Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Introduction

For over three decades, Computer Science has been pursuing the goal of software reuse. There have been a variety of approaches, none of them as successful as similar attempts in other engineering disciplines. Generic programming [MuSt89] offers an opportunity to achieve what these other approaches have not. It is based on the principle that software can be decomposed into components which make only minimal assumptions about other components, allowing maximum flexibility in composition.

Reuse has been successful in the area of libraries. Examples include system interface libraries such as Unix [KeMa81], numeric libraries such as Lapack [Demmel89], and window management libraries such as X [ScGe92]. However, these libraries have the characteristic that they use fully specified interfaces that support a pre-determined set of types, and make little or no attempt to operate on arbitrary user types. These fully specified interfaces have been instrumental in encouraging use, by allowing users to comfortably use them with full knowledge of how they will behave. Paradoxically, however, this strength turns into a weakness: for example, while people can use the C library routine `sqrt` on any machine with predictable results, they cannot use it when a new type like quad-precision floating point is added. In order to make progress, we must overcome this limitation.

Generic programming recognizes that dramatic productivity improvements must come from reuse without modification, as with the successful libraries. Breadth of use, however, must come from the separation of underlying data types, data structures, and algorithms, allowing users to combine components of each sort from either the library or their own code. Accomplishing this requires more than just simple, abstract interfaces – it requires that a wide variety of components share the same interface so that they can be substituted for one another. It is vital that we go beyond the old library model of reusing identical interfaces with pre-determined types, to one which identifies the minimal requirements on interfaces and allows reuse by similar interfaces which meet those requirements but may differ quite widely otherwise. Sharing similar interfaces across a wide variety of components requires careful identification and abstraction of the patterns of use in many programs, as well as development of techniques for effectively mapping one interface to another.

We call the set of axioms satisfied by a data type and a set of operations on it a ***concept***. Examples of concepts might be an integer data type with an addition operation satisfying the usual axioms; or a list of data objects with a first element, an iterator for traversing the list, and a test for identifying the end of the list. The critical insight which produced generic programming is that highly reusable components must be programmed assuming a minimal collection of such concepts, and that the

concepts used must match as wide a variety of concrete program structures as possible. Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm – it requires identifying the common requirements of a broad collection of similar components. The final requirement is that we accomplish this without sacrificing performance relative to programming with concrete structures. A good generic library becomes a repository of highly efficient data structures and algorithms, based on a small number of broadly useful concepts, such that a library user can combine them and his own components in a wide variety of ways.

The C++ Standard Template Library (STL) [StLe95] is the first extensive instance of this paradigm in wide use. It provides a variety of data containers and algorithms which can be applied to either built-in or user types, and successfully allows their composition. It achieves the performance objectives by using the C++ template mechanism to tailor concept references to the underlying concrete structures at compile time instead of resolving generality at runtime. However, it must be extended far beyond its current domain in order to achieve full industrialization of software development. This requires identifying the principles which have made STL successful

In our search for these principles, we start by placing generic programming in an historic progression. The first step was a generalized machine architecture, exemplified by the IBM 360, based on a uniform view of the machine memory as a sequence of bytes, referenced by uniform addresses (pointers) independent of the type of data being referenced. The next step was the C programming language [KeRi78], which was effectively a generalized machine language for such architectures, providing composite data types to model objects in memory, and pointers as identifiers for such memory objects with operations (dereferencing and increment/decrement) that were uniform across types.

The C++ programming language [Stroustrup97] was the next step. It allows us to generalize the use of C syntax, applying the built-in operators to user types as well, using class definitions, operator overloading, and templates. The final step in this progression is generic programming, which generalizes the semantics of C++ in addition to its syntax. If we hope to reuse code containing references to the standard C++ operators, and apply it to both built-in and user types, we must extend the semantics as well as the syntax of the standard operators to user types. That is, the standard operators must be understood to implement well-defined concepts with uniform axioms rather than arbitrary functions. A key aspect of this is generalizing C's pointer model of memory to allow uniform and efficient traversal of more general data structures than simple arrays, accomplished in the STL by its iterator concepts.

