Design of Concept Libraries for C++

Andrew Sutton and Bjarne Stroustrup

Texas A&M University
Department of Computer Science and Engineering
{asutton, bs}@cse.tamu.edu

Abstract. We present a set of concepts (requirements on template arguments) for a large subset of the ISO C++ standard library. The goal of our work is twofold: to identify a minimal and useful set of concepts required to constrain the library's generic algorithms and data structures and to gain insights into how best to support such concepts within C++. We start with the design of concepts rather than the design of supporting language features; the language design must be made to fit the concepts, rather than the other way around. A direct result of the experiment is the realization that to simply and elegantly support generic programming we need two kinds of abstractions: *constraints* are predicates on static properties of a type, and *concepts* are abstract specifications of an algorithm's syntactic and semantic requirements. Constraints are necessary building blocks of concepts. Semantic properties are represented as axioms. We summarize our approach: $concepts = constraints + axioms$. This insight is leveraged to develop a library containing only 14 concepts that encompassing the functional, iterator, and algorithm components of the C++ Standard Library (the STL). The concepts are implemented as constraint classes and evaluated using Clang's and GCC's Standard Library test suites.

Keywords: Generic programming, concepts, constraints, axioms, C++.

1 Introduction

Concepts (requirements on template arguments) are the central feature of C++ generic library design; they define the terms in which a library's generic data structures and algorithms are specified. Every working generic library is based on concepts. These concepts may be represented using specifically designed language features (e.g. [15, 20]), in requirements tables (e.g., [2,22]), as comments in the code, in design documents (e.g., [21]), or simply in the heads of programmers. However, without concepts (formal or informal), no generic code could work.

For example, a Matrix library that allows a user to supply element types must have a concept of what operations can be used on its elements. In other words, the Matrix library has a concept of a "number," which has a de facto definition in terms of operations used in the library code. Interpretations of "number" can vary dramatically from library to library. Is a polynomial function a number'? Are numbers supposed to support division? Are operations on numbers supposed to be associative? Regardless, every generic Matrix library has a "number" concept.

In this work, we experiment with designs for concepts by trying to minimize the number of concepts required to constrain the generic algorithms and data structures of a library. To do such experiments, we need ways to represent concepts in C++ code. Our approach is to use constraint classes, which allows us to experiment with different sets of concepts for a library without pre-judging the needs for specific language support features. Once we have experimented with the design of concepts for several libraries, we expect our criteria for concept design and use to have matured to the point where we can confidently select among the reasonably well-understood language design alternatives. Language support should be determined by ideal usage patterns rather than the other way around.

Previous work in the development of concept libraries has resulted in the definition of large numbers of concepts to constrain comparatively few generic components. For example, the last version of the C++ Draft Standard to include concepts [5] defined 130 concepts to constrain approximately 250 data structures and algorithms. Nearly 2/3rds of that library's exported abstractions were concept definitions. Out of the 130 concepts, 108 relate are used to (directly or indirectly) express the functional, iterator, and algorithms parts of the standard library that we address here. In comparison, the Elements of Programming (EoP) book [34] covers similar material with 52 concepts for about 180 different algorithms (just under 1/3rd).

Obviously, not all of the concepts in either publication are "equally abstract". That C++ Draft Standard includes about 30 concepts enumerating requirements on overloadable operator's alone and several metaprogramming-like concepts such as **Rvalue_of**. The EoP book lists many concepts that explore variations on a theme; invariants are strengthened or weakened, operators are dropped, and definition spaces are restricted. From these and similar designs of concepts, it has been concluded that writing a generic library requires exhaustive enumeration of overloadable operations, conceptual support for template metaprogramming, and massive direct language support for concepts.

Fortunately, such conclusions are false. Furthermore, they are obviously false. The overwhelming majority of generic libraries, including the original version of STL, were written without language support for concepts and used without enforcement mechanisms. The authors of these libraries did not consider some hundred different concepts during their development. Rather, the generic components of these libraries were written to a small set of idealized abstractions.

We argue that generic libraries are defined in terms of small sets of abstract and intuitive concepts, and that an effective specification of concepts is the product of an iterative process that minimizes the number of concepts while maintaining expressive and effective constraints. Ideal concepts are fundamental to their application domain (e.g., *String*, *Number*, and *Iterator*), and consequently it is a rare achievement to find a genuine new one. These abstract and intuitive concepts must not be lost in the details needed to express them in code. Neither should the ease of learning and ease of use offered by these concepts be compromised by an effort to minimize the constraints of every algorithm.

To explore these ideas concretely, we develop a minimal design of the concepts for the STL that encompasses its functional, iterator, and algorithm libraries. The resulting concept library defines only 14 concepts, 94 fewer than originally proposed for the corresponding parts of the C++ Standard Library. We leverage the results of the experiment and the experience gained to derive a novel perspective on concepts for the C++ programming language. Our approach reveals a distinct difference between requirements that represent domain abstractions and those that support their specification: *concepts* and *constraints*, respectively. Concepts based on this distinction results in a conceptually simpler and more modular design.

We validate the design of the concepts by implementing them within the framework of a concept emulation library for the Origin C++ Libraries [38]. The libraries are implemented using the 2011 ISO C++ standard facilities. The concepts are then integrated into the STL subset of Clang's libc++ and GCC's libstdc++ and checked against their test suites for validation.

The results of this work yield several contributions to our knowledge of generic programming. First, we demonstrate that it is both feasible and desirable to experiment to seek a minimal conceptual specification for a generic library. Second, we demonstrate that distinguishing between concepts (as abstractions) and constraints (as static requirements) is an effective way to achieve this goal. Third, we identify semantic properties, axioms, as the key to practical discrimination of concepts from constraints. Finally, we provide a simpler and easier to use specification of key parts of the ISO C++ standard library.

2 Related Work

Generic programming is rooted in the ability to specify code that will work with a variety of types. Every language supporting generic programming must address the issue of how to specify the interface between generically written code and the set of concrete types on which it operates. Several comparative studies have been conducted in support for generic programming and type constraints [18]. These studies were leveraged to support the definition of concepts for C++

ML relies on *signatures* to specify the interface of modules and constrain the type parameters of functors [27]. Operations in the signature are matched against those defined by a structure to determine conformance. In Haskell, *type classes* denote sets of types that can be used with same operations [24]. A type is made a member of that set by explicitly declaring it an *instance* and implementing the requisite operations. Type classes are used as constraints on type parameters in the signatures of polymorphic functions. Similarities between Haskell type classes and C++ concepts have also been explored [6, 7]. AXIOM *categories* are used to define the syntax and semantics of algebraic domains [10]. The specification of algebraic structures helped motivate the design of its category system [9, 11]. Requirements on type parameters in Eiffel, Java, and C# are specified in terms of inherited interfaces. Checking the conformance of a supplied type argument entails determining if it is a subtype of the required interface or class [8, 25, 26, 41].

