Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.

Contents

1 Introduction

By generic programming, we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms. The purpose of this paper is to convey the key ideas of generic programming, focusing mainly on establishing a methodological framework that includes appropriate notions of algorithm and data abstractions, and what it means to formally specify and verify such abstractions.

We present the issues relating to generic algorithms mainly in terms of a single example, a `Partition` algorithm such as is used in quicksort, but we will also allude to a large collection of examples we have developed in Ada as part of an Ada Generic Library project [6], [7]. The structure of the library is designed to achieve a much higher degree of modularity than has been found in previous libraries, by completely separating data representations from algorithmic and other data abstraction issues. Some of our goals are in common with the "parameterized programming" approach advocated by J. Goguen [4], but the most fundamental difference is that Goguen mainly addresses meta-issues—namely, how to manipulate theories—while our primary interest is in building useful theories to manipulate.

The notion of generic algorithms is not new, but we are unaware of any similar attempt to structure a software library founded on this idea. The Ada library developed by G. Booch [1], for example, makes only very limited use of generic algorithms. Booch has developed an interesting taxonomy of data structures and has populated it with many different abstract data types, but the implementations of these data types are for the most part built directly on Ada's own data structure facilities rather than using other data abstractions in the library; i.e., there is very little layering of the implementations. (Some use of generic algorithms and layering is described for list and tree structure algorithms, but almost as an afterthought in a chapter on utilities).

In fact, most work on development of abstraction facilities for the past decade or more has focused on data abstraction [2], [9]. Algorithmic abstraction has received little attention, even in the more recent work on object oriented programming. Most work on procedural abstraction has been language- rather than algorithm-oriented, attempting to find elegant and concise linguistic primitives; our goal is to find abstract representations of efficient algorithms.

As an example of algorithmic abstraction, consider the task of choosing and implementing a sorting algorithm for linked list data structures. The merge sort algorithm can be used and, if properly implemented, provides one of the most efficient sorting algorithms for linked lists. Ordinarily one might program this algorithm directly in terms of whatever pointer and record field access operations are provided in the programming language. Instead, however, one can abstract away a concrete representation and express the algorithm in terms of the smallest possible number of generic operations. In this case, we essentially need just four operations: `Next` and `Set_Next` for accessing the next cell in a list, `Is_End` for detecting the end of a list, and `Test`, a binary predicate on (the data in) cells. For a particular representation of linked lists, one then obtains the corresponding version of a merge sorting algorithm by instantiating the generic access operations to be subprograms that access that representation.

We believe it is better whenever possible to give programming examples in a real language rather than using pseudo-language (as is so frequently done). Although we do not argue that Ada is perfect for expressing the programming abstractions we have found most useful, it has been adequate in most cases and it supports our goal of efficiency through its compile time expansion of generics and provision for directing that subprograms be compiled inline. For numerous examples of the use of generic programming techniques in the Scheme language, and a brief discussion of the relative merits of Ada and Scheme for this type of programming, see [5].

2 Classification of Abstractions

We discuss four classes of abstractions that we have found useful in generic programming, as shown in Table 1, which lists a few examples of packages in our Ada Generic library. Each of these Ada packages has been written to provide generic algorithms and generic data structures that fall into the corresponding abstraction class.

2.1 Data abstractions

Data abstractions are data types and sets of operations defined on them (the usual definition); they are abstractions mainly in that they can be understood (and formally specified by such techniques as algebraic axioms) independently of their actual implementation. In Ada, data abstractions can be written as packages which define a new type and procedures and functions on that type. Another degree of abstractness is achieved by using a generic package in which the type of elements being stored is a generic formal parameter. In our library, we program only a few such data abstractions directly—those necessary to create some fundamental data representations and define how they are implemented

Data Abstractions Data types with operations defined on them	System_Allocated_Singly_Linked User_Allocated_Singly_Linked {Instantiations of representational abstractions}
Algorithmic Abstractions Families of data abstractions with common algorithms	Sequence_Algorithms Linked_List_Algorithms Vector_Algorithms
Structural Abstractions Intersections of algorithmic abstractions	Singly_Linked_Lists Doubly_Linked_Lists Vectors
Representational Abstractions Mappings from one structural abstraction to another	Double_Ended_Lists Stacks Output_Restricted_Deques

Table 1: Classification of Abstractions and Example Ada Packages

in terms of Ada types such as arrays, records and access types. Most other data abstractions are obtained by combining existing data abstraction packages with packages from the structural or representational classes defined below.

