOp, typename MultOp, typename Element>
  : GenericRing<AddOp, MultOp, Element>,
    CommutativeSemiGroup<MultOp, Element>
{};
```

9.2.3 GenericRingWithIdentity

A *Ring with Identity* introduces a neutral element w.r.t. multiplication into the concept of a ring.

Refinement of

GenericRing<math::add<Element>, math::mult<Element>, Element> and Monoid<math::mult<Element>, Element>

Implementation in ConceptGCC

```
auto concept GenericRingWithIdentity<typename AddOp, typename MultOp, typename Element>
   : GenericRing<AddOp, MultOp, Element>,
     Monoid<MultOp, Element>,
     algebra::RingWithIdentity<AddOp, MultOp, Element>
{};
```

Models

- Square matrices with matrix product and addition.

9.2.4 GenericCommutativeRingWithIdentity

A *Commutative Ring with Identity* adds adds the notion of commutativity w.r.t. multiplication to the concept of a ring with identity.

Refinement of

GenericRingWithIdentity<math::add<Element>, math::mult<Element>, Element> and GenericCommutativeRing<math::add<Element>, math::mult<Element>, Element>

Implementation in ConceptGCC

```
auto concept GenericCommutativeRingWithIdentity<typename AddOp, typename MultOp, typename Element>
  : GenericRingWithIdentity<AddOp, MultOp, Element>,
    GenericCommutativeRing<AddOp, MultOp, Element>,
    CommutativeMonoid<MultOp, Element>
{};
```

Models

- Rectangular matrices with matrix addition with element-wise multiplication if the product of the elements commutes, see. MultiplicativePartially-InvertibleMonoid.

9.2.5 GenericDivisionRing

A *Division Ring* is a GenericRingWithIdentity where 0 is different from 1 and each non-zero element has a reciprocal element, see. notes on inverse of zero. Hence, a is a AdditiveAbelianGroup and a MultiplicativePartiallyInvertibleMonoid where only 0 has no reciprocal. This implies that if the set S is a division ring then $S \setminus \{0\}$ is an MultiplicativeGroup whereby the exclusion on an element is not expressible in the programming language.

Refinement of

GenericRingWithIdentity<math::add<Element>, math::mult<Element>, Element>

Notation

{AddOp, MultOp, Element}	are types that build a model of GenericDivisionRing.
add	is an object of type AddOp.
mult	is an object of type MultOp.
x	is an objects of type Element.

Valid Expressions

In addition to the concepts of GenericRingWithIdentity

- Inversion
  inverse(x)
  Return Type: Element or a type convertible to Element

Invariants

- Zero is different from one
  identity(add, x) $\neq$ identity(mult, x)
- Non-zero divisibility from left
  if (x $\neq$ identity(add, x))
  mult(inverse(mult, x), x) = identity(mult, x)
- Non-zero divisibility from right
  if (x $\neq$ identity(add, x))
  mult(x, inverse(mult, x)) = identity(mult, x)

Implementation in ConceptGCC

```
concept GenericDivisionRing<typename AddOp, typename MultOp, typename Element>
  : GenericRingWithIdentity<AddOp, MultOp, Element>
{
    typename inverse_result_type;
```

```
        inverse_result_type inverse(MultOp, Element);
        where std::Convertible<inverse_result_type, Element>;

        axiom ZeroIsDifferentFromOne(AddOp add, MultOp mult, Element x)
        {
            identity(add, x) != identity(mult, x);
        }

        axiom NonZeroDivisibility(AddOp add, MultOp mult, Element x)
        {
            if (x != identity(add, x))
                mult(inverse(mult, x), x) == identity(mult, x);
            if (x != identity(add, x))
                mult(x, inverse(mult, x)) == identity(mult, x);
        }
    };
```

Notes

The set $\{0\}$ with $0+0=0$ and $0*0=0$ fulfills all requirements of a division ring and Field. As this structure does not provide any practical usage we exclude it from our considerations.

Allowing $0=1$ in some set implies that this set is $\{0\}$. For this reason we require $0 \neq 1$ in the description.

Instead of introducing the inversion in this concept as valid expression one can refine from PartiallyInvertibleMonoid<math::mult<Element>, Element>. This would additionally require the definition of a function is_invertible which introduces some overhead as it is given that all non-zero elements are invertible. Refining from Group<math::mult<Element>, Element> would be wrong because zero is not invertible.

9.2.6 GenericField

Commutative division rings are *Fields*.

Refinement of

GenericDivisionRing<math::add<Element>, math::mult<Element>, Element> and GenericCommutativeRingWithIdentity<math::add<Element>, math::mult<Element>, Element>

Implementation in ConceptGCC

```
concept GenericField<typename AddOp, typename MultOp, typename Element>
  : GenericDivisionRing<AddOp, MultOp, Element>,
    GenericCommutativeRingWithIdentity<AddOp, MultOp, Element>
{};
```

9.3 Operator-Based Concepts

9.3.1 Operator-Based Concepts

The operator-based concepts have the same requirements as their generic equivalents. Therefore, they can be expressed as refinements of generic two-operation concepts and one-operation concepts with operators, only the non-zero invertibility is repeated using a short cut.

9.3.2 Ring

Auto-Refinement of

AdditiveAbelianGroup, MultiplicativeSemiGroup, and GenericRing<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept Ring<typename Element>
  : AdditiveAbelianGroup<Element>,
    MultiplicativeSemiGroup<Element>,
    GenericRing<math::add<Element>, math::mult<Element>, Element>
{};
```