From the earliest days of the design of C++ templates, people have been looking for ways to specify and constrain template arguments [35, 36]. For the C++0x standards effort two proposals (with many variants) were considered [15, 20]. A language framework supporting generic programming was developed in support of these proposals [32, 33].

There has been less work on the design of concepts themselves (as opposed to studying language support). The dominant philosophy of concept design has focused on "lifting" algorithms from specific implementations to generic algorithms with specific requirements on arguments [28]. However, applying the same process to the concepts (iterative generalization) can lead to an explosion in the number of concepts as requirements are minimized for each algorithm in isolation. The (ultimately unsuccessful) design for C++0x included a carefully crafted and tested design for the complete C++0x standard library including around 130 concepts [5]. In [13], Dehnert and Stepanov defined properties of regular types and functions. Stepanov and McJones carefully worked out a set of concepts for their variant of the STL in EoP [34]; Dos Reis implemented verification for those and found a few improvements [14].

Other research has focused on the use of concepts or their expression in source code. Bagge and Haveraaen explored the use of axioms to support testing and the semantic specification of algebraic concepts [4]. Pirkelbauer et al [29] and Sutton and Maletic [40] studied concepts through mechanisms for automatically extracting requirements from actual source code. Also, many aspects of concepts can be realized idiomatically in C++0x [12, 39]; this is the basis of our implementation in this work.

3 Requirements for Concept Design

The immediate and long-term goals of this research are to develop an understanding of the principles of concepts and to formulate practical guidelines for their design. A midterm goal is to apply that understanding to the design of language features to support the use of concepts, especially in C++.

Here, we present an experiment in which we seek a conceptual specification for the STL that defines a minimum number of concepts. This goal is in stark contrast to previous work [5] where the explicit representation of even the subtlest distinctions in requirements was the ideal.

The minimization aspect of the experiment constrains the design in such a way that the resulting concepts must be abstract and expressive. A concept represents a generic algorithm's requirements on a type such as a *Number* or an *Iterator*. A concept is a predicate that can be applied to a type to ascertain if it meets the requirements embodied by the concept. An expressive concept, such as Iterator, allows a broad range of related expressions on a variety of types. In contrast, a simple syntactic requirement, such as requiring default construction, is a constraint on implementers and does not express a general concept.

The effect of this minimization is ease of learning and ease use. In particular, it provides library designers and users with a simple, strong design guideline that could never be achieved with (say) 100 primitive requirements (primitive in the mathematical sense). The design of a concept library is the result of two minimization problems: concept and constraint minimization.

Concept minimization seeks to find the smallest set of concepts that adequately represent the abstractions of a generic library. The problem is na?vely solved by defining a single concept that satisfies the requirements of all templates. For example, mutable random-access iterators work with practically all STL algorithms so a minimum concept specification might simply be a **Random_access_iterator**. This would result in over-constrained templates and make many real-world uses infeasible. For example, a linked-list cannot easily and economically support random access and an input stream cannot support random access at all. Conversely, the popular object-oriented notion of a universal *Object* type under-constrains interfaces, so that programmers have to rely on runtime resolution, runtime error handling, and explicit type conversion.

Constraint minimization seeks to find a set of constraints that minimally constrain the template arguments. This problem is na?vely solved by naming the minimal set of type requirements for the implementation of an algorithm. If two algorithms have non-identical but overlapping sets of requirements, we factor out the common parts, which results in three logically minimal sets of requirements. This process is repeated for all algorithms in a library. Concepts developed in this manner resemble the syntactic constructs from which they are derived; the number of concepts is equal to the number of uniquely typed expressions in a set of the algorithms. This results in the total absence of abstraction and large numbers of irregular, non-intuitive concepts. In the extreme, every implementation of every algorithm needs its own concept, thus negating the purpose of "concepts" by making them nothing but a restatement of the requirements of a particular piece of code.

Effective concept design solves both problems through a process of iterative refinement. An initial, minimal set of concepts is defined. Templates are analyzed for requirements. If the initial set of concepts produces overly strict constraints, the concept definitions must be refactored to weaken the constraints on the template's arguments. However, the concepts must not be refactored to the extent that they no longer represent intuitive and potentially stable abstractions.

4 Concepts = Constraints + Axioms

Previous work on concepts use a single language mechanism to support both the abstract interfaces (represented to users as concepts) and the queries about type properties needed to eliminate redundancy in concept implementations. From a language-technical point of view, that makes sense, but it obscures the fundamental distinction between interface and implementation of requirements. Worse, it discourages the programmer from making this distinction by not providing language to express the distinction. We conclude that we need separate mechanisms for the definition of concepts and for their implementation.

We consider three forms of template requirements in our design:

– *Constraints* define the statically evaluable predicates on the properties and syntax of types, but do not represent cohesive abstractions.
– *Axioms* state semantic requirements on types that should not be statically evaluated. An axiom is an invariant that is assumed to hold (as opposed required to be checked) for types that meet a concept.
– *Concepts* are predicates that represent general, abstract, and stable requirements of generic algorithms on their argument. They are defined in terms of constraints and axioms.

Constraints are closely related to the notion of type traits, metafunctions that evaluate the properties of types. We choose the term "constraint" over "trait" because of the varied semantics already associated the word "trait." The word "constraint" also emphasizes the checkable nature of the specifications. The terms "concept" and "axiom" are well established in the C++ community [1, 2, 5, 16, 18, 22, 34, 39].

Constraints and axioms are the building blocks of concepts. Constraints can be used to statically query the properties, interfaces, and relationships of types and have direct analogs in the Standard Library as type traits (e.g., **is_const**, **is_constructible**, **is_-same**). In fact, a majority of the concepts in C++0x represent constraints [5].

Axioms specify the meaning of those interfaces and relationships, type invariants, and complexity guarantees. Previous representations of concepts in C++ define axioms as features expressed within concepts [5, 16]. In our model, we allow axioms to be written outside of concept specifications, not unlike *properties* in the EoP book [34]. This allows us to distinguish between semantics that are inherent to the meaning of a concept and those that can be stated as assumed by a particular algorithm. The distinction also allows us to recombine semantic properties of concepts without generating lattices of semantically orthogonal concepts, which results in designs with (potentially far) fewer concepts.