2.2 Algorithmic abstractions

These are families of data abstractions that have a set of efficient algorithms in common; we refer to the algorithms themselves as *generic algorithms*. For example, in our Ada library there is a package of generic algorithms for linked-lists and a more general package of sequence algorithms whose members can be used on either linked-list or vector representations of sequences. The linked-list generic algorithms package contains 31 different algorithms such as, for example, generic merge and sort algorithms that are instantiated in various ways to produce merge and sort subprograms in structural abstraction packages such as singly-linked lists and doubly-linked lists. We stress that the algorithms at this level are derived by abstraction from concrete, efficient algorithms.

2.3 Structural abstractions

Structural abstractions (with respect to a given set of algorithmic abstractions) are also families of data abstractions: a data abstraction A belongs to a structural abstraction

S if and only if S is an intersection of some of the algorithmic abstractions to which A belongs. An example is singly-linked-lists, the intersection of sequence- , linked-list-, and singly-linked-list-algorithmic abstractions. It is a family of all data abstractions that implement a singly-linked representation of sequences (it is this connection with more detailed structure of representations that inspires the name "structural abstraction").

Note that, as an intersection of algorithmic abstractions, such a family of data abstractions is smaller than the algorithm abstraction classes in which it is contained, but a *larger* number of algorithms are possible, because the structure on which they operate is more completely defined.

Programming of structural abstractions can be accomplished in Ada with the same kind of generic package structure as for generic algorithms. The `Singly_Linked_Lists` package contains 66 subprograms, most of which are obtained by instantiating or calling in various ways some member of the `Sequence_Algorithms` package or one of the linked-list algorithms packages. In Ada, to place one data abstraction in the singly-linked-lists family, one instantiates the `Singly_Linked_Lists` package, using as actual parameters a type and the set of operations on this type from a data abstraction package such as `System_Allocated_Singly_Linked` that defines an appropriate representation.

2.4 Representational abstractions

These are mappings from one structural abstraction to another, creating a new type and implementing a set of operations on that type by means of the operations of the domain structural abstraction. For example, stacks can easily be obtained as a structural abstraction from singly-linked-lists. Note that what one obtains is really a family of stack data abstractions, whereas the usual programming techniques give only a single data abstraction.

The following sections give more detailed examples of algorithmic abstractions. Further discussion and examples of data, structural, and representational abstraction may be found in [6].

3 Algorithmic Abstractions

As an example of generic algorithms, we consider the *sequence* algorithmic abstraction: diverse data abstractions which can be sequentially traversed. These data abstractions belong to numerous different families: singly-linked lists, doubly-linked lists, vectors, trees, and many others. There are many algorithms that make sense on all of them and require only a few simple access operations for their implementation: find an element, accumulate values together (by $+$ or $\times$, for example), count elements satisfying some predicate, etc.

The solution that has been taken in Common Lisp [8] is to index all kinds of sequences by natural numbers. So the Common Lisp generic `find` function always returns a natural number, which is not particularly useful on linked lists.

In the generic programming approach, we use generic indexing by a generic formal type, `Coordinate`. `Coordinate` is instantiated to type `Natural` for vectors; for linked lists, however, cells themselves can serve as `Coordinate` values. A generic `Find` can thus return a `Coordinate` value that can be used to reference the located element directly.