This extension of C built-in operator semantics is the key to at least part of the STL's success in finding widely applicable concepts. The development of built-in types and operators on them in programming languages over the years has led to relatively consistent definitions which match both programmer intuition and our underlying mathematical understanding. Therefore, concepts which match the semantics of built-in types and operators provide an excellent foundation for generic programming.

In this paper, we will investigate some of the implications of extending built-in operator semantics to user-defined types. We will introduce the idea of a regular type as a type behaving like the built-in types, and will investigate how several of the built-in operators should behave when applied to such user-defined types.

Regular types

The C++ programming language allows the use of built-in type operator syntax for user-defined types. This allows us, as programmers, to make our user-defined types *look* like built-in types. Since we wish to extend semantics as well as syntax from built-in types to user types, we introduce the idea of a ***regular type***, which matches the built-in type semantics, thereby making our user-defined types *behave* like built-in types as well.

The built-in types in C++ vary substantially in the number and semantics of the built-in operators they support, so this is far from a rigorous definition. However, we observe that there is a core set of built-in operators which are defined for all built-in types. These are the constructors, destructors, assignment and equality operators, and ordering operators. Figure 1 gives their syntax in C++. We use C++ as our basis, although our principles are not language-specific.

Fig. 1. Fundamental Operations on Type T

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
Ordering, e.g.	`a < b`

The first four fundamental operations in Figure 1 have default definitions for all built-in types and structures in C/C++. The remaining ones have default definitions only for built-in types, but could be defined reasonably for structures as well. Equality and inequality could be defined component-wise, and the ordering operators could be defined with a lexicographic order, using the component orderings recursively. (The ordering case is interesting. C++ does not define total ordering operations on pointer types, and it is not possible to define efficient operations which would produce the same results for all implementations. Even without such a portability guarantee, however, there are applications for which universal availability

of efficient operators is useful. But this subject is beyond the scope of the current paper.)

As we shall see below, the default definitions of these operations are not always adequate, but their semantics when applied to the built-in types provide the model we want for our more general requirements on regular types. In the remainder of this paper, we shall attempt to identify the essential semantics of these operations, which we call the ***fundamental operations*** on a type T. By doing so, we will fill in some of the details of a precise definition of regular types.

Copy, Assignment, and Equality

First, we consider the interactions among the copy constructor, assignment, and equality operators. These operations are central to our understanding of a programming language. What can we say about them?

(1) `T a = b; assert(a==b);`

Our first axiom simply says that after constructing a new object a of type T, with an initial value copied from object b, we expect objects a and b to be equal. Furthermore, we expect this construction to be equivalent to constructing a with a default constructor and then assigning the value of b to it:

(2) `T a; a = b;` ⇔ `T a = b;`

So far, our axioms would be satisfied equally well by a language like C++ which copies values on assignment, or by a language like Lisp which simply copies addresses leaving both names pointing to the same copy of the value. Our next axiom says that we intend the C++ copy semantics:

(3) `T a = c; T b = c; a = d; assert(b==c);`

Here, after assigning the same value c to both a and b, we expect to be able to modify a without changing the value of b. In fact, we want an even stronger condition. If `zap` is an operation which always changes the value of its operand, we expect the following to hold:

(4) `T a = c; T b = c; zap(a); assert(b==c && a!=b);`

That is, b and c do not continue to be equal simply because their values were changed along with a's, but rather because changing a's value did not change theirs.

As an example of the power of these axioms, let us considered a small example. Figure 2a contains a simple template function for swapping two values, and figure 2b contains a code fragment which specifies he expected semantics of that swap function.

Fig. 2. Generic swap function

***swap* function**	***Swap* specification**

`template <class T>`
`void swap (T& x, T& y) {`
`  T tmp = x;`
`  x = y;`
`  y = tmp;`
`}`
	`T a, b;`
	`…`
	`T old_a = a;`
	`T old_b = b;`
	`Swap (a, b);`
	`Assert (a == old_b);`
	`Assert (b == old_a);`

Now, if we substitute the body of the swap function for its call in the specification, and apply the axioms above along with the usual axioms of quality (in particular transitivity), we get the expected result, as shown in figure 3.