The distinctive property that separates a concept from a constraint is that it has semantic properties. In other words, we can write axioms for a concept, but doing so for a constraint would be farfetched. This distinction is (obviously) semantic so it is possible to be uncertain about the classification of a predicate, but we find that after a while the classification becomes clear to domain experts. In several cases, the effort to classify deepened our understanding of the abstraction and in four cases the "can we state an axiom?" criterion changed the classification of a predicate, yielding—in retrospect—a better design. These improvements based on the use of axioms were in addition to the dramatic simplifications we had achieved using our earlier (less precise) criteria of abstractness and generality.

Our decision to differentiate concepts and constraints was not made lightly, nor was the decision to allow axioms to be decoupled from concepts. These decisions are the result of iteratively refining, balancing, and tuning concepts for the STL subject to the constraints of the experiment. These insights, once made, have resulted in a clear, concise, and remarkably consistent view of the abstractions in the STL. The distinction between concepts, constraints, and axioms is a valuable new design tool that supports modularity and reuse in conceptual specifications. We expect the distinction to have implications on the design of the language support for concepts. For example, if we need explicit modeling statements (concept maps [20]; which is by no means certain), they would only be needed for concepts. Conversely, many constraints are compiler intrinsic [22]. These two differences allow for simpler compilation model and improved compile times compared to designs based on a single language construct [5, 19].

As an example of the difference, consider the C++0x concepts **HasPlus**, **HasMinus**, **HasMultiply**, and **HasDivide**. By our definition, these are not concepts. They are nothing but requirements that a type has a binary operator **+**, **-**, *****, and, **/**, respectively. No meaning (semantics) can be ascribed to each in isolation. No algorithm could be written based solely on the requirement of an argument type providing (say) - and *. In contrast, we can define a concept that requires a combination of those (say, all of them) with the usual semantics of arithmetic operations conventionally and precisely stated as axioms. Such concepts are the basis for most numeric algorithms.

It is fairly common for constraints to require just a single operation and for a concept to require several constraints with defined semantic relationships among them. However, it is quite feasible to have a single-operation concept and, conversely, a multi-operation constraint. For example, consider the interface to balancing operations from a framework for balanced-binary tree implementations [3]:

```
constraint Balancer<typename Node> {
  void add_fixup(Node*);
  void touch(Node*);
  void detach(Node*);
}
```

This specifies the three functions that a balancer needs to supply to be used by the framework. It is clearly an internal interface of little generality or abstraction; it is an implementation detail. If we tried hard enough, we might come up with some semantic specification (which would depend on the semantics of **Node**), but it would be unlikely to be of use outside this particular framework (where it is used in exactly one place). Furthermore, it would be most unlikely to survive a major revision and extension of the framework unchanged. In other words, the lack of generality and the difficulty of specifying semantic rules are strong indications that Balancer is not a general concept, so we make it a constraint. It is conceivable that in the future we will understand the application domain well enough to support a stable and formally defined notion of a **Balancer** and then (and only then) would we promote **Balancer** to a concept by adding the necessary axioms. Partly, the distinction between concept and constraint is one of maturity of application domain.

Constraints also help a concept design accommodate *irregularity*. An irregular type is one that almost meets requirements but deviates so that a concept cannot be written to express a uniform abstraction that incorporates the irregular type. For example, **ostream_iterator** and **vector<bool>::iterator** are irregular in that their value type cannot be deduced from their reference type. Expression templates are paragons of irregularity: they encode fragments of an abstract syntax tree as types and can support lazy evaluation without additional syntax [42]. We can't support every such irregularity without creating a mess of "concepts" that lack proper semantic specification and are not stable (because they essentially represent implementation details). However, such irregular iterators and expression templates are viable type arguments to many STL algorithms. Constraints can be used to hide these irregularities, thus simplifying the specification of concepts. A long-term solution will have to involve cleaner (probably more constrained) specification of algorithms.

5 Concepts for the STL

Our concept design for the STL is comprised of only 14 concepts, 17 supporting constraints, and 4 independent axioms. These are summarized in Table 1.

We present the concepts, constraints, and axioms in the library using syntax similar to that developed for C++0x [5, 15, 20]. The syntax used in this presentation can be mapped directly onto the implementation, which supports our validation method. Also,

Table 1. Concepts, constraints, and axioms

Concepts		***Constraints***	
Regularity	*Iterators*	*Operators*	*Language*
Comparable	**Iterator**	**Equal**	**Same**
Ordered	**Forward_iterator**	**Less**	**Common**
Copyable	**Bidirectional_iterator**	**Logical_and**	**Derived**
Movable	**Random_access_iterator**	**Logical_or**	**Convertible**
Regular		**Logical_not**	**Signed_int**
		Callable	
Functional	*Types*	*Initialization*	*Other*
Function	**Boolean**	**Destructible**	**Procedure**
Operation		**Constructible**	**Input_iterator**
Predicate		**Assignable**	**Output_iterator**
Relation			
Axioms			
Equivalence_relation			
Strict_weak_order			
Strict_total_order			
Boolean_algebra			

the concept and constraint names are written as we think they should appear when used with language support, not as they appear in the Origin library.

To distinguish C++ from purely object-oriented or purely functional type systems, we preface the presentation of these concepts with a summary view of values and objects within the context of the C++ type system. In brief, a *value* is an abstract, immutable element such as the number 5 or the color red. An *object* resides in a specific area of memory (has identity) and may hold a value. In C++, values are represented by *rvalues*: literals, temporaries, and constant expressions (**constexpr** values). Objects are *lvalues* that support mutability in the forms of assignment and move (i.e., variables). Objects are uniquely identified by their address. A *constant* of the form **const T** is an object that behaves like a value in that it is immutable, although it still has identity. A *reference* is an alias to an underlying value (rvalue reference) or object (lvalue reference). In order for our design to be considered viable, it must address the differences between the various kinds of types in the C++ type system [37].

5.1 Regular Types

In our design, all abstract types are rooted in the notion of *regularity*. The concept Regular appears in some form or other in all formalizations of C++ types [22, 32, 34]; it expresses the notion that an object is fundamentally well behaved, e.g. it can be constructed, destroyed, copied, and compared to other objects of its type. Also, *Regular* types can be used to define objects. Regularity is the foundation of the value-oriented semantics used in the STL, and is rooted in four notions: *Comparability*, *Order*, *Movability*, and *Copyability*. Their representation as concepts follow.

```
concept Comparable<typename T> {
  requires constraint Equal<T>; // syntax of equality
  requires axiom Equivalence_relation<equal<T>, T>; // semantics of equivalence

  template<Predicate P>
```

```
    axiom Equality(T x, T y, P p) {
        x==y => p(x)==p(y); // if x==y then for any Predicate p, p(x) == p(y)
    }
    axiom Inequality(T x, T y) {
        (x!=y) == !(x==y); // inequality is the negation of equality
    }
}
```

The **Comparable** concept defines the notion of equality comparison for its type argument. It requires an operator **==** via the constraint **Equal**, and the meaning of that operator is imposed by the axiom **Equivalence_relation**. The **Equality** axiom defines the actual meaning of equality, namely that two values are equal if, for any **Predicate**, the result of its application is equal. We use the C++0x axiom syntax with the**=>** (implies) operator added [16]. The **Inequality** axiom connects the meaning of equality to inequality. If a type defines **==** but does not a corresponding **!=**, we can automatically generate a canonical definition according to this axiom (as described in Sect. 6). The **Inequality** axiom requires that user-defined **!=** operators provide the correct semantics.

