Toward a More Complete Taxonomy of Algebraic Properties for Numeric Libraries in TR2

Document number: N2650 08-0160
Authors: Peter Gottschling, Technische Universität Dresden
Walter E. Brown, Fermi National Accelerator Laboratory
Date: 2008-05-16
Project: Programming Language C++, Library Working Group
Reply to: Peter.Gottschling@tu-dresden.de
wb@fnal.gov

1 Introduction

We propose in this document mathematical concepts that allow compilers to

- Verify the semantically correct usage of generic functions and
- Select optimal algorithms according to semantic properties of operations.

The concepts presented here extend the algebraic characterizations from N2645.

Although the concepts of this document are not used in the context of current standard libraries they are fundamental for the definition of *Every* numeric generic library. All sophisticated mathematical domains and entities — like SobolevSpace, DifferentialOperator, or FiniteElement — are defined upon these fundamental algebraic properties. Such concepts will allow for an entirely new era of scientific programming with verified mathematical properties. For the sake of behavioral consistency of advanced numeric software a careful design and immaculate consistency of these fundamental concepts is paramount. Only if the essential mathematical properties are defined and used consistently in different libraries, the integration of numeric software packages is *semantically sound*. The best way to establish consistency in the mathematical behavior is to standardize semantic concepts.

2 Synopsis

```
namespace std {

  // Basic mathematical concepts
  auto concept AbelianGroup<typename Operation, typename Element>;

  // Additive concepts
  concept Additive<typename Element>;
```

```
  auto concept AdditiveCommutative<typename Element>;
  auto concept AdditiveSemiGroup<typename Element>;
  concept AdditiveMonoid<typename Element>;
  concept AdditivePIMonoid<typename Element>;
  auto concept AdditiveGroup<typename Element>;
  auto concept AdditiveAbelianGroup<typename Element>;

  // Multiplicative concepts
  concept Multiplicative<typename Element>;
  auto concept MultiplicativeCommutative<typename Element>;
  auto concept MultiplicativeSemiGroup<typename Element>;
  concept MultiplicativeMonoid<typename Element>;
  concept MultiplicativePIMonoid<typename Element>;
  auto concept MultiplicativeGroup<typename Element>;
  auto concept MultiplicativeAbelianGroup<typename Element>;

  // Generic two operation concepts
  concept Distributive<typename AddOp, typename MultOp, typename Element>;
  auto concept Ring<typename AddOp, typename MultOp, typename Element>;
  auto concept RingWithIdentity<typename AddOp, typename MultOp, typename Element>;
  concept DivisionRing<typename AddOp, typename MultOp, typename Element>;
  auto concept Field<typename AddOp, typename MultOp, typename Element>;

  // Concepts for two operators
  auto concept OperatorRing<typename Element>;
  auto concept OperatorRingWithIdentity<typename Element>;
  auto concept OperatorDivisionRing<typename Element>;
  auto concept OperatorField<typename Element>;
}
```

3 Concepts [concept.math]

We begin this section with the remaining concept definition for arbitrary binary operations. Then we specialize it together with the concepts of N2645 to addition and multiplication. Subsequently we will introduce concepts for two binary operation and again specialize them to addition and multiplication.

3.1 Basic Concepts for Binary Operations [concept.math.basic]

3.1.1 Abelian Group [concept.math.abel]

To complete the definitions from Nxxx, an AbelianGroup combines commutativity with the group requirements:

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

3.2 Additive Concepts [concept.add]

In the following section we specialize the functor-based concepts to addition. The operator-based concepts are more convenient to use and more appropriate for numerical software. This specializa-

tion can be expressed as concept refinement by involving an add functor. The incorporation of a default functor allows for the refinement of a two-argument concept by a one-argument concept.

3.2.1 Consistency of the Additive Concepts [concept.add.additive]

The concept Additive specifies the consistency between the functor and operator-based computations:

```
concept Additive<typename Element>
  : HasPlus<Element>
{
    typename plus_assign_result_type;
    plus_assign_result_type operator+=(Element& x, Element y) {
        x= x + y; return x;
    }
    requires std::Convertible<plus_assign_result_type, Element&>;

    axiom Consistency(add<Element> op, Element x, Element y) {
        op(x, y) == x + y;
        op(x, y) == (x += y, x);
    }
}
```

3.2.2 Additive Commutativity [concept.add.commutative]

The commutativity of the + operator is implied by the consistency ([concept.add.additive]) of the operator and the commutativity of the functor ([concept.math.commutative]):

```
auto concept AdditiveCommutative<typename Element>
  : Additive<Element>,
    Commutative< math::add<Element>, Element >
{}
```

3.2.3 Additive Semi-group [concept.add.semigroup]

The concept:

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

defines the associativity for **operator**+. It is implied by the associativity of the functor add.