The intended semantics for `Coordinate` is that there are functions

- `Initial` from `Sequence` to `Coordinate`,
- `Next` from `Coordinate` to `Coordinate`,
- `Is_End` from `Sequence` $\times$ `Coordinate` to `Boolean`, and
- `Ref` from `Sequence` $\times$ `Coordinate` to a third type `Element`,

such that for any sequence S there are a natural number N (called the *length* of S) and coordinates $I_0, I_1, \ldots, I_N$ such that

- $I_0 = \texttt{Initial}(S)$ and $I_i = \texttt{Next}(I_{i-1})$ for $i = 1, \ldots N$;
- the elements of S are given by $\texttt{Ref}(S, I_0), \texttt{Ref}(S, I_1), \ldots, \texttt{Ref}(S, I_{N-1})$;
- $\texttt{Is\_End}(S, I_i)$ is false for $i = 0, 1, \ldots, N-1$, and true for $i = N$.

We further assume that each of the functions `Initial`, `Next`, `Ref`, and `Is_End` is a constant time operation. It is important that `Ref` provides constant time access, so that after `Find` returns the coordinate it is possible to access the data without any additional traversal of the sequence. Thus, for example, one could not use natural numbers as coordinates when the sequences are linked lists.

In Ada we can write:

```
generic
  type Sequence   is private;
  type Coordinate is private;
  type Element is private;
  with function Initial(S : Sequence) return Coordinate;
```

```
    with function Next(C : Coordinate) return Coordinate;
    with function Is_End(S : Sequence; C : Coordinate) return Boolean;
    with function Ref(S : Sequence; C : Coordinate) return Element;
  package Sequence_Algorithms is
    -- definitions of sequence algorithms such as
    -- Count, Find, Every, Notany, Some, Search; e.g.,
    generic
      with function Test(S : Sequence; C : Coordinate) return Boolean;
    procedure Find(S        : Sequence;
                   Result   : out Coordinate;
                   Is_Found : out Boolean);
end Sequence_Algorithms;
```

We have made Find a procedure instead of a function so that the case in which an element satisfying Test is not found does not require some "extra" coordinate value to be returned; such an extra value might not exist for some instances of the coordinate type. Note also that Find is a generic procedure; in addition to forming an instance of the Sequence_Algorithms package, the programmer would also create particular instances of Find in which some particular test, such as equality to a particular value, would be substituted for Test.

The body of Find could be expressed as follows:

```
  package body Sequence_Algorithms is
  -- among other things
    procedure Find(S        : Sequence;
                   Result   : out Coordinate;
                   Is_Found : out Boolean) is
      Current : Coordinate;
      Flag    : Boolean;
    begin
      Current := Initial(S);
      while not Is_End(S, Current) loop
        if Test(S, Current) then
          Result := Current; Is_Found := True; return;
        end if;
        Current := Next(Current);
      end loop;
      Result := Current; Is_Found := False;
```

```
  end Find;
end Sequence_Algorithms;
```

It should be noted that not every possible data abstraction which contains a set of elements can be a member of the sequence algorithmic abstraction. Some data abstractions do not contain an explicit coordinate type (stacks, or queues, for example). Intuitively speaking, the intended data abstractions are those which can be iterated through without side-effects, and where there is a coordinate type which can be used to represent the current position in a manner allowing constant time access.

4 A Generic Partition Algorithm

Another limitation of sequence algorithms as we have defined them in the previous section is that they allow only for one-directional traversals of sequences. There are several algorithms which require a bidirectional traversal of sequences by two variables of the type `Coordinate` advancing towards each other. If we assume there is a `Prev` operation such that `Prev`(`Next`(I)) = `Next`(`Prev`(I)) = I, and a `Swap` operation for exchanging elements, then we can obtain generic implementations of such procedures as `Reverse` and `Partition` (as in quicksort). We examine the `Partition` algorithm in particular as a more detailed example of the issues that arise with programming and reasoning about algorithmic abstractions.