We define the notion of *Order* similarly:

```
concept Ordered<Regular T> {
    requires constraint Less<T>;
    requires axiom Strict_total_order<less<T>, T>;
    requires axiom Greater<T>;
    requires axiom Less_equal<T>;
    requires axiom Greater_equal<T>;
}
```

We factor out the axioms just to show that we can, and because they are examples of axioms that might find multiple uses:

```
template<typename T>
axiom Greater(T x, T y) {
    (x>y) == (y<x);
}
template<typename T>
axiom Less_equal(T x, T y) {
    (x<=y) == !(y<x);
}
template<typename T>
axiom Greater_equal(T x, T y) {
    (x>=y) == !(x<y);
}
```

As with **Comparable**, the definition of requirements is predicated on a syntactic constraint (Less) and a semantic requirement (**Strict_total_order**). Obviously, not all types inherently define a total order; IEEE 754 floating point values define only a partial order when considering NaN values. Because this is an axiom and can't be proven by a C++ compiler, we are allowed to assume that it holds. The required axioms connect the meaning of the other relational operations to **<**.

```
concept Copyable<Comparable T> {
    requires constraint Destructible<T> && Constructible<T, const T&>

    axiom Copy_equality(T x, T y) {
        x==y => T{x}==y && x==y; // copy construction copies (non-destructively)
    }
};
```

A **Copyable** type is both copy constructible and **Comparable**. The **Copy_equality** axiom states that a copy of an object is equal to its original. **Copyable** (and also **Movable**) types must be **Destructible**, ensuring that the program can destroy the constructed objects.

```
concept Movable<typename T> {
    requires constraint Destructible<T> && Constructible<T, T&&>

    axiom Move_effect(T x, T y) {
        x==y => T{move(x)}==y && can_destroy(x); // original is valid but unspecified
    }
}
```

A **Movable** type is move constructible. Moving an object puts the moved-from object in a valid but unspecified state. The C++0x axiom syntax provides no way of expressing “valid but unspecified” so we introduce the primitive predicate **can_destroy()** to express that requirement.

A *Regular* type can be used to create objects, declare variables, make copies, move objects, compare values, and default-construct. In essence, the notion of regularity defines the basic set operations and guarantees that should be available for all value-oriented types.

```
concept Regular<typename T> {
    requires Movable<T> && Copyable<T>;
    requires constraint Constructible<T> // default construction
                        && Assignable<T, T&&> // move assignment
                        && Assignable<T, const T&>; // copy assignment

    axiom Object_equality(T& x, T& y) {
        &x==&y => x==y; // identical objects have equal values
    }
    axiom Move_assign_effect(T x, T y, T& z) {
        x==y => (z=move(x))==y && can_destroy(x); // original is valid but unspecified
    }
    axiom Copy_assign_equality(T x, T& y) {
        (y = x) == x; // a copy is equal to the original
    }
}
```

The **Object_equality** axiom requires equality for identical objects (those having the same address). The **Move_assign_effect** and **Copy_assign_equality** axioms extend the semantics of move and copy construction to assignment. Note that the requirement of assignability implies that **const**-qualified types are not **Regular**. Furthermore, **volatile**-qualified types are not regular because they cannot satisfy the **Object_equality** axiom; the value of a **volatile** object may change unexpectedly. These design decisions are intentional. Objects can be **const**- or **volatile**-qualified to denote immutability or volatility, but that does not make their *value's* type immutable or volatile. Also, including assignment as a requirement for Regular types allows a greater degree of freedom for algorithm implementers. Not including assignability would mean that an algorithm using temporary storage (e.g., a variable) would be required state the additional requirement as part of its interface, leaking implementation details through the user interface.

We note that *Order* is not a requirement of *Regular* types. Although many regular types do admit a natural order, others do not (e.g., **complex<T>** and **thread**), hence the two concepts are distinct.

This definition is similar to those found in previous work by Dehnert and Stepanov [13] and also by Stepanov and McJones [34] except that the design is made modular in order to accommodate a broader range of fundamental notions. In particular, these basic concepts can be reused to define concepts expressing the requirements of **Value** and **Resource** types, both of which are closely related to the **Regular** types, but have restricted semantics. A **Value** represents pure (immutable) values in a program such as temporaries or constant expressions. **Value**s can be copy and move constructed, and compared, but not modified. A **Resource** is an object with limited availability such an **fstream**, or a **unique_ptr**. **Resource**s can be moved and compared, but not copied. Both **Value**s and **Resource**s may also be **Ordered**. We omit specific definitions of these concepts because they were not explicitly required by any templates in our survey; we only found requirements for **Regular** types.

By requiring essentially all types to be **Regular**, we greatly simplify interfaces and give a clear guideline to implementers of types. For example, a type that cannot be copied is unacceptable for our algorithms and so is a type with a copy operator that doesn't actually copy. Other design philosophies are possible; the STL's notion of type is value-oriented; an objected-oriented set of concepts would probably not have these definitions of copying and equality as part of their basis.

5.2 Type Abstractions

There are a small number of fundamental abstractions found in virtually all programs: *Boolean*, *Integral*, and *Real* types. In this section, we describe how we might define concepts for such abstractions. We save specific definitions for future work, pending further investigation and experimentation.

The STL traditionally relies on the bool type rather than a more abstract notion, but deriving a Boolean concept is straightforward. The *Boolean* concept describes a generalization of the **bool** type and its operations, including the ability to evaluate objects in Boolean evaluation contexts (e.g., an **if** statement). More precisely, a *Boolean* type is a *Regular*, Ordered type that can be operated on by logical operators as well as constructed over and converted to **bool** values. The **Boolean** concept would require constraints for logical operators (e.g., **Logical_and**) and be defined by the semantics of the **Boolean_algebra** axiom.