Fig. 3. Validation of swap function

swap	**Assertions**
`T a,b;`	
`T old_a = a;`	`// a == old_a`
`T old_b = b;`	`// b == old_b`
`swap (T& a, T& b) {`	
`  T tmp = a;`	`// tmp==a && tmp==old_a`
`  a = b;`	`// a==b && a==old_b`
`  b = tmp;`	`// b==tmp && b==old_a`
`}`	`// b==old_a && a==old_b`

The axioms above provide us with a reasonable characterization of the semantics of copy constructors and assignment operators in terms of the equality of their operands after they are applied. A copy constructor creates a new object equal to the object from which it is copied; and assignment copies its right-hand side operand to its left-hand side object, leaving their values equal.

However, we do not yet have a satisfactory definition of the equality of two objects of a regular type. We shall investigate this question in the next section.

Equality of Regular Types

Some writers have defined equality as a relation that is reflexive, symmetric, and transitive. While these are certainly attributes of equality, they do not constitute a definition. To see this, simply consider a hypothetical equality function which always returns true. It has these three attributes, but certainly does not satisfy our expectations for an equality operator. We must look further.

Logicians might define equality via the following equivalence:

$$\texttt{x == y} \iff \forall \text{ predicate } \texttt{P, P(x) == P(y)}$$

That is, two values are equal if and only if no matter what predicate one applies to them, one gets the same result. This appeals to our intuition, but it turns out to have significant practical problems. One direction of the equivalence:

$$\texttt{x == y} \implies \forall \text{ predicate } \texttt{P, P(x) == P(y)}$$

is useful, provided that we understand the predicates P for which it holds. We shall return to this question later. The other direction, however:

$$\forall \text{ predicate } \texttt{P, P(x) == P(y)} \implies \texttt{x == y}$$

is useless, even if P is restricted to well behaved predicates, for the simple reason that there are far too many predicates P to make this a useful basis for deciding equality. Again, we must look further.

Fortunately, our computer hardware generally defines an equality relation on the built-in types which it implements efficiently. This equality relation is normally bitwise equality (although there are sometimes minor deviations like distinct positive and negative zero representations). Starting from this basis, there is a natural default equality for types composed of simpler types, i.e. equality of corresponding parts of the composite objects. (Although this definition is natural, neither C nor C++ provides a default equality operator for composite types.) While this definition is appealing for arbitrary data structures, we must resolve several questions.

In order to apply this definition to build an equality operator for a composite type from the equality operators on the types of its parts, we must identify its parts. Intuitively, they are the (data) members of a struct, but this is still not sufficient.

First, a C/C++ struct cannot represent all objects of interest. Specifically, it cannot represent an object of variable size. (This design decision was made because allowing variable size types would not allow the creation of arrays of those types with efficient access.) As a result, objects which are naturally variable sized must be constructed in C++ out of multiple simple structs, connected by pointers. In such cases, we say that the object has ***remote*** parts. For such objects, the equality operator must compare the remote parts of two objects rather than the pointers to them, since we would not like objects with equal remote parts to compare unequal simply because the remote parts were in different memory and their addresses were unequal.

The next subtle problem in identifying the parts of composite objects is that such objects sometimes contain components which are not essential to our concept of value. A good example of this situation is a struct which contains a count of the

number of pointers which reference it, perhaps for memory management purposes. We do not view such a reference count component to be part of the value of the object, and would not want otherwise equal objects to compare unequal simply because of unequal reference counts. Our second caveat then is that an equality operator should ignore inessential components.

The final problem relates to the question of where one object ends and another begins. In the physical world, we would think of the legs, seat, back, and arms, of a chair as being parts of the chair – the chair would be very different without them. However, we would not think of a person or dog sitting in the chair as a part of the chair, even though it is closely associated with the chair, at least for a time. Similarly if we were writing a program to produce a graphics display of a scene containing a chair, we might represent the chair as a struct containing pointers to remote parts (i.e. other structs) for its legs, seat, etc. However, even if we were to keep track of who or what was sitting in the chair by keeping a pointer to that other object in the chair object, we should probably not consider the other object to be part of the chair. That is, some components of composite objects reflect relationships between objects, and should not be considered as parts for equality testing purposes.