In Ada, we could provide such algorithms in a generic package,

```
generic
  type Sequence   is private;
  type Coordinate is private;
  with function Next(I : Coordinate) return Coordinate;
  with function Prev(I : Coordinate) return Coordinate;
  with procedure Swap(S : in out Sequence; I, J : Coordinate);
package Bidirectional_Sequence_Algorithms is
  -- for example:

  procedure Reverse(S : in out Sequence);

  generic
    with function Test(S : Sequence; C : Coordinate) return Boolean;
  procedure Partition(S         : in out Sequence;
```

```
                    F, L      : in Coordinate;
                    Middle    : out Coordinate;
                    Middle_OK : out Boolean);

end Bidirectional_Sequence_Algorithms;
```

which could be used along with the `Sequence_Algorithms` package to construct collections of algorithms for different kinds of linear lists. In this package we do not need to have `Element` and `Ref` as generic parameters, since algorithms such as `Reverse` and `Partition` do not directly refer to them.

To describe the `Partition` algorithm informally (a formal treatment follows in Section 5), we speak of `Next`(I) giving a "larger" coordinate than I and `Prev` giving one that is smaller; and we also speak of elements that satisfy `Test` as "good" and those that don't as "bad." Provided `Swap`(S, I, J) exchanges the elements with coordinates I and J and leaves all other elements of S unchanged, the `Partition` algorithm rearranges the elements of S between those with coordinates F and L so that all of the good elements come first, followed by all the bad elements. `Middle` $= M$ is computed as a coordinate between F and L (inclusive) such that all of the elements with smaller coordinates are good and all elements with greater coordinates are bad; the M-th element is good if and only if `Middle_OK` is true.

`Middle_OK` is needed because for N elements there are $N + 1$ possibilities for the boundary between good and bad elements, but we are guaranteed of having only N coordinate values; in general we cannot assume the existence of coordinate values outside the range from F to L. This complication does not arise with the usual concrete partition algorithm in which coordinates are integers, since one could use values $F - 1$ or $L + 1$ for M. This is an example of the extra care that must be taken in expressing an algorithm at a more abstract level.

The `Partition` algorithm can be expressed in Ada as shown in Figure 1. A somewhat shorter implementation could be achieved in which calls to `Test` with the same arguments might be repeated, but since `Test` is a generic parameter we must be careful to avoid such redundant calls, since one might instantiate the algorithm with a `Test` that is fairly expensive to compute.

5 Abstract Algorithm Specification and Verification

The main idea of our approach to specification and verification of a generic algorithm is similar to classical program verification techniques, e.g., Dijkstra's idea of weakest pre-

```
procedure Partition(S          : in out Sequence;
                    F, L       : in Coordinate;
                    Middle     : out Coordinate;
                    Middle_OK : out Boolean) is
  First : Coordinate := F;
  Last  : Coordinate := L;
begin
  loop
    loop
      if First = Last then
        Middle := First;
        Middle_OK := Test(S, First);
        return;
      end if;
      exit when not Test(S, First);
      First := Next(First);
    end loop;
    loop
      exit when Test(S, Last);
      Last := Prev(Last);
      if First = Last then
        Middle := First;
        Middle_OK := False;
        return;
      end if;
    end loop;
    Swap(S, First, Last);
    First := Next(First);
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Last := Prev(Last);
  end loop;
end Partition;
```

Figure 1: Body of Partition Algorithm

conditions [3], in which one attempts to obtain a strong statement about the result of a computation while making as few assumptions as possible about its initial conditions. In discussing preconditions of ordinary, nongeneric algorithms, the assumptions one makes about the operations in terms of which the algorithm is expressed are fixed, since the operations themselves are fixed. But for generic algorithms we want to make these operations generic parameters and vary the assumptions about them; our goal is both to consider a variety of possible postconditions and to maximize the number of different models (abstract data types) under which an an algorithm attains a given postcondition. In terms of proof theory, we want to consider how to prove various postconditions under a variety of assumptions about the generic parameters, so that later we can easily prove correctness of a wide variety of instances.