Other type abstractions can be found in the STL's numeric library, which is comprised of only six algorithms. Although we did not explicitly consider concepts in the numeric domain, we can speculate about the existence and definition of some concepts. A principle abstraction in the numeric domain is the concept of *Arithmetic* types, those that can be operated on by the arithmetic operators **+**, *****, **-**, and **/** with the usual semantics, which we suspect should be characterized as an *Integral Domain*. As such, all integers, real numbers, rational numbers, and complex numbers are Arithmetic types. Stronger definitions are possible; an *Integral* type is an *Arithmetic* type that also satisfies the semantical requirements of a *Euclidean Domain*. We note that *Matrices* are not *Arithmetic* because of non-commutative multiplication.

The semantics of these concepts can be defined as axioms on those types and their operators. Concepts describing *Groups*, *Rings*, and *Fields* for these types should be analogous to the definition of **Boolean_algebra** for **Boolean** types. We leave the exact specifications of these concepts as future work as a broader investigation of numeric algorithms and type representations is required.

5.3 Function Abstractions

Functions and function objects (functors) are only "almost regular" so we cannot define them in terms of the **Regular** concept. Unlike regular types, functions are not default constructible and function objects practically never have equality defined. Many solutions have been suggested such as a concept **Semiregular** or implicitly adding the missing operations to functions and function objects.

To complicate matters, functions have a second dimension of regularity determined by *application equality*, which states that a function applied to equal arguments yields equal results. This does not require functions to be pure (having no side effects), only that any side effects do not affect subsequent evaluations of the function on equal arguments. A third property used in the classification of functions and function objects is the homogeneity of argument types. We can define a number of mathematical properties for functions when the types of their arguments are the same.

To resolve these design problems, we define two static constraints for building functional abstractions: *Callable* and *Procedure*. The *Callable* constraint determines whether or not a type can be invoked as a function over a sequence of argument types. The *Procedure* constraint establishes the basic type requirements for all procedural and functional abstractions.

```
constraint Procedure<typename F, typename... Args> {
  requires constraint Constructible<F, F const&> // has copy construction
                   && Callable<F, Args...>; // can be called with Args...
  typename result_type = result_of<F(Args...)>::type;
}
```

Procedure types are both copy constructible and callable over a sequence of arguments. The variadic definition of this concept allows us to write a single constraint for functions of any *arity*. The **result_type** (the *codomain*) is deduced using **result_of**.

There are no restrictions on the argument types, result types or semantics of **Procedures**; they may, for example, modify non-local state. Consequently, we cannot specify meaningful semantics for all procedures, and so it is static constraint rather than a concept. A (random number) *Generator* is an example of a Procedure that takes no arguments and (probably) returns different values each time it is invoked.

A *Function* is a Procedure that returns a value and guarantees application equivalence and deterministic behavior. It is defined as:

```
concept Function<typename F, typename... Args> : Procedure<F, Args...> {
   requires constraint !Same<result_type, void>; // must return a value

   axiom Application_equality(F f, tuple<Args...> a, tuple<Args...> b) {
      (a==b) => (f(a...)==f(b...)); // equal arguments yield equal results
   }
}
```

The **Application_equality** axiom guarantees that functions called on equal arguments yield equal results. This behavior is also invariant over time, guaranteeing that the function always returns the same value for any equal arguments. The syntax **a...** denotes the expansion of a tuple into function arguments. The hash function parameter of unordered containers is an example of a **Function** requirement.

An *Operation* is a *Function* whose argument types are homogeneous and whose domain is the same as its codomain (result type):

```
concept Operation<typename F, Common... Args> : Function<F, Args...> {
    requires sizeof...(Args) != 0; // F must take at least one argument
    typename domain_type = common_type<Args?>::type;
    requires constraint Convertible<result_type, domain_type>;
}
```

From this definition, an **Operation** is a **Function** accepting a non-empty sequence of arguments that share a **Common** type (defined in Sect 6). The common type of the function's argument types is called the **domain_type**. The **result_type** (inherited indirectly from **Procedure**) must be convertible to the **domain_type**.

Note that the concept's requirements are not applied directly to **F** and its argument types. Instead, the concept defines a functional abstraction over **F** and the unified domain type. Semantics for **Operation**s are more easily defined when the domain and result type are interoperable. This allows us to use the **Operation** concept as the basis of algebraic concepts, which we have omitted in this design. We note that many of the function objects in the STL's functional library generate **Operation** models (e.g., **less**, and **logical_and**).

Parallel to the three functional abstractions *Procedure*, *Function*, and *Operation*, we define two predicate abstractions: *Predicate* and *Relation*. A *Predicate* is a *Function* whose result type can be converted to bool. **Predicate**s are required by a number of STL algorithms; for example, any algorithm ending in **_if** requires a **Predicate** (e.g., **find_-if**). This specification also matches the commonly accepted mathematical definition of a predicate and is a fundamental building block for other abstractions functional abstractions. A *Relation* is a binary *Predicate*. The axioms **Strict_weak_order** and **Strict_total_order** are semantic requirements on the **Relation** concept. These concepts are easily defined, and their semantics are well known.

5.4 Iterators

The distinction between concept and constraint has a substantial impact on the traditional iterator hierarchy [22]. We introduce a new general *Iterator* concept as the base of the iterator hierarchy. The input and output aspects of the previous hierarchy are relegated to constraints.

An *Iterator* is a *Regular* type that describes an abstraction that can "move" in a single direction using **++** (both pre- and post-increment) and whose referenced value can be accessed using unary *****. The concept places no semantic requirements on either the traversal operations or the dereferencing operation. In this way, the *Iterator* concept is not dissimilar from the ITERATOR design pattern [17], except syntactically. As with the previous designs, there are a number of associated types. Its definition follows:

```
concept Iterator<Regular Iter> {
    typename value_type  = iterator_traits<Iter>::value_type;
    Movable reference  = iterator_traits<Iter>::reference;
    Signed_int difference_type = iterator_traits<Iter>::difference_type;
    typename iterator_category  = iterator_traits<Iter>::iterator_category;

    Iter& Iter::operator++();    // prefix increment: move forward
    Dereferenceable Iter::operator++(int); // postfix increment
    reference Iter::operator*();  // dereference
}
```

Here, the pre-increment operator always returns a reference to itself (i.e., it yields an **Iterator**). In contrast, the result of the post-increment operator only needs to be **Dereferenceable**. The weakness of this requirement accommodates iterators that return a state-caching proxy when post-incremented (e.g., **ostream_iterator**). The **Movable** requirement on the reference type simply establishes a basis for accessing the result. This also effectively requires the result to be non-**void**.