3.2.4 Additive Monoid [concept.add.monoid]

The AdditiveMonoid introduces the shortcut zero for the identity element:

```
concept AdditiveMonoid<typename Element>
  : AdditiveSemiGroup<Element>, Monoid< math::add<Element>, Element >
{
    Element zero(Element x) {
        return identity(math::add<Element>(), x);
    }
```

```
    axiom IdentityConsistency (math::add<Element> op, Element x) {
        zero(x) == identity(op, x);
    }
};
```

If the function zero is not provided for the type, the default implementation uses the identity element from the add functor. Otherwise the axiom demands that this consistency is given.

Please note that zero can be different from Element(0). This definition can cause crashes when concatenation of std::string is treated as AdditiveMonoid).

3.2.5 Additive Partially Invertible Monoid [concept.add.pimonoid]

Inversion is for addition given by the unary -. Mathematically spoken, the binary - is only an abbreviation for x + -y. These operators and their consistency rules are defined in concept AdditivePIMonoid:

```
concept AdditivePIMonoid<typename Element>
  : std::HasMinus<Element>, AdditiveMonoid<Element>,
    PIMonoid< math::add<Element>, Element >
{
    typename minus_assign_result_type;
    minus_assign_result_type operator-=(Element& x, Element y) {
        x= x -y; return x;
    }
    requires std::Convertible<minus_assign_result_type, Element&>;

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

    axiom InverseConsistency(math::add<Element> op, Element x, Element y) {
        if (is_invertible(op, y))
            op(x, inverse(op, y)) == x -y;
        if (is_invertible(op, y))
            op(x, y) == (x -= y, x);
        if (is_invertible(op, y))
            inverse(op, y) == -y;
        if (is_invertible(op, x))
            identity(op, x) -x == -x;
    }
}
```

Discussion: Under normal circumstances, the addition is always invertible. A pathological counter-example could be the declaration of unsigned integers as AdditivePIMonoid where only the 0 is invertible (we rather consider unsigned as non-invertible). For signed integers we can exclude the smallest integers whos inverse is computed incorrectly in the two-complement.

More important reasons for this concept are the symmetry with other concept categories and the fact that it lies entirely in the hand of the users how **operator**+ behaves.

3.2.6 Additive Group [concept.add.group]

The concept AdditiveGroup applies the group properties to the + operator:

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

3.2.7 Additive Abelian Group [concept.add.abel]

The concept AdditiveAbelianGroup characterizes a commutative AdditiveGroup:

```
auto concept AdditiveAbelianGroup<typename Element>
  : AdditiveGroup<Element>, AdditiveCommutative<Element> {}
```

3.3 Multiplicative Concepts [concept.mult]

Analogously to the additive concepts, we define multiplicative concepts for the operators $*$ and /.

3.3.1 Consistency of the Multiplicative Concepts [concept.mult.multiplicative]

The following concept specifies the consistency between the functor and operator-based computations:

```
concept Multiplicative<typename Element>
  : std::HasMultiply<Element>
{
    typename times_assign_result_type;
    times_assign_result_type operator*=(Element& x, Element y) {
        x= x * y; return x;
    }
    requires std::Convertible<times_assign_result_type, Element&>;

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

3.3.2 Multiplicative Commutativity [concept.mult.commutative]

The commutativity of the + operator is implied by the consistency ([concept.mult.multiplicative]) of the operator and the commutativity of the functor ([concept.math.commutative]):

```
auto concept MultiplicativeCommutative<typename Element>
  : Multiplicative<Element>,
    Commutative< math::mult<Element>, Element >
{}
```

3.3.3 Multiplictive Semi-group [concept.mult.semigroup]

The concept:

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

defines the associativity for **operator**$*$. It is implied by the associativity of the functor mult.

3.3.4 Multiplicative Monoid [concept.mult.monoid]

The MultiplicativeMonoid introduces the shortcut one for the identity element:

```
concept MultiplicativeMonoid<typename Element>
  : MultiplicativeSemiGroup<Element>, Monoid< math::mult<Element>, Element >
{
    Element one(Element x) {
        return identity(math::mult<Element>(), x);
    }
    axiom IdentityConsistency (math::math::mult<Element> op, Element x) {
        one(x) == identity(op, x);
    }
};
```

3.3.5 Multiplicative Partially Invertible Monoid [concept.mult.pimonoid]

This concept introduces the inversion in terms of an operator. In contrast to the addition, there exist no operator to represent the inversion itself. The binary / operator defines (analogously to the binary -) the combination of multiplication with the reciprocal.

```
concept MultiplicativePIMonoid<typename Element>
  : std::HasDivide<Element>, MultiplicativeMonoid<Element>,
    PIMonoid< math::mult<Element>, Element >
{
    typename divide_assign_result_type;
    divide_assign_result_type operator/=(Element& x, Element y) {
        x= x / y; return x;
    }
    requires std::Convertible<divide_assign_result_type, Element&>;

    axiom InverseConsistency(math::mult<Element> op, Element x, Element y) {
        if (is_invertible(op, y))
            op(x, inverse(op, y)) == x / y;
        if (is_invertible(op, y))
            op(x, y) == (x /= y, x);
    }
}
```

Typically one element at least is not invertible: 0 (or more generaly speaking the identity element of the addition). Examples with multiple non-invertible elements are given in [concept.math.pimonoid] (N2645).

Remark: for some types the test for invertibility is as expensive as the inversion itself, e.g. square matrices. Once we know that a matrix is invertible we already know its inverse. We recommand specialized implementations for such types in high-performance applications. A potential solution can be a function called maybe_inverse that returns a std::pair of **bool** and Element. If the boolean is false the value is undefined (and does not matter), otherwise it contains the inverse. (Unfortunately, it will be more difficult with this approach to handle the pair as r-value and to avoid expensively copying the inverse element.)

3.3.6 Multiplicative Group [concept.mult.group]

The concept MultiplicativeGroup applies the group properties to the $*$ operator:

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

As mentioned before, at least one element typically has no reciprocal. Thus, this concept is usually not modeled under rigorous criteria. Two reasons nevertheless suggest the presence of this concepts:

- Symmetry to the other concept categories and
- User-defined multiplications that are invertible for every element.

3.3.7 Multiplicative Abelian Group [concept.mult.abel]

A commutative MultiplicativeGroup is called multiplicative Abelian Group:

```
auto concept MultiplicativeAbelianGroup<typename Element>
  : MultiplicativeGroup<Element>, Commutative<math::mult<Element>, Element> {}
```

3.4 Generic Concepts with Two Operations [concept.math.two.functors]

These concepts specify algebraic structures with two operations. One is called "additive" and the other one "multiplicative". However, this is only a naming convention. The actual operations do not need to be addition and multiplication, they only must hold the same algebraic properties as addition and multiplication in the algebraic definitions.

In mathematical literature exist different definitions for these concepts. There is a general agreement that the additive structure is always an ABELIAN group; different opinions exist regarding the multiplication.

We oriented on definitions from text books, like the one from VAN DER WAERDEN [2], and also on the formal definitions in Tecton [1]. In these documents, rings are defined as semi-groups w.r.t. multiplication. Other authors like BOURBAKI define rings as multiplicative monoids. We call such structures 'ring with identity'.

3.4.1 Distributivity [concept.gen.distributive]

Two operations $\oplus$ and $\otimes$ are distributive if they hold:

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$
$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$$

The generic representation with concepts reads:

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

Please note that the naming AddOp and MultOp does not impose *any* semantic implication. The AddOp can be a logical **or** and the MultOp a logical **and**.

3.4.2 Ring [concept.gen.ring]

A ring is a mathematical structure with two operations that are distributive. One operation is an Abelian group and the other a semi-group.

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

The distributivity and the properties of one-operator concepts imply the behavior the Ring.

3.4.3 Ring with Identity [concept.gen.ring.identity]

Adding the identity to the multiplicative operation leads to a ring with identity:

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

Again, the requirements of this concepts are implied by the refined concepts.

3.4.4 Division Ring [concept.gen.division.ring]

A division ring is a ring with identity where in principle all elements except 0 have a reciprocal.