These observations leave us with a definition of equality which is workable in practice, although it still leaves room for judgment:

Definition: Two objects are equal if their corresponding parts are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

Once we have identified the parts of an object which must be tested for equality, we know from the earlier discussion that at least those parts must be copied by copy constructors or assignment operators. In particular, these operators must make copies of remote parts, rather than simply copying the pointers to them.

Now let us return to part of the logicians' definition of equality. Recall that we would like the following statement to be true:

`x == y` $\Rightarrow \forall$ "reasonable" function `foo`, `foo(x)==foo(y)`

It is necessary to limit our expectations to some subset of possible functions `foo`. For instance, this statement will not hold for the "address-of" function applied to distinct objects with equal values, nor will it hold for any other function which distinguishes between individual objects rather than between their values. Considering an example will demonstrate some of the challenges facing a designer of generic components.

Most of us would intuitively assume that a visible accessor function, that is a public function which returns the value of some component of a composite type, would be a reasonable function which should satisfy the above condition. However, that assumption constrains the combination of the equality operator definition and the choice of the visible parts of an object. To see how, suppose that we define a rational number object as a pair (p,q) representing its numerator and denominator. Given such

an object type, we cannot defined equality mathematically and still allow p and q to be visible parts, since doing so would yield the following incorrect assumption:

```
r1 == r2  ⇒  r1.p == r2.p
(1,2) == (2,4)  ⇒  1 == 2
```

Faced with this situation, several reasonable design decisions are possible which preserve our intuition. First, we could avoid defining an equality operator (perhaps defining an `equiv` function with the mathematical definition instead). Second, we could avoid making p and q visible parts of our rational number type. Finally, we could require that any rational number represented by this type is always in reduced form, i.e. its numerator and denominator have no common divisors.

Optimization

It is fair to ask why all of these details are important. After all, we can always take an arbitrary type definition with a sufficiently extensive set of operations defined on it, and write programs which use it effectively by following its own usage expectations. Most software development has operated this way to date, hand crafting each new component type to use the features exported by the types on which it depends, and exporting features designed to simplify its own implementation or that of known clients.

Generic programming, however, changes the rules substantially. If we are to succeed in producing widely reusable components, idiosyncratic interfaces are no longer usable. A component programmer must be able to make some fundamental assumptions about the interfaces she uses, without ever seeing their implementations or even imagining their applications. Similarly, her eventual users must provide the types implementing those interfaces, and if the same types are to interface with a variety of generic components, the interfaces must be consistent with one another.

The operations we have discussed here, equality and copy, are central because they are used by virtually all programs. They are also critically important because they are the basis of many optimizations which inherently depend upon knowing that copies create equal values, while not affecting the values of objects not involved in the copy. Such optimizations include, for example, common subexpression elimination, constant and copy propagation, and loop-invariant code hoisting and sinking. These are routinely applied today by optimizing compilers to operations on values of built-in types. Compilers do not generally apply them to operations on user types because language specifications do not place the restrictions we have described on the operations of those types.

However, users do apply such optimizations by hand. They often do so without thinking because they intuitively expect the conditions to apply. If they are to produce efficient generic components without seeing the underlying type definitions, they must be able to make the assumptions which allow such optimizations. Our

axioms, then, are necessary to allow users to reliably make the optimizations commonly made both by optimizing compilers and by optimizing programmers.