The definition presented here omits requirements for an associated pointer type and for the availability of **->**: the arrow operator. The requirements for **->** are odd and conditionally dependent upon the iterator's value type [15]. In previous designs, the **->** requirement was handled as an intrinsic; that's most likely necessary.

Forward, *Bidirectional*, and *Random Access Iterators* have changed little in this design. These concepts refine the semantics of traversal for *Iterators*. However, *Input Iterators* and *Output Iterators* have been "demoted' to constraints (see below). A *Forward Iterator* is a multi-pass *Iterator* that abstracts the traversal patterns of singly-linked lists:

```
concept Forward_iterator<typename Iter> : Iterator<Iter> {
   requires constraint Convertible<iterator_category, forward_iterator_tag>;
   requires constraint Input_iterator<Iter>;

   Iter Iter::operator++(int); // postfix increment---strengthen Iterator's requirement

   axiom Multipass_equality(Iter i, Iter j) {
       (i == j) => (++i == ++j); // equal iterators are equal after moving
   }
```

```
    axiom Copy_preservation(Iter i) {
        (++Iter{i}, *i) == *i; // modifying a copy does not invalidate the original
    }
}
```

An **Input_iterator** only requires its reference type to be convertible to its value type. For a **Forward_iterator** this requirement strengthened so that, like the pre-increment operator, its post-increment must return an **Iterator**. The two axioms specify the semantic properties of multi-pass iterators: equivalent iterators will be equivalent after incrementing and incrementing a copy of an iterator does not invalidate the original.

Finally (and notably), **Forward_iterator**s are statically differentiated from **Input_-iterators** by checking convertibility of their iterator categories. This allows the compiler to automatically distinguish between models of the two concepts without requiring a concept map (as was needed in the C++0x design). Consider:

```
template<typename T, Allocator A>
class vector {
    template<Iterator Iter>
    vector(Iter first, Iter last) { // general version (uses only a single traversal)
        for( ; first != last; ++first)
            push_back(*first);
    }
    template<Forward_iterator Iter>
    vector(Iter first, Iter last) {
        resize(first, last); // traverse once to find size
        copy(first, last, begin()); // traverse again to copy
    }
};
```

The **vector**'s range constructor is optimized for **Forward_Iterators**. However, this is not just a performance issue. Selecting the wrong version leads to serious semantic problems. For example, invoking the second (multi-pass) version for an input stream iterator would cause the system to hang (wait forever for more input). With the automatic concept checking enabled by the **Convertible** requirement, we leverage the existing iterator classification to avoid this problem.

A *Bidirectional Iterator* is a *Forward Iterator* that can be moved in two directions (via **++** and **–**). It abstracts the notion of traversal for doubly linked lists:

```
concept Bidirectional_iterator<Iter> : Forward_iterator<Iter> {
    Iter& Iter::operator--(); // prefix decrement: move backwards
    Iter Iter:: operator--(int); // postfix decrement

    axiom Bidirectional(Iter i, Iter j) {
        i==j => --(++j)==i;
    }
}
```

A *Random Access Iterator* is an *Ordered Bidirectional Iterator* that can be moved multiple "jumps" in constant time; it generalizes the notion of pointers:

```
concept Random_access_iterator<Ordered Iter> : Bidirectional_iterator<Iter> {
    Iter& operator+=(Iter, difference_type);
    Iter operator+(Iter, difference_type);
    Iter operator+(difference_type, Iter);
    Iter& operator-=(Iter, difference_type);
    Iter operator-(Iter, difference_type);
    difference_type operator-(Iter, Iter); // distance
    reference operator[](difference_type n); // subscripting

    axiom Subscript_equality(Iter i, difference_type n) {
        i[n] == *(i + n); // subscripting is defined in terms of pointer arithmetic
    }
    axiom Distance_complexity(Iter i, Iter j) {
        runtime_complexity(i - j)==O(1); // the expression i-j must be constant time
    }
}
```

The requirements on the result type of the decrement operators are analogous to the requirements for increment operators of the **Forward_iterator** concept. The **Random_-access_iterator** concept requires a set of operations supporting random access (**+** and **-**). Of particular interest are the requirement on **Ordered** and the ability to compute the distance between two iterators (in constant time) via subtraction. Semantically, the **Subscript_equality** axiom defines the meaning of the subscript operator in terms of pointer arithmetic. The **Distance_complexity** axiom requires the computation of distance in constant time. Similar requirements must also be stated for random access addition and subtraction, but are omitted here. Here, **runtime_complexity** and **O** are intrinsics used to describe complexity requirement.

Input_iterator and **Output_iterator** are defined as constraints rather than concepts. To allow proxies, their conversion and assignment requirements are expressed in terms of a type "specified elsewhere." The C++0x design relies on associated types, which are expressed as member types. These are implicit parameters that cannot be universally deduced from the iterator. As is conventional, we rely on type traits to support this association. A constrained template or other concept must be responsible for supplying the external argument. However, our definition provides the obvious default case (the iterator's value type):

```
constraint Input_iterator<typename Iter, typename T = value_type<Iter>> {
    requires constraint Convertible<iterator_traits<Iter>::reference, T>;
}
constraint Output_iterator<typename Iter, typename T = value_type<Iter>&&> {
    requires constraint Assignable<iterator_traits<Iter>::reference, T>;
}
```

Here, we assume that **value_type<Iter>** is a template alias yielding the appropriate value type. The constraints for **Input_iterator** and **Output_iterator** support the ability to read an object of type **T** and write an object of type **T** respectively.

6 Constraints

In this section, we give an overview of constraints that have been described in previous sections. Many discussed thus far have direct corollaries in the C++ Standard Library type traits or are easily derived; they are basically a cleaned-up interface to what is already standard (and not counted as concepts in any design). Several that we have used in our implementation are described in Table 2.

Table 2. Type constraints

Constraint	*Definition*
Same<Args...>	Defined in terms of **is_same**
Common<Args..>	True if **common_type<Args...>** is valid
Derive<T, U>	**is_base_of<T, U>**
Convertible<T, U>	**is_convertible<T, U>**
Signed_int<T>	**is_signed<T, U>**

The **Same** constraint requires that all argument types be the same type. It is a variadic template that is defined recursively in terms of **is_same**. The **Common** constraint requires its argument types to all share a common type as defined by the language requirements of the conditional operator (**?:**). Finding the common type of a set of types implies that all types in that set can be implicitly converted to that type. This directly supports the definition of semantics on the common type of a function's arguments for the **Operation** concept. **Derived** and **Convertible** are trivially defined in terms of their corresponding type traits. We note that **Signed_int** is a constraint because its definition is closed: only built-in signed integral types satisfy the requirement (e.g., **int**, and **long**).