It appears that the best approach is to build up the specifications and the verification lemmas in stages, just as we build up algorithmic capabilities in layers. In fact we will find it advantageous to have even more layering in the specifications and proofs than in the construction of the algorithms.

We introduce the main ideas in terms of the `Partition` algorithm given in the previous section. What are the minimal assumptions we need to make about the generic parameters in terms of which `Partition` is programmed, namely the `Sequence` and `Coordinate` types and the `Next`, `Prev`, and `Swap` operations? If we want to use the algorithm only for partitioning in the usual sense, and the only use of abstraction is in the use of the `Coordinate` type rather than a more specific integer type (for an array version) or pointer type (for a linked list version), then we could carry out the specification and proof in one step in which we make strong assumptions about the generic parameters.

Instead, however, we begin with weaker assumptions about these generic parameters, and obtain a lemma about the results of `Partition` that enables us to deal with less conventional instances of partitioning. For example, suppose that we are only interested in the part of the output consisting of elements that satisfy `Test`; i.e., we do not need to process the elements that don't satisfy it. Then we can obtain a more efficient partitioning algorithm by using for `Swap`(S, I, J) an operation that just performs the assignment of the element with the J-th coordinate to the I-th coordinate. If we had made a stronger assumption about `Swap`, that it exchanges two elements in the sequence, then the theorem about the generic algorithm would not be applicable to this instance.

Instead, we carry out the specification and proofs in layers, one of which allows us to "tap in" at the level we need to verify the second kind of `Partition`, while the first, more usual, kind can be verified when additional assumptions are made about `Swap` and combined with the lemma stated at the first level.

First, we assume that associated with the `Coordinate` type there is a predicate $<$

which is a well-founded partial ordering[1] on `Coordinate`, defined by

$$I < J \equiv \exists N(N > 0 \text{ and } J = \texttt{Next}^N(I)).$$

For coordinates F and L, where $F \leq L$, we define

$$\texttt{Coordinate\_Range}(F, L) = \{\texttt{Next}^i(F) : 0 \leq i \leq N\},$$

where N is the smallest integer such that $\texttt{Next}^N(F) = L$. Note that

$$\texttt{Prev}^i(L) = \texttt{Next}^{N-i}(F).$$

5.1 Basic specification of Partition

Initially we make only a weak assumption about `Swap`:

Swap Assumption 1 If $I, J \in \texttt{Coordinate\_Range}(F, L)$, then $\texttt{Swap}(S, I, J)$ computes S_1 such that for all $K \in \texttt{Coordinate\_Range}(F, L) - \{I, J\}$,

$$\texttt{Test}(S_1, K) = \texttt{Test}(S, K).$$

The reason that we can get by with such a weak assumption is simply that we express the result `Partition` computes as equivalent to that produced by a straight-line sequence of calls of `Swap`. The specification asserts the existence of two sequences of coordinate values that serve as arguments to the `Swap` calls and constrains the relationship between these values.

Syntactic Specification

```
procedure Partition(S         : in out Sequence;
                    F, L      : in Coordinate;
                    Middle    : out Coordinate;
                    Middle_OK : out Boolean);
```

Formal Semantics

`Next` and `Prev` are assumed to obey the relations discussed above and to have no side effects; `Swap` is assumed to obey Swap Assumption 1. With inputs $\texttt{S} = S, \texttt{F} = F, \texttt{L} = L$, with $F \leq L$ according to the partial ordering relation $<$ defined above, `Partition` outputs $\texttt{S} = S_1, \texttt{Middle} = M, \texttt{Middle\_OK} = B$ such that:

[1] Note that $<$ is not generic parameter of the package because it is not used in expressing the algorithms themselves, as it would be expensive to implement for, say, doubly-linked lists.

