Fundamental Mathematical Concepts for the STL in C++0x

Document number: N2645 = 08-0155
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 STL functions and
- Select optimal algorithms according to semantic properties of operations.

One example for the concept-based verification is the standard function power. Functions that can be accelerated by concepts are accumulate and inner_product We therefore propose to replace the standard implementations of power, accumulate, and inner_product with the implementations in this document.

In addition to the usage in numeric STL algorithms, these concepts are the foundation for all advanced mathematical concepts like AdjointLinearOperator or HilbertSpace. 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 the careful design and standardization of these fundamental concepts is paramount.

2 Why Do We Need Semantic Mathematical Concepts?

Many mathematical properties as commutativity or invertibility are orthogonal to each other and most importantly can be considered independently on the specific computations that are performed in a certain operation. This independence of properties from computational details is successfully explored in mathematics. Our goal is to benefit in the same manner from known properties by choosing optimal algorithms according to the properties and regardless of the specific implementations.

Polymorphic function overloading and template specialization use this mechanism of algorithmical improvement for special cases. Unfortunately, the selective criterion must be squeazed into a type, a base type or a type pattern (for partial template specialization). This unnecessary restrictions can be avoided by using *semantic concepts*. They are entirely independent on any type or function definition.

Specializing generic function with respect to semantic concepts allows for algorithmical optimization that is also entirely independent on type or function definitions. The implications of this new opportunity are tremendous: generic functions can be algorithmically optimized according to properties that are impossible to express in existing programming languages for types that are not yet implemented.

Conversely, mathematical properties cannot be deduced in general from type information or structure. For instance, consider the binary operator +. It implements certain behaviors for standard data types: numbers are added, strings concatenates etc. Defining **operator**+ for new types suggests that the behavior is consistent to existing programs. How this consistency manifests lies in the eye of the beholder — but this is not the issue nonetheless. The point is moreover that C++ does not impose the slightest implication on the behavior of user-defined operators. The order of addition can be changed for intrinsic arithmetic types (modulus numeric subtleties) but not for strings. What shall we assume for user-defined types?

As a consequence, algorithms upon user-defined operators or functions must either be implemented conservatively, i.e. with minimal performance, or assume properties on computations that cannot be guaranteed. We will illustrate this on the example of std::accumulate. Computations of this style typically benefit from unrolling as this better explores super-scalar processors' performance. To add correctness to the acceleration, the binary operation in question must be commutative, which is unknown in this generic context.

The concepts we introduce here will provide the C++ programmer with the expressive power to specify the algebraic requirements in order to implement and use generic functionality optimally. More complex concepts like vector and Hilbert spaces rely on those proposed in this document and are subject to further proposals.

3 Synopsis

```
namespace std {

  // Basic mathematical concepts
  concept Commutative<typename Operation, typename Element>;
  concept SemiGroup<typename Operation, typename Element>;
  concept Monoid<typename Operation, typename Element>;
  auto concept Inversion<typename Operation, typename Element>;
  concept PIMonoid<typename Operation, typename Element>;
  concept Group<typename Operation, typename Element>;

  // Helper concepts for ALL maps of semantic concepts; not limited to this proposal
  concept IntrinsicType<typename T>;
  concept IntrinsicArithmetic<typename T>;
  concept IntrinsicIntegral<typename T>;
  concept IntrinsicSignedIntegral<typename T>;
  concept IntrinsicUnsignedIntegral<typename T>;
  concept IntrinsicFloatingPoint<typename T>;

  template <typename Iter, typename Value>
  Value inline accumulate(Iter first, Iter last, Value init);

  template <typename Iter, typename Value, typename Op>
  Value inline accumulate(Iter first, Iter last, Value init, Op op);
```

```
  template <typename Op, std::Semiregular Element, Integral Exponent>
  Element inline power(const Element& a, Exponent n, Op op);
}

namespace math {
  template <typename Element> struct add;
  template <typename Element> struct mult;
  template <typename Element> struct min;
  template <typename Element> struct max;
  template <typename Element> struct bitwise_and;
  template <typename Element> struct bitwise_or;
  template <typename Element> struct bitwise_xor;
}
```

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

The concepts in [concept.math.basic] specify the general behavior of binary operations.