9.3.3 CommutativeRing

Auto-Refinement of

Ring, MultiplicativeCommutativeSemiGroup, and GenericCommutativeRing<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept CommutativeRing<typename Element>
  : Ring<Element>,
    MultiplicativeCommutativeSemiGroup<Element>,
    GenericCommutativeRing<math::add<Element>, math::mult<Element>, Element>
{};
```

9.3.4 RingWithIdentity

Auto-Refinement of

Ring, MultiplicativeMonoid, and GenericRingWithIdentity<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept RingWithIdentity<typename Element>
  : Ring<Element>,
    MultiplicativeMonoid<Element>,
    GenericRingWithIdentity<math::add<Element>, math::mult<Element>, Element>
{};
```

9.3.5 CommutativeRingWithIdentity

Auto-Refinement of

RingWithIdentity, CommutativeRing, and GenericCommutativeRingWithIdentity<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept CommutativeRingWithIdentity<typename Element>
  : RingWithIdentity<Element>,
    CommutativeRing<Element>,
    GenericCommutativeRingWithIdentity<math::add<Element>, math::mult<Element>, Element>
{};
```

Models

- **int** and all other signed integer types.
- Cyclic groups, see. Section 5.

9.3.6 DivisionRing

Auto-Refinement of

RingWithIdentity, MultiplicativePartiallyInvertibleMonoid, and GenericDivisionRing<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept DivisionRing<typename Element>
  : RingWithIdentity<Element>,
    MultiplicativePartiallyInvertibleMonoid<Element>,
    GenericDivisionRing<math::add<Element>, math::mult<Element>, Element>
{
    axiom NonZeroDivisibility(Element x)
    {
        if (x != zero(x))
            x / x == one(x);
    }
};
```

9.3.7 Field

Auto-Refinement of

DivisionRing, CommutativeRingWithIdentity, and GenericField<math::add<Element>, math::mult<Element>, Element>.

Implementation in ConceptGCC

```
concept Field<typename Element>
  : DivisionRing<Element>,
    CommutativeRingWithIdentity<Element>,
    GenericField<math::add<Element>, math::mult<Element>, Element>
{};
```

Models

- **float** and all other floating point types rounding errors and closure issues aside.
- std::complex<**double**>
- Cyclic groups where the size is prime, see. Section 5.

Chapter 10

Models

10.1 String Concatenation

STL strings are concatenated using a + operator. The concatenation is associative and has an identity element. Unfortunately, the default of identity is not only incorrect for this operation but also can cause the program to crash (interpreting the 0 as pointer in the string constructor). Therefor, it is crucial to define a specialized identity.

```
namespace math {
    template<>
    struct identity_t< math::add<string>, string >
    {
        string operator()(const math::add<string>&, const string&) const
        {
            return string();
        }
    } ;

    concept_map Monoid< math::add<string>, string > {}
}
```

10.2 Non-Negative Real Values

. As an example of a simple monoid we introduce a class that only allows positive real numbers (and 0). The concept map for AdditiveCommutativeMonoid implies amongst others the model declaration of (math::add<positive_real>, positive_real) as CommutativeMonoid.

```
class positive_real
{
protected:
    double value;
public:
    positive_real(double m): value(m)
    {
        if (m < 0.0) throw "Negative value not allowed!\n";
    }

    double get_value() const
    {
        return value;
    }

    positive_real operator+(positive_real const& y) const
    {
        return value + y.value;
    }

    /* .... */
};

namespace math {
    concept_map AdditiveCommutativeMonoid<mtl::positive_real> {};
}
```

Bibliography

[1] Jean-Camille Birget, Stuart Margolis, John Meakin, and Mark Sapir, editors. *Algorithmic Problems in Groups and Semigroups.* Birkhäuser, Boston, 2000.

[2] Rolf Bonderer. ConceptC++ implementation of iterative solvers. Master's thesis, ETH Zürich, 2006.

[3] Peter Gottschling. Angel – an extensible library for Jacobian accumulation. In *Proceedings of the 4th International Conference on Automatic Differentiation, Chicago*, July 19th–23rd 2004.

[4] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06).* ACM Press, October 2006. Accepted. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution.

[5] Douglas Gregor and Jeremy Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[6] Douglas Paul Gregor. The ConceptGCC home page. `http://www.generic-programming.org/software/ConceptGCC`.

[7] Karsten Henckell and Jean-Eric Pin. *Ordered Monoids in J-Trivial*, pages 121–137. In Birget et al. [1], 2000.

[8] William George Horner. A new method of solving numerical equations of alll orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308–335, July 1819.

[9] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, Reading, 1996.

[10] David R. Musser, Sibylle Schupp, Christoph Schwarzweller, and Rüdiger Loos. The Tecton concept library. Technical report, Fakultät für Informatik, Universität Tübingen, 1999.

[11] David R. Musser and Alexander A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[12] Jean-François Raymond, Pascal Tesson, and Denis Thérien. *Multiparty Communication Complexity of Finite Monoids*, pages 217–233. In Birget et al. [1], 2000.

[13] Jeremy Siek. *Boost Concept Check Library*. Boost, 2000. `http://www.boost.org/libs/concept_check/`.

[14] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.

[15] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: A unifying framework for numerical linear algebra. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.

[17] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[18] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. `http://www.sgi.com/tech/stl/`.

[19] Alexander A. Stepanov, Jim Dehnert, and John Wilkinson. Generic programming: Arithmetic, 1998. `http://www.stepanovpapers.com/gprog/arith.html`.

[20] Bjarne Stroustrup and Gabriel Dos Reis. A concept design. C++ Extensions reflector message c++std-ext-7073, April 2005.

[21] Bartel Leendert van der Waerden. *Algebra, Volume I and II*. Springer, 1990.