Ultimately, we would like compilers to be able to perform such optimizations at a high semantic level as well as they do at the built-in type level. This will require more formal adherence to the axioms we have described for the fundamental operations. But we would like to go further. Specifically, let us return to the equality principle mentioned above:

`x == y` $\Rightarrow \forall$ "reasonable" function `foo`, `foo(x)==foo(y)`

Again, what is a reasonable function? For optimization purposes, there are several classes of functions we would like to capture. First are the standard operators on built-in types that do not have side effects, for example a+b, c-d, or p%q. Second are the visible member accesses, e.g. s.first or c->imaginary. A third class is the well-known pure functions, e.g. abs(x), sqrt(y), and cos(z). Knowledge of most of these could be built into compilers if we made the appropriate restrictions on the user definitions of them. However, there are many more possibilities among arbitrary functions defined by users. Compilers cannot identify them all without assistance, both because the compiler cannot always see the function definitions, and because the compiler cannot make the necessary distinctions between essential and inessential parts or between pointers to remote parts or to related objects. The ultimate solution, then, must be to identify the important attributes, and allow programmers to specify them explicitly. This is an important language design issue, but is beyond the scope of this paper.

Complexity

It is often claimed that complexity is only an attribute of an implementation, and not properly part of component specification. This is wrong, and becomes more so with generic components. Users (and algorithms) make basic assumptions about operator complexity, and make decisions about the data structures or algorithms they will use based on those assumptions. Consider several examples:

- We expect the push and pop operations on a stack to required amortized constant time. If this expectation were not met, we would often use a different data structure, or perhaps implement a stack explicitly based on another data structure known to behave that way (such as an STL vector). This means that a stack implementation which does a re-allocation and copy whenever the stack grows is not just a poor implementation – it is an unacceptable implementation.
- We expect an implementation of character strings to have a copy constructor which is linear in the length of the string being copied. A well-known C++ standard library implementation contains a string copy constructor which is quadratic, requiring hours to copy a million-character string on a large server. Obviously, such an implementation is unusable for large strings.
- In the C++ STL, it would be possible for bidirectional iterators to support the random access iterator interface, i.e. providing operations to step through the data by

more than 1 element at a time. However, it is important to keep them distinct – the best algorithms for some functions (e.g. rotate or random shuffle) differ dramatically for bidirectional and random access iterators.

For the fundamental operations, users also have intuitive expectations of complexity. For regular types, we therefore require that constructors, destructors, and assignment operators be linear (average-case) in the ***area*** (i.e. the total size of all parts) of the object involved. Similarly, we require that the equality operator have linear worst-case complexity. (The average-case complexity of equality is typically nearly constant, since unequal objects tend to test unequal in an early part.)

Summary

In this paper, we have investigated several of the fundamental operations on built-in types in C++, and identified characteristics they should have when applied to user-defined types. This process is central to defining broadly applicable concepts which can enable generic programming to produce components which can be reused with a wide variety of built-in and user-defined types. We believe, based on the success of the C++ STL, that this scientific approach of observing widespread commonality in existing programs and then axiomatizing its properties consistent with existing programming and mathematical practice, holds promise that we will ultimately achieve the elusive goal of widespread software reuse.

Bibliography

[Demmel89] J. Demmel, *LAPACK: A portable linear algebra library for supercomputers*, Proc. of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design, December 1989.

[KeMS88] Aaron Kershenbaum, David R. Musser, and Alexander A. Stepanov, *Higher-Order Imperative Programming*, Computer Science Technical Report 88-10, Rensselaer Polytechnic Institute, April 1988.

[KeMa81] Brian W. Kernighan and John R. Mashey, *The Unix Programming Environment*, Computer 14(4), 1981, pp. 12-24.

[KeRi78] Brian W. Kernighan and Dennis M. Ritchie, **The C Programming Language**, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[MuSt89] David R. Musser, and Alexander A. Stepanov, *Generic Programming*, in P. Gianni, ed., **Symbolic and Algebraic Computation: International Symposium ISSAC 1988**, Lecture Notes in Compuber Science v. 38, Springer-Verlag, Berlin, 1989, pp. 13-25.

[ScGe92] Robert W. Scheifler and James Gettys, **X Window System**, 3rd Ed., Digital Press, 1992.

[StLe95] Alexander Stepanov and Meng Lee, *The Standard Template Library*, Tech. Report HPL-95-11, HP Laboratories, November 1995.

[Stroustrup97] Bjarne Stroustrup, **The C++ Programming Language**, 3rd Ed., Addison-Wesley, Reading, MA, 1997.