4.1 Commutative Operations [concept.math.commative]

The concept Commutative defines that the arguments can be switched.

```
concept Commutative<typename Operation, typename Element>
  : std::Callable2<Operation, Element, Element>
{
    axiom Commutativity(Operation op, Element x, Element y) {
        op(x, y) == op(y, x);
    }
};
```

4.2 Semi-group [concept.math.semigroup]

SemiGroups allow to change the order of the operations (not the arguments).

```
concept SemiGroup<typename Operation, typename Element>
  : std::Callable2<Operation, Element, Element>
{
    axiom Associativity(Operation op, Element x, Element y, Element z) {
        op(x, op(y, z)) == op(op(x, y), z);
    }
};
```

We refrained from defining a separate concept Associative because this would be identical with SemiGroup.

4.3 Monoid [concept.math.monoid]

Adding an identity element results in a Monoid:

```
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;
    }
};
```

The axiom specifies the behavior of the identity element when applied from left and from right. We refrain from defining left and right identities for the sake of compactness. However, users are free to define their own concepts for the separation of identities.

Passing instances of Operation and of Element to the identity function has two advantages. Firstly, the syntax is more natural with type deduction than with explicitly passed template parameters. Secondly, access to run-time information is possible, e.g. to return a zero matrix of the proper size.

4.4 Inversion [concept.math.inversion]

The concept Inversion is only a structural concept:

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

};
```

That is, it only requires the existence of a unary inverse function with respect to the given binary operation. The behavior of this function is not characterized within this concept.

4.5 Partially Invertible Monoid [concept.math.pimonoid]

We extend the concept Monoid with partial inversion:

```
concept PIMonoid<typename Operation, typename Element>
  : Monoid<Operation, Element>,
    Inversion<Operation, Element>
{
    bool is_invertible(Operation, Element);
    requires std::Convertible<Inversion<Operation, Element>::result_type, Element>;

    axiom Invertibility(Operation op, Element x) {
        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);
    }
}
```

The concept relates the inverse function to the identity element. The concept of a partially invertible monoid — PIMonoid — is not presented in mathematical text books but important in practice. Most multiplications have at least one element that is not invertible. There are multiple examples where more than one non-invertible element (w.r.t. multiplication) exists:

- All singular matrices in the monoid of square matrices;
- All intervals containing zero in interval arithmetic; or
- All elements of a cyclic group that are co-prime to the cycle length.

The function is_invertible generalizes the test for division by zero.

4.6 Group [concept.math.group]

Algebraic structures with invertibility in every element are called group:

```
concept Group<typename Operation, typename Element>
  : PIMonoid<Operation, Element>
{
    bool is_invertible(Operation, Element) { return true; }
    axiom AlwaysInvertible(Operation op, Element x) {
        is_invertible(op, x);
    }
};
```

Due to the global invertibility the function is_invertible can be implemented by default. The axiom AlwaysInvertible holds for the default implementation and defined for the case that is_invertible is provided by the user.

5 Concepts for Intrinsic Types [concept.intrinsic]

The following concepts have only supporting character in order to ease the definition of **concept_map**s. They can be omitted at the price of more declaration effort in the **concept_map**s.

The concepts in this section must not be confused with the corresponding core concepts [concept.arithmetic] in N2502. For instance, the concept SignedIntegralLike is intended to characterize types that have the *same interface* as intrinsic signed. Thus, the same operators *exist* but it is *not guaranteed that they have the same semantics*. As structural concepts, these core concepts have the attribute **auto** so that it is not under the control of the programmer which types are models. As a consequence, modeling such interface requirements is *not a sufficient condition* for modeling semantic concepts. as in Section 7. Therefore, we propose to define concepts that are *only* modeled by the respective intrinsic types.

At first we introduce a concept for all intrinsic data types:

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

This concept is refined to IntrinsicArithmetic to represent all intrinsic arithmetic types:

```
concept IntrinsicArithmetic<typename T> : IntrinsicType<T> {}
```

The concept is further refined to:

```
concept IntrinsicIntegral<typename T> : IntrinsicArithmetic<T> {}
```

The following definitions are thus refinements of their respective structural concepts and the more general intrinsic concepts. IntrinsicSignedIntegral is refined from signed integral-like:

```
concept IntrinsicSignedIntegral<typename T>
  : std::SignedIntegralLike<T>,
    IntrinsicIntegral<T>
{}
```

Please note that these concept are not **auto**. Otherwise structurally equivalent types can automatically model the concepts. This is exactly what we want to avoid with these concepts.

Correspondingly, we define the concept for intrinsic unsigned integral types:

```
concept IntrinsicUnsignedIntegral<typename T>
  : std::UnsignedIntegralLike<T>,
    IntrinsicIntegral<T>
{}
```

Last but not least, we introduce a concept for intrinsic floating point types:

```
concept IntrinsicFloatingPoint<typename T>
  : std::FloatingPointLike<T>,
    IntrinsicArithmetic<T>
{}
```

6 Supporting Classes [concept.math.support]

We propose to define functors add and mult to connect the functor-based with the operator-based concepts. We suggest not to use the functors plus and multiplies from STL for three reasons.

1. The return type is hard-wired to the argument type and this disables expression templates.
2. The return type of a + a is not always the same as a. For instance, adding to **short int** yields an **int**.
3. plus and multiplies cannot be specialized by the user because it is defined in namespace std.

The add functor is defined by means of concept HasPlus:

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

Correspondingly we define the functor mult:

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

The min functor can be defined without the help of concepts:

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

The max functor is defined the analogously:

```
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; }
};
```

Bit-wise functors are accordingly defined. Conjunction reads:

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

Bit-wise disjunction is described as:

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

Bit-wise exclusive 'or' is given by:

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

6.1 Default Identity Elements [math.identity]

The additive identity is computed by default:

```
template <typename Element>
inline Element identity(const math::add<Element>&, const Element&)
{
    return 0;
}
```

String concatenation as Monoid operation requires a different definition:[1]

```
inline Element identity(const math::add<string>&, const string&)
{
    return std::string();
}
```

Containers as matrices and vectors typically must refer to some reference element in order to provide the corresponding identity element. For instance, to compute the sum of 2×3 matrices requires as identity element a zero matrix of dimension 2×3. The default identity for containers (or more generally collections) can be implemented in the following form:

[1] The default above tends to abort the program due to referring null pointers.

```
template <typename Coll>
  requires Collection<Coll>
inline Coll identity(const math::add<Coll>&, const Coll& ref)
{
    // Copy constructor to access dimension and other run-time data
    Coll tmp(ref);
    typedef Collection<C>::value_type Element;
    tmp= identity(math::add<Element>(), *Coll.begin());
    return tmp;
}
```

Remark: the definition for collections is not meant to be proposed for standardization (not yet, at least); it rather demonstrates why we propose to pass a reference element to the identity function.

The default implementation for multiplicative identities reads:

```
template <typename Element>
inline Element identity(const math::mult<Element>&, const Element&)
{
    return 1;
}
```

The extension to appropriate collections as square matrices would be analog.

The identity element for min is the maximal representable value:

```
template <typename Element>
inline Element identity(const math::min<Element>&, const Element&)
{
    using std::numeric_limits;
    return numeric_limits<Element>::max();
}
```

Accordingly, the minimal possible value is the identity for max:

```
template <typename Element>
inline Element identity(const math::max<Element>&, const Element&)
{
    using std::numeric_limits;
    return numeric_limits<Element>::min();
}
```

The bit-wise disjunction has an element with all bits set to 0 as identity element:

```
template <typename Element>
inline Element identity(const math::bitwise_or<Element>&, const Element&)
{
    return 0;
}
```

The same applies to bit-wise exclusive 'or':

```
template <typename Element>
inline Element identity(const math::bitwise_xor<Element>&, const Element&)
{
    return 0;
}
```

Conversely, the identity element of the bit-wise conjunction is an element with all bits set to 1:

```
template <typename Element>
inline Element identity(const math::bitwise_and<Element>&, const Element&)
{
    return 0 -1;
}
```

Remark: -1 might cause warnings with some compilers for unsigned integers.

6.2 Default Inverse Elements [math.identity]

The additive inverse in its generic form is based on the unary **operator**-:

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

The inverse of user-defined types can thus be defined in two ways: either by overloading the inverse function or by defining the unary **operator**-.

The multiplicative inverse bases on the identity element and the division by default:

```
template <typename Element>
inline Element inverse(const math::mult<Element>& op, const Element& v)
{
    return identity(op, v) / v;
}
```

The inverse element of exclusive 'or' is itself:

```
template <typename Element>
inline Element inverse(const math::bitwise_xor<Element>& op, const Element& v)
{
    return v;
}
```

7 Concept Maps

7.1 Intrinsic Types [concept.map.intrinsic]

To abbreviate the map definitions of mathematical concepts we start with the maps of the intrinsic concepts:

```
concept_map IntrinsicSignedIntegral<char> {}
concept_map IntrinsicSignedIntegral<signed char> {}
concept_map IntrinsicUnsignedIntegral<unsigned char> {}
concept_map IntrinsicSignedIntegral<short> {}
concept_map IntrinsicUnsignedIntegral<unsigned short> {}
concept_map IntrinsicSignedIntegral<int> {}
concept_map IntrinsicUnsignedIntegral<unsigned int> {}
concept_map IntrinsicSignedIntegral<long> {}
concept_map IntrinsicUnsignedIntegral<unsigned long> {}
concept_map IntrinsicSignedIntegral<long long> {}
concept_map IntrinsicUnsignedIntegral<unsigned long long> {}
```

```
concept_map IntrinsicFloatingPoint<float> {}
concept_map IntrinsicFloatingPoint<double> {}

concept_map IntrinsicType<bool> {}
```

7.2 Arithmetic Operations [concept.map.math.arithmetic]

Signed integrals form commutative 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 Commutative< math::add<T>, T > {}

template <typename T>
  requires IntrinsicSignedIntegral<T>
concept_map Group< math::add<T>, T > {}

template <typename T>
  requires IntrinsicSignedIntegral<T>
concept_map Commutative< math::mult<T>, T > {}

template <typename T>
  requires IntrinsicSignedIntegral<T>
concept_map Monoid< math::mult<T>, 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 Commutative< math::add<T>, T > {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map Monoid< math::add<T>, T > {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map Commutative< math::mult<T>, T > {}

template <typename T>
  requires IntrinsicUnsignedIntegral<T>
concept_map Monoid< math::mult<T>, T > {}
```

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

```
template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Commutative< math::add<T>, T > {}
```

```
template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Group< math::add<T>, T > {}

template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Commutative< math::mult<T>, T > {}

template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map PIMonoid< math::mult<T>, T > {}
```

The same is true for complex types of intrinsics:

```
template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Commutative< math::add<std::complex<T> >, std::complex<T> > {}

template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Group< math::add<std::complex<T> >, std::complex<T> > {}

template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map Commutative< math::mult<std::complex<T> >, std::complex<T> > {}

template <typename T>
  requires IntrinsicFloatingPoint<T>
concept_map PIMonoid< math::mult<std::complex<T> >, std::complex<T> > {}
```

7.3 Concept Maps for Min and Max [concept.map.math.minmax]

The minimum computation of two values is a commutative monoid for all intrinsic types:

```
template <typename T>
  requires IntrinsicArithmetic<T>
concept_map Commutative< max<T>, T > {}

template <typename T>
  requires IntrinsicArithmetic<T>
concept_map Monoid< max<T>, T > {}
```

(Most likely it is a commutative monoid for all other types as well but we cannot guarantee this.) As a consequence, the accelerated version of accumulate can be used. The identity element of max<T> is the smallest representable value of type T.

Conversely the identity element of min<T> is the largest representable value.

```
template <typename T>
  requires IntrinsicArithmetic<T>
concept_map Commutative< min<T>, T > {}

template <typename T>
  requires IntrinsicArithmetic<T>
concept_map Monoid< min<T>, T > {}
```

7.4 Concept Maps for Logical Operations [concept.map.math.logical]

Logical conjunction is a commutative monoid. Its identity element is **true** for **bool** and for integral types 1:

```
template <typename T>
  requires Intrinsic<T> && std::HasLogicalAnd<T>
concept_map Commutative< std::logical_and<T>, T > {}

template <typename T>
  requires Intrinsic<T> && std::HasLogicalAnd<T>
concept_map Monoid< std::logical_and<T>, T > {}
```

Logical disjunction is also a commutative monoid.

```
template <typename T>
  requires Intrinsic<T> && std::HasLogicalOr<T>
concept_map Commutative< std::logical_or<T>, T > {}

template <typename T>
  requires Intrinsic<T> && std::HasLogicalOr<T>
concept_map Monoid< std::logical_or<T>, T > {}
```

Both operations are distributive. Please note that for logical operations the distributivity holds in both directions.

```
template <typename T>
  requires Intrinsic<T> && std::HasLogicalAnd<T> && std::HasLogicalOr<T>
concept_map Distributive<std::logical_and<T>, std::logical_or<T>, T> {}

template <typename T>
  requires Intrinsic<T> && std::HasLogicalAnd<T> && std::HasLogicalOr<T>
concept_map Distributive<std::logical_or<T>, std::logical_and<T>, T> {}
```