1. $M \in \texttt{Coordinate\_Range}(F, L)$,
2. There are two sets of coordinates

$$\begin{aligned} \texttt{Accept} &= \begin{cases} \texttt{Coordinate\_Range}(F, M) & \text{if } B \text{ is true} \\ \texttt{Coordinate\_Range}(F, M) - \{M\} & \text{if } B \text{ is false} \end{cases} \\ \texttt{Reject} &= \texttt{Coordinate\_Range}(F, L) - \texttt{Accept} \end{aligned}$$

   and a nonnegative integer n and two `Coordinate` sequences $I_1, \ldots, I_n$ and $J_1, \ldots, J_n$ such that

   (a) $F \leq I_1 < \ldots < I_n \leq M \leq J_n < \ldots < J_1 \leq L$

   (b) For $k = 1, \ldots, n$, $\texttt{Test}(S, I_k)$ is false while $\texttt{Test}(S, J_k)$ is true.

   (c) For $P \in \texttt{Accept} - \{I_1, \ldots, I_n\}$, $\texttt{Test}(S, P)$ is true.

   (d) For $P \in \texttt{Reject} - \{J_1, \ldots, J_n\}$, $\texttt{Test}(S, P)$ is false.

   (e) S_1 is the final value of `S` computed by

   $$\texttt{Swap(S, } I_1\texttt{, } J_1\texttt{); ... ; Swap(S, } I_n\texttt{, } J_n\texttt{);}$$

The fact that the coordinate type is abstract compels taking considerable care in this specification to avoid the mention of coordinate values that might not exist; e.g., we write $\texttt{Coordinate\_Range}(F, M) - \{M\}$ instead of $\texttt{Coordinate\_Range}(F, \texttt{Prev}(M))$.

Later we show how the above input-output specification can be proved as a lemma, by annotating the algorithm with assertions and using the inductive assertions method. For now, we consider how to combine this specification with additional assumptions about `Swap` in order to draw stronger conclusions about the output of `Partition`. We add these assumptions one at a time, thereby obtaining several useful lemmas that apply to different instances of `Partition`.

5.2 Obtaining a sequence satisfying Test

By making a second assumption about `Swap`, we can draw a stronger conclusion about `Partition`.

Swap Assumption 2 If $I, J \in \texttt{Coordinate\_Range}(F, L)$, then $\texttt{Swap}(S, I, J)$ computes S_1 such that

$$\texttt{Test}(S_1, I) = \texttt{Test}(S, J).$$

This can be combined with the basic `Partition` specification to deduce:

Partition Lemma 1 *If* `Swap` *satisfies Swap Assumptions 1 and 2, then the sequence* S_1 *computed by* `Partition` *satisfies*

Partition Property 1 *For all* $K \in$ `Accept`*,* `Test`(S_1, K) *is true.*

Similarly, we can introduce

Swap Assumption 3 If $I, J \in$ `Coordinate_Range`(F, L), then `Swap`(S, I, J) computes S_1 such that

$$\texttt{Test}(S_1, J) = \texttt{Test}(S, I).$$

and this can be combined with the basic `Partition` specification to deduce:

Partition Lemma 2 *If* `Swap` *satisfies Swap Assumptions 1 and 3, then the sequence* S_1 *computed by* `Partition` *satisfies*

Partition Property 2 *For all* $K \in$ `Reject`*,* `Test`(S_1, K) *is false.*

Partition Lemma 3 *If* `Swap` *satisfies Swap Assumptions 1, 2 and 3, then the sequence* S_1 *computed by* `Partition` *satisfies Partition Properties 1 and 2.*

Note that we have not yet made any assumption about `Swap` actually exchanging two elements of a sequence; we have only assumed that it does not affect elements other than those with coordinates I and J insofar as can be determined by `Test`, and that it changes the I-th element to have the same `Test`-value as the J-th, or vice-versa. Thus, for example, if we have sequences of integers and `Test` just checks whether an element is positive, a `Swap` operation that assigns 1 or -1 to the I-th element according to whether the J-th is nonpositive or positive would satisfy Swap Assumption 1 and 2!