```
concept DivisionRing<typename AddOp, typename MultOp, typename Element>
  : RingWithIdentity<AddOp, MultOp, Element>, Inversion<MultOp, 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);
    }
}
```

In addition to the requirements of the refined concepts, we added the condition that the identity elements of the two operations are different. Their equality is mathematically not inconsistent but the set of elements would be reduced to one single element. The applicability of this set in practice would be thus rather limited.

3.4.5 Field [concept.gen.field]

A field is a commutative ring with division:

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

3.5 Concepts with Two Operators [concept.math.two.operators]

The concepts in this section refine the generic concepts with two operations (from this document and from N2645) to concepts with respect to the arithmetic operators.

3.5.1 Operator-based Ring [concept.op.ring]

A ring with respect to the operators +, -, and $*$ is called (by us) OperatorRing:

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

As with many concepts above, the requirements are entirely implied by the refined concepts.

3.5.2 Operator-based Ring with Identity [concept.op.ring.identity]

The concept ring with identity can be specialized to operators by:

```
auto concept OperatorRingWithIdentity<typename Element>
  : OperatorRing<Element>,
    MultiplicativeMonoid<Element>,
    RingWithIdentity<math::add<Element>, math::mult<Element>, Element>
{}
```

3.5.3 Operator-based Division Ring [concept.op.division.ring]

The concept division ring can be specialized to operators by:

```
auto concept OperatorDivisionRing<typename Element>
  : OperatorRingWithIdentity<Element>,
    MultiplicativePIMonoid<Element>,
    DivisionRing<math::add<Element>, math::mult<Element>, Element>
{}
```

3.5.4 Operator-based Field [concept.op.field]

The concept field can be specialized to operators by:

```
auto concept OperatorField<typename Element>
  : OperatorDivisionRing<Element>,
    Field<math::add<Element>, math::math::mult<Element>, Element>
{}
```

4 Concept Maps [concept.map.math.operators]

The declarations of modeling arithmatic concepts uses the intrinsic concepts from N2645.

Signed integrals form Abelian groups for addition (as long as no overflow occurs and the smallest representable value is not inverted). The multiplication has an identity element but we cannot define an inverse function (except for 1 and -1). The two operations are distributive so that signed integrals form a commutative ring with identity:

```
template <typename T>
  requires IntrinsicSignedIntegral<T>
concept_map OperatorRingWithIdentity<T> {}

template <typename T>
  requires IntrinsicSignedIntegral<T>
concept_map MultiplicativeCommutative<T> {}
```

All other models are implied.

Unsigned integrals have no inverse for addition. As a consequence they do not model the ring concepts but are only commutative monoids for the two operations:

```
template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map AdditiveCommutative<T> {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map AdditiveMonoid<T> {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map MultiplicativeCommutative<T> {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map MultiplicativeMonoid<T> {}
```

Intrinsic floating point types model Field and implicitly all other concepts:

```
template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Field<T> {}
```

The same is true for complex types of intrinsics:

```
template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Field< std::complex<T> > {}
```

Remark: The concept maps in this document imply the arithmetic concept maps in N2645.

5 Conclusion

Numeric libraries will benefit tremendously from defining and verifying mathematical characteristics. The concepts in this document might not be required very often directly in generic functions but will be very often refined in other concepts. They are the fundament of all algebraic concepts. The semantic definitions that can be build on top of these concepts will consilidate generic numeric libraries essentially. We expect a whole new era of scientific software with embedded semantic raising radically the confidence in the computed results by numeric software.

6 Acknowledgments

We thank Andrew Lumsdaine from Indiana University and Axel Voigt from Technische Universtität Dresden for supporting this work on mathematical concepts for the sake of the scientific computing community. We are also grateful to Karl Meerbergen for his discussions. Finally, we thank the Fermi National Accelerator Laboratory's Computing Division for its past and continuing support of our efforts to improve C++ for all our user communities.

References

[1] D. R. Musser, S. Schupp, C. Schwarzweller, and R. Loos. The Tecton concept library. Technical report, Fakultät für Informatik, Universität Tübingen, 1999.

[2] B. L. van der Waerden. *Algebra, Volume I and II.* Springer, 1990.