7.5 Concept Maps for Bit-wise Operations [concept.map.math.bit]

Bit-wise 'and' is a commutative monoid for integral types. The identity element is a value of type T with all 1s in its binary representation.

```
template <typename T>
  requires IntrinsicIntegral<T>
concept_map Commutative< bitwise_and<T>, T > {}

template <typename T>
  requires IntrinsicIntegral<T>
concept_map Monoid< bitwise_and<T>, T > {}
```

Likewise, bit-wise 'or' is a commutative monoid for integral types The identity element is a value of type T with all 0s in its binary representation.

```
template <typename T>
  requires IntrinsicIntegral<T>
concept_map Commutative< bitwise_or<T>, T > {}

template <typename T>
  requires IntrinsicIntegral<T>
concept_map Monoid< bitwise_or<T>, T > {}
```

Both operations are distributive in both directions.

```
template <typename T>
  requires IntrinsicIntegral<T>
concept_map Distributive<bitwise_and<T>, bitwise_or<T>, T> {}

template <typename T>
  requires IntrinsicIntegral<T>
concept_map Distributive<bitwise_or<T>, bitwise_and<T>, T> {}
```

The bit-wise exclusive 'or' is a commutative group:

```
template <typename T>
  requires IntrinsicIntegral<T>
concept_map Commutative< bitwise_xor<T>, T > {}

template <typename T>
  requires IntrinsicIntegral<T>
concept_map Group< bitwise_xor<T>, T > {}
```

7.6 Concept Map for String Concatenation [concept.map.math.string]

The string concatenation is a monoid with the empty string as identity:

```
concept_map AdditiveMonoid<std::string> {}
```

8 Applications

We will motivate the definition of algebraic concepts with the help of two examples that are choosen because they are:

- Generic;
- Algorithmically simple; and
- Most importantly can be accelerated when mathematical properties are known.

8.1 Application 1: std::accumulate [lib.accumulate]

The standard accumulate function is a prototype of a generic function. It can be applied to any input iterator and with respect to any binary operation. In preparation for the later specialization we modify the standard implementation slightly:

```
template <std::InputIterator Iter, std::CopyConstructible Value, typename Op>
      requires std::Callable2<Op, Value, Value>
            && std::CopyAssignable<Value, std::Callable2<Op, Value, Value>::result_type>
Value inline accumulate(Iter first, Iter last, Value init, Op op)
{
    for (; first != last; ++first)
        init= op(init, Value(*first));
    return init;
}
```

Unrolled high-performance implementation for standard arithmetic are significantly faster (when the data is in cache).

The unrolling relies on semantic properties that the generic implementation cannot assume in general. Concepts allow us to define the semantic requirements generically. To accelerate the accumulate function generically, the iterator must be randomly accessible and the operation must be a commutative monoid:

```
template <std::RandomAccessIterator Iter, std::CopyConstructible Value, typename Op>
    requires std::CopyAssignable<Value, std::Callable2<Op, Value, Value>::result_type>
          && Commutative<Op, Value>
          && Monoid<Op, Value>
          && std::Convertible<Monoid<Op, Value>::identity_result_type, Value>
Value inline accumulate(Iter first, Iter last, Value init, Op op)
{
    typedef typename std::RandomAccessIterator<Iter>::difference_type difference_type;
    Value 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, Value(first[i]));
        t1= op(t1, Value(first[i+1]));
        t2= op(t2, Value(first[i+2]));
        t3= op(t3, Value(first[i+3]));
    }
    for (; i < size; i++)
        t0= op(t0, Value(first[i]));

    t0= op(t0, t1), t2= op(t2, t3), t0= op(t0, t2);
    return t0;
}
```

In our opinion, the requirements Commutative<Op, Value> and Monoid<Op, Value> are by far the most important details in the implementation above because this constitutes a significant qualitative improvement over C++98. Requiring *SEMANTIC* properties in the source code and checking this with the compiler is an entirely new opportunity provided by concepts. The *semantically correct choise* between the two implementations would not be possible without concepts.[2]

A numerical experiment with a std::vector of **double** shows that the unrolling effectively speeds up the computation of a product. Other experiments that computed sums, products, minima, and maxima of **int**, **double**, and complex<**double**> vectors exhibited similar accelerations.