Constraints describing the syntax of construction, destruction, and assignment are given in Table 3, and constraints describing overloadable operators in Table 4. The constraints for construction, destruction, and assignment are trivially defined in terms of existing type traits. The operator constraints require the existence of overloadable operators for the specified type parameters and may specify conversion requirements on their result types. For binary operators, the second parameter defaults to the first so that, say, **Equal<T>** evaluates the existence of **operator==(T, T)**.

Table 3. Constraints for object initialization

Constraint	*Definition*
Destructible<T>	**is_destructible<T>**
Constructible<T, Args..>	**is_constructible<T, Args...>**
Assignable<T, U>	**is_assignable<T, U>**

The **Equal** and **Less** constraints require their results to be bool, which allows the constrained expression to be evaluated in a Boolean evaluation context. The other operators do not impose constraints on their result types because we cannot reasonably define universally applicable conversion requirements for all uses of those operators. In essence, these constraints are purely syntactic; any meaning must be imposed by some other concept or template.

Table 4. Constraints for overloadable operators

Constraint	*Definition*
Equal<T, U=T>	**bool operator==(T, U)**
Less<T, U=T>	**bool operator<(T, U)**
Logical_and<T, U=T>	**auto operator&&(T, U)**
Logical_or<T, U=T>	**auto operator||(T, U)**
Logical_not<T>	**auto operator!(T)**
Derefernce<T>	**auto operator*(T)**

7 Implementation and Validation

We implemented the described concepts and traits by building a custom library of constraint classes in C++11. Our approach blends traditional techniques for implementing constraint classes [31] with template metaprogramming facilities [1]. The result is a lightweight concept emulation library that can be used to statically enforce constraints on template parameters and supports concept overloading using concept-controlled polymorphism (**enable_if**) [23]. The library is implemented as a core component of the Origin Libraries [38].

To simplify experimentation, the library does not differentiate between concepts and constraints except through naming conventions. Concepts names in the implementation are written **cConcept**, constraint names are written **tConstraint** ("t" stands for "type property") and axioms, **aAxiom**. This naming convention is chosen for the implementation so that it will not collide with "real" concepts when defined with language support. For example, the **Constructible** constraint described in 6 is implemented in Origin as **tConstructible**:

```
template<typename T, typename... Args>
struct tConstructible {
  tConstructible() { auto p = constraints; }

  static void constraints(Args... args) {
    T{forward<Args>(args)...}; // use pattern for copy construction
  }

  typedef tuple<std::is_constructible<T, Args...>> requirements;
  typedef typename requires_all<requirements>::type type;
  static constexpr bool value = type::value;
};
```

tConstructible is a constraint because no sensible semantics can be defined for construction, beyond what the language already guarantees for constructors. The **tConstructible** constructor is responsible for instantiating the constraints function, which contains the use patterns for the concept, which are similar to those introduced in [15]. The function parameters of the constraints function introduce objects, which simplifies the writing of use patterns.

The type and value members satisfy the requirements of a type trait or Boolean metafunction. These members, especially value, are used to reason about the type at compile time without causing compiler errors. This is used to select between overloads based on modeled satisfied requirements using **enable_if** [23].

All concepts in the library are automatically checked. Implementing a concept library that requires users to write explicit concept maps would require us to do so for every data structure tested. That approach is not needed for the STL and, in our opinion, does not scale well. Axioms are not (and cannot be) checked by the compiler so for our validation we treat them as comments.

Function template requirements are specified by explicit constructions of temporary objects. For example:

```
template<typename T>
T const& min(T const& x, T const& y) {
    cOrdered<T>{};  // requires that T has operators <, >, <=, and >=
    return y < x ? y : x;
}
```

Here **cOrdered<T>** denotes a requirement on the template parameter **T**. Instantiating the algorithm entails instantiating the **cOrdered<T>** constructor and its nested requirements. Compilation terminates if a constraint class is instantiated with template arguments that do not satisfy the required use patterns.

For class templates, the requirements are specified as base classes. For example:

```
template<typename T> class Vector : private cRegular<T> { /* ... */ };
```

This ensures that the compiler instantiates the concept checks when constructing objects of the constrained class. Requirements within constraint classes are written in exactly the same way: explicit construction is used in conjunction with use patterns, and inheritance is used to emulate concept refinement.

There are no memory or performance costs induced by the use of Origin's constraint classes. Constraint instances are optimized out of the generated code either through dead-code elimination or the empty base optimization. Compile times can be increased marginally but are no worse than using any other concept checking library.

We applied the constraint classes to a subset of the Clang and GCC implementations of the Standard Library: the functional, algorithm, and iterator components (the STL). Class and function template constraints were written for each data structure and algorithm exported by those components. We iteratively refined both the concepts and the constraints as required by the limits of the experiment.

We use the libraries' test suites (just over 9,000 programs at the time of writing) to check the correctness of the concepts. A more substantial validation of our design could be achieved by compiling a large number of C++ applications against the modified libraries, but the test suite cover a sufficient number of instantiations so we are confident in the design.

Test suite failures generally indicated overly strict constraints on a type or algorithm. In some cases, such failures also indicated what we perceive as problems with the original conceptual specification for the library. In such cases, these failures are due to the representation of irregular types as legitimate concepts. For example, strict output iterators such as **ostream_iterator** are *Iterators* in principle but are neither default constructible nor equality comparable. We modified these irregular cases so they would model the required concepts. There are only four such iterators in the STL, and they are easily adapted to model our proposed concepts.

8 Conclusions

We studied concept design rather than language design for expressing concepts with the aim of bringing empirical evidence to the center of language design discussions. We found that explicitly differentiating between concepts and constraints based on semantic requirements (axioms) improved our analyses and clarified long-standing “dark corners” of the STL design. It led to spectacular simplification and a dramatic reduction in the number of concepts needed to describe the STL interfaces (14 rather than 108). The STL interfaces were already considered well understood after more than a decade’s use and much analysis, so we conclude that our concept design technique is nowhere near as obvious as it seems in retrospect. Our technique is rooted in classical algebraic theory, so we further conjecture that it will be very widely applicable.

Our conclusions on language design for concepts are, as we expected them to be, very tentative. However, we have demonstrated a central role for axioms, a feature that was widely conjectured to be unnecessary during the C++0x design. Beyond that, we found constraints classes so expressive and manageable in our implementation that we want to re-examine the use-pattern approach for expressing syntactic requirements.

