Foreword

When I first looked at this book, I felt envious. After all, what led me to the discovery of generic programming was the desire to build a library like the Boost Graph Library (BGL). In 1984 I joined the faculty of Polytechnic University in Brooklyn with some vague ideas about building libraries of software components. Well, to tell you the truth that was my secondary interest—my real interest at that time was to construct formal underpinnings of natural language, something like Aristotle's Organon, but more complete and formal. I was probably the only assistant professor in any Electrical Engineering or Computer Science department who meant to obtain tenure through careful study of Aristotle's Categories. Interestingly enough, the design of the Standard Template Library (STL)—in particular the underlying ontology of objects—is based on my realization that the whole-part relation is a fundamental relation that describes the real world and that it is not at all similar to the element-set relation familiar to us from set theory. Real objects do not share parts: my leg is nobody else's leg. STL containers are like that: two containers do not share parts. There are operations like `std::list::splice` that move parts from one container to another; they are similar to organ transplant: my kidney is mine until it is spliced into somebody else.

In any case, I was firmly convinced that software components should be functional in nature and based on John Backus's FP system. The only novel intuition was that functions should be associated with some axioms: for example, the “Russian peasant algorithm” that allows one to compute the *n*th power in *O* (log *n*) steps is defined for any object that has an associative binary operation defined on it. In other words, I believed that algorithms should be associated with what we now call concepts (see §2.3 of this book), but what I called structure types and what type-theorists call *multi-sorted algebras*.

It was my great luck that Polytechnic had a remarkable person on its faculty, Aaron Kershenbaum, who combined deep knowledge of graph algorithms with an unusual desire to implement them. Aaron saw potential in my attempts to decompose programs into simple primitives, and spent a lot of time teaching me graph algorithms and working with me on implementing them. He also showed me that there were some fundamental things that cannot be done functionally without prohibitive change in the complexity. Although it was often possible for me to implement linear time algorithms functionally without changing the asymptotic complexity, it was impossible in practice to implement logarithmic time algorithms without making them linear. In particular, Aaron explained to me why priority queues were so important for many graph algorithms (and he was well qualified to do so: Knuth in his Stanford GraphBase book [22] attributes the discovery of how to apply binary heaps to Prim's and Dijkstra's algorithms to Aaron).

It was a moment of great joy when we were able to produce Prim's and Dijkstra's algorithms as two instances of the same generic—we called it “high-order” then—algorithm. It is quite remarkable how close BGL code is to what we had (see, for example, a footnote to §13.4.2). The following code in Scheme shows how the two algorithms were implemented in terms of the same higher-order algorithm. The only difference is in how distance values are combined: using addition for Dijkstra's and by selecting the second operand for Prim's.

```
(define dijkstra
  (make-scan-based-algorithm-with-mark
    make-heap-with-membership-and-values + > ))

(define prim
  (make-scan-based-algorithm-with-mark
    make-heap-with-membership-and-values (lambda (x y) y) > ))
```

It took me a long time—almost 10 years—to find a language in which this style of programming could be *effectively* realized. I finally found C++, which enabled me to produce something that people could use. Moreover, C++ greatly influenced my design by providing a crisp C-based machine model. The features of C++ that enabled STL are templates and overloading.

I often hear people attacking C++ overloading, and, as is true with most good mechanisms, overloading can be misused. But it is an essential mechanism for the development of useful abstractions. If we look at mathematics, it has been greatly driven by overloading. Extensions of a notion of numbers from natural numbers to integers, to rational numbers, to Gaussian integers, to p-adic numbers, etc, are examples of overloading. One can easily guess things without knowing exact definitions. If I see an expression that uses both addition and multiplication, I assume distributivity. If I see less-than and addition, I assume that if $a > b$ then $a + c > b + c$ (I seldom add uncountable cardinals). Overloading allows us to carry knowledge from one type to another.

It is important to understand that one can write generic algorithms just with overloading, without templates: it does, however, require a lot of typing. That is, for every class that satisfies, say, random access iterator requirements, one has to define all the relevant algorithms by hand. It is tedious, but can be done (only signatures would need to be defined: the bodies will be the same). It should be noted that generics in Ada require hand-instantiation and, therefore, are not that helpful, since every algorithm needs to be instantiated by hand. Templates in C++ solve this problem by allowing one to define things once.