Remark: In the same fashion as accumulate we can accelerate std::inner_product.

8.2 Application 2: std::power [lib.power]

> If you have defined $*$ or MonoidOperation to be a non-associative operation, then power *will give you the wrong answer.*
> *From the STL documentation of power.*

For efficiency reasons, this function relies on associativity. Without concepts, this requirement cannot be expressed and the function cannot be limited to associative operations.

[2] Admittedly, the behavior could be theoretically emulated with type traits but its realization would be unbearably cumbersome.

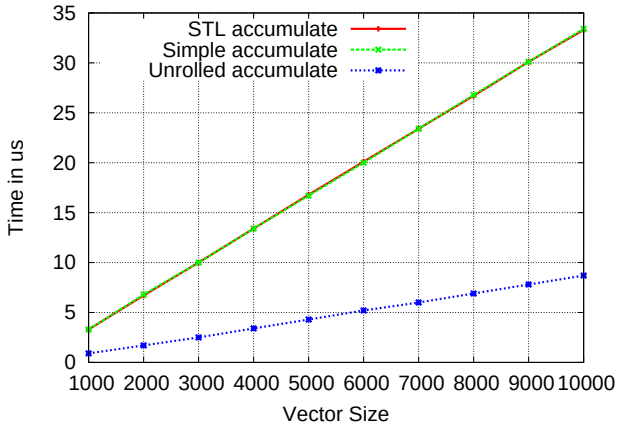

Figure 1: Multiplication of **double**

With concepts, we can express the semantic prerequisite. The usage of concepts also allows for generalizing the applicability in two directions. For a sub-range of exponents we can accept more general mathematical concepts. Vice versa, the range of exponents can be extended for refined concepts. The most general version does not require any algebraic property. The absence of an identity element requires the exponent to be at least 1.

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires std::Callable2<Op, Element, Element>
          && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    if (n < 1) throw std::range_error("power [magma]: n must be > 0");

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

The complexity of this computation is linear. If the operation is associative, i.e. models SemiGroup, the complexity can be reduced to logarithmic by computing sub-expressions:

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires SemiGroup<Op, Element>
          && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    if (n < 1) throw std::range_error("power [SemiGroup]: n must be > 0");
    Exponent half= n >> 1;
    if (half == 0)
        return a;
    Element value= power(a, half, op);
    value= op(value, value);
    if (n & 1)
        value= op(value, a);
    return value;
```

```
}
```

The code simplifies significantly if the operation has an identity element, i.e. the structure is a monoid:

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires Monoid<Op, Element>
        && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    if (n < 0) throw std::range_error("power [Monoid]: n must be >= 0");

    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;
}
```

In addition, zero is now allowed as exponent.

The exponent can be extended further to negative numbers if an inverse operation exists. If not all elements are invertible, a test is needed to avoid illegal operations (e.g. division by zero):

```
template <typename Op, std::Semiregular Element, Integral Exponent>
   requires PIMonoid<Op, Element>
        && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
     if (n < 0 && !is_invertible(op, a))
          throw std::range_error("power [PIMonoid]: a must be invertible with n < 0");
     return n < 0 ? multiply_and_square(Element(inverse(op, a)), Exponent(-n), op)
                  : multiply_and_square(a, n, op);
}
```

The function multiply_and_square is the out-sourced code from power for monoids.

Last but not least for groups we can omit the test:

```
template <typename Op, std::Semiregular Element, Integral Exponent>
    requires Group<Op, Element>
        && std::Convertible<std::Callable2<Op, Element, Element>::result_type, Element>
inline Element power(const Element& a, Exponent n, Op op)
{
    return n < 0 ? multiply_and_square(Element(inverse(op, a)), Exponent(-n), op)
                 : multiply_and_square(a, n, op);
}
```

9 Conclusion

As much as we appreciate the more readable error message yielded by syntactic concepts, this is mainly a convenience to ease the debugging of compiler-detectable errors. In contrast to this,

semantic concepts allow for the discovery of wrong behavior that was not even possible to express in source code before let alone to verify. The mathematical concepts allow to implement the power function from STL properly and provide well-tuned versions according to the mathematical behavior of the operation. The standard functions accumulate and inner_product can be accelerated significantly by exploring algebraic properties. We therefore propose to replace the standard implementations of power, accumulate, and inner_product with the implementations in this document.

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

10 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.