We continue to investigate concepts for the C++ Standard Library by broadening our conceptual analysis to cover numeric and scientific computing domains and containers. With Origin, we are pursuing concepts related to heaps and graphs [30]. Exploring conceptual designs in different domains supports language design by addressing a broader set of use cases. In particular, we plan to examine uses of concepts in algorithm specifications with the aim of simplifying such specifications.

Concepts are abstract, general, and meaningful. Consequently, they are unlikely to be specific to a specific library. As concepts mature, they become a repository of fundamental domain knowledge. Thus, we expect concepts to cross library boundaries to become more widely useful. We expect that our exploration of the definition and use of concepts will decrease the total number of concepts (among all libraries) while improving their quality and utility.

Acknowledgements

Thanks to Matt Austern, Paul McJones, Gabriel Dos Reis, and Alex Stepanov for comments that led to major improvements and will be the basis for further work. This project was partially supported by NSF grants A3350-32525-CS and A0040-32525-CS and Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ In-Depth, Addison Wesley (2004)
2. Austern, M.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Welsey Longman, Boston, Massachusetts, 7th edn. (1998)

3. Austern, M., Stroustrup, B., Thorup, M., Wilkinson, J.: Untangling the Balancing and Searching of Balanced Binary Search Trees. Software: Practice and Experience 33(13), 1273–1298 (2003)
4. Bagge, A.H., David, V., Haveraaen, M.: The Axioms Strike Back: Testing with Concepts and Axioms in C++. In: 8th International Conference on Generative Programming and Component Engineering (GPCE'09). pp. 15–24. Denver, Colorado (2010)
5. Becker, P.: Working Draft, Standard for the Programming Language C++. Tech. Rep. N2914, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2009)
6. Bernardy, J.P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A Comparison of C++ Concepts and Haskell Type Classes. In: Workshop on Generic Programming (WGP'08). pp. 37–48. Victoria, Canada (2008)
7. Bernardy, J.P., Jansson, P., Zalewski, M., Schupp, S.: Generic Programming with C++ Concepts and Haskell Type Classes–A Comparison. Journal of Functional Programming 20(3-4), 271–302 (2010)
8. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: 13th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98). pp. 183–200. Vancouver, Canada (1998)
9. Davenport, J.H., Gianni, P.M., Trager, B.M.: Scratchpad's View of Algebra II: A Categorical View of Factorization. In: International Symposium on Symbolic and Algebraic Computation (ISSAC'91). pp. 32–38. Bonn, Germany (1991)
10. Davenport, J.H., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer (1992)
11. Davenport, J.H., Trager, B.M.: Scratchpad's View of Algebra I: Basic Commutative Algebra. In: International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'90). pp. 40–54. Capri, Italy (1990)
12. David, V.: Concepts as Syntactic Sugar. In: 9th International Working Conference on Source Code Analysis and Manipulation (SCAM'09). pp. 147–156. Alberta, Canada (2009)
13. Dehnert, J., Stepanov, A.: Fundamentals of Generic Programming. In: International Seminar on Generic Programming. vol. 1766, pp. 1–11. Springer, Dagstuhl Castle, Germany (1998)
14. Dos Reis, G.: Personal Communication (Oct 2010)
15. Dos Reis, G., Stroustrup, B.: Specifying C++ Concepts. In: 33rd Symposium on Principles of Programming Languages (POPL'06). pp. 295–308. Charleston, South Carolina (2006)
16. Dos Reis, G., Stroustrup, B., Merideth, A.: Axioms: Semantics Aspects of C++ Concepts. Tech. Rep. N2887, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2009)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
18. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An Extended Comparative Study of Language Support for Generic Programming. Journal of Functional Programming 17, 145–205 (2007)
19. Gregor, D.: ConceptGCC (2008), http://www.generic-programming.org/software/ConceptGCC/
20. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06). pp. 291–310. Portland, Oregon (2006)
21. Hewlett-Packard: Standard Template Library Programmer's Guide (1994), http://www.sgi.com/tech/stl/index.html
22. International Organization for Standards: International Standard ISO/IEC 14882. Programming Languages — C++ (2003)
23. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-Controlled Polymorphism. In: 2nd International Conference on Generative Programming and Component Engineering (GPCE'03). pp. 228–244. Erfurt, Germany (2003)
24. Jones, M.P.: Type Classes with Functional Dependencies. In: 9th European Symposium on Programming (ESOP'00). pp. 230–244. Berlin, Germany (2000)

25. Kennedy, A., Syme, D.: Design and Implementation of Generics for the .NET Common Language Runtime. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01). pp. 1–12. Snowbird, Utah (2001)
26. Meyer, B.: Eiffel : The Language. Prentice-Hall (1991)
27. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML - Revised. The MIT Press (1997)
28. Musser, D., Stepanov, A.: Algorithm-oriented Generic Libraries. Software: Practice and Experience 24(7), 623–642 (1994)
29. Pirkelbauer, P., Dechev, D., Stroustrup, B.: Support for the evolution of C++ generic functions. In: 3rd International Conference on Software Language Engineering (SLE'10). pp. 123–142. Eindhoven, the Netherlands (2010)
30. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2001)
31. Siek, J., Lumsdaine, A.: Concept Checking: Binding Parametric Polymorphism in C++. In: 1st Workshop on C++ Template Programming. Erfurt, Germany (2000)
32. Siek, J., Lumsdaine, A.: Essential Language Support for Generic Programming. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). pp. 73–84. Chicago, Illinois (2005)
33. Siek, J., Lumsdaine, A.: Language Requirements for Large-scale Generic Libraries. In: 4th International Conference on Generative Programming and Component Engineering (GPCE'05). pp. 405–421. Tallinn, Estonia (2005)
34. Stepanov, A., McJones, P.: Elements of Programming. Addison Wesley, Boston, Massachusetts (2009)
35. Stroustrup, B.: Parameterized Types for C++. Computing Systems 2(1), 55–85 (1989)
36. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)
37. Stroustrup, B.: "New" Value Terminology (2010), http://www2.research.att.com/ bs/terminology.pdf
38. Sutton, A.: Origin C++0x Libraries (2011), http://code.google.com/p/origin
39. Sutton, A., Holeman, R., Maletic, J.I.: Identification of Idiom Usage in C++ Generic Libraries. In: 18th International Conference on Program Comprehension (ICPC'10). pp. 160–169. Braga, Portugal (2010)
40. Sutton, A., Maletic, J.I.: Automatically Identifying C++0x Concepts in Function Templates. In: 24th International Conference on Software Maintenance (ICSM'04). pp. 57–66. Beijing, China (2008)
41. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding Wildcards to the Java Programming Language. In: ACM Symposium on Applied Computing (SAC'04). pp. 1289–1296. Nicosia, Cyprus (2004)
42. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (1995)