There are still things that are needed for generic programming that are not yet representable in C++. Generic algorithms are algorithms that work on objects with similar interfaces. Not identical interfaces as in object-oriented programming, but similar. It is not just the handling of binary methods (see §2.1.3) that causes the problem, it is the fact that interfaces are described in terms of a single type (single-sorted algebra). If we look carefully at things like iterators we observe that they are describable only in terms of multiple types: the iterator type itself, the value type, and the distance type. In other words, we need three types to define the interfaces on one type. And there is no machinery in C++ to do that. The result of this is that we cannot define what iterators are and, therefore, cannot really compile generic algorithms. For example, if we define the reduce algorithm as:

```
template <class InputIterator, class BinaryOperationWithIdentity>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first, InputIterator last, BinaryOperationWithIdentity op)
{
  typedef typename iterator_traits<InputIterator>::value_type T;
  if (first == last) return identity_element(op);
  T result = *first;
  while (++first != last) result = op(result, *first);
  return result;
}
```

but instead of: `++first != last` we write: `++first < last`, no compiler can detect the bug at the point of definition. Though the standard clearly states that `operator<` does not need to be defined for Input Iterators, there is no way for the compiler to know it. Iterator requirements are just words. We are trying to program with concepts (multi-sorted algebras) in a language that has no support for them.

How hard would it be to extend C++ to really enable this style of programming? First, we need to introduce concepts as a new interface facility. For example, we can define:

```
concept SemiRegular : Assignable, DefaultConstructible {};
```

```
concept Regular : SemiRegular, EqualityComparable {};
concept InputIterator : Regular, Incrementable {
  SemiRegular value_type;
  Integral distance_type;
  const value_type& operator*();
};

value_type(InputIterator)
reduce(InputIterator first, InputIterator last, BinaryOperationWithIdentity op)
(value_type(InputIterator) == argument_type(BinaryOperationWithIdentity))
{
  if (first == last) return identity_element(op);
  value_type(InputIterator) result = *first;
  while (++first != last) result = op(result, *first);
  return result;
}
```

Generic functions are functions that take concepts as arguments and in addition to an argument list have a list of type constraints. Now full type checking can be done at the point of definition without looking at the points of call, and full type-checking can be done at the points of call without looking at the body of the algorithm.

Sometimes we need multiple instances of the same concept. For example,

```
OutputIterator merge(InputIterator[1] first1, InputIterator[1] last1,
                     InputIterator[2] first2, InputIterator[2] last2,
                     OutputIterator result)
(bool operator<(value_type(InputIterator[1]), value_type(InputIterator[2])),
 value_type(InputIterator[1]) == value_type(InputIterator[2]),
 output_type(OutputIterator) == value_type(InputIterator[2]));
```

Note that this merge is not as powerful as the STL merge. It cannot merge a list of `floats` and a vector of `doubles` into a deque of `ints`. STL algorithms will often do unexpected and, in my opinion, undesirable type conversions. If someone needs to merge `doubles` and `floats` into `ints`, he or she should use an explicit function object for asymmetric comparison and a special output iterator for conversion.

C++ provides two different abstraction mechanisms: object-orientedness and templates. Object-orientedness allows for exact interface definition and for run-time dispatch. But it cannot handle binary methods or multi-method dispatching, and its run-time binding is often inefficient. Templates handle richer interfaces and are resolved at compile-time. They can, however, cause a software engineering nightmare because of the lack of separation between interfaces and implementation. For example, I recently tried compiling a 10-line STL-based program using one of the most popular C++ compilers, and ran away in shock after getting several pages of incomprehensible error messages. And often one needs run-time dispatch that cannot be handled by templates. I do believe that introduction of concepts will unify both approaches and resolve both sets of limitations. And after all, it is possible to represent concepts as virtual tables that are extended by pointers to type descriptors: the virtual table for input iterator contains not just pointers to `operator*` and `operator++`, but also pointers to the actual type of the iterator, its value type, and its distance type. And then one could introduce pointers to concepts and references to concepts!

Generic programming is a relatively young subdiscipline of computer science. I am happy to see that the small effort—started twenty years ago by Dave Musser, Deepak Kapur, Aaron Kershenbaum and me—led to a new generation of libraries such as BGL and MTL. And I have to congratulate Indiana

University on acquiring one of the best generic programming teams in the world. I am sure they will do other amazing things!

Alexander Stepanov
Palo Alto, California
September, 2001*

---

* I would like to thank John Wilkinson, Mark Manasse, Marc Najork, and Jeremy Siek for many valuable suggestions.