Thus to be able to conclude that the sequence S_1 computed by `Partition` is a permutation of its input S, we need to assume that `Swap` satisfies

Swap Assumption 4 If $I, J \in$ `Coordinate_Range`(F, L), then `Swap`(S, I, J) computes S_1 such that for all $K \in$ `Coordinate_Range`$(F, L) - \{I, J\}$,

$$\texttt{Ref}(S_1, K) = \texttt{Ref}(S, K).$$

Swap Assumption 5 If $I, J \in$ `Coordinate_Range`(F, L), then `Swap`(S, I, J) computes S_1 such that `Ref`$(S_1, I) =$ `Ref`(S, J).

Swap Assumption 6 If $I, J \in$ `Coordinate_Range`(F, L), then `Swap`(S, I, J) computes S_1 such that `Ref`$(S_1, J) =$ `Ref`(S, I).

From these assumptions we obtain

Partition Lemma 4 *If* `Swap` *satisfies Swap Assumptions 4, 5, and 6, then it also satisfies Swap Assumptions 1, 2 and 3, and* `Partition` *has Partition Properties 1 and 2, as well as*

Partition Property 3 *The output sequence S_1 is a permutation of the input sequence S.*

5.3 Correctness when Swap is an assignment operation

Now suppose, instead of performing an exchange of two elements, `Swap` just does an assignment operation, so that it satisfies Swap Assumption 5 but not Swap Assumption 6. We can still conclude:

Partition Lemma 5 *If* `Swap` *satisfies Swap Assumptions 5 and 6 , then it also satisfies Swap Assumptions 1 and 2, and* `Partition` *has Partition Property 1, as well as*

Partition Property 4 *The sequence of elements of the output sequence S_1 with coordinates in* `Accept` *is the subsequence of the elements of S with coordinates in* `Coordinate_Range`(F, L) *and satisfying* `Test`*.*

This tells us that by instantiating `Partition` with `Swap` as an assignment operation, we obtain a version that brings together all the elements that satisfy `Test`. It does not yield all the elements that do not satisfy `Test`, but in some applications we would not need this information.

The main benefit of dividing the specification of assumptions about `Swap` and conclusions about `Partition` into small pieces is that we can deal with the question of correctness of different instances merely by citing the appropriate lemmas (or we can create and prove new lemmas with comparatively little effort). The work of proof has been factored into small steps that allow us the same benefits of reuse in proofs as generic algorithms allow us in programming.

5.4 Correctness of the body of Partition

To prove partial correctness, we add three internal assertions to the body of `Partition` (Figure 1). In these assertions note that S refers to the *initial* value of variable `S`, while S_1 refers to the current value at the point of the assertion:

1. (At the beginning of the first inner loop.) There are a nonnegative integer n and two `Coordinate` sequences $I_1, \ldots, I_n$ and $J_1, \ldots, J_n$ such that

   (a) $F \leq I_1 < \ldots < I_n < \texttt{First} \leq \texttt{Last} < J_n < \ldots < J_1 \leq L$;

   (b) For $k = 1, \ldots, n$, $\texttt{Test}(S, I_k)$ is false while $\texttt{Test}(S, J_k)$ is true;

   (c) for $P \in \texttt{Coordinate\_Range}(F, \texttt{First}) - \{I_1, \ldots, I_n, \texttt{First}\}$, $\texttt{Test}(S, P)$ is true;

   (d) for $P \in \texttt{Coordinate\_Range}(\texttt{Last}, L) - \{\texttt{Last}, J_1, \ldots, J_n\}$, $\texttt{Test}(S, P)$ is false;

   (e) The current value S_1 of `S` is the value of `S` computed by

   $$\texttt{Swap(S, } I_1\texttt{, } J_1\texttt{); ...; Swap(S, } I_n\texttt{, } J_n\texttt{);}$$

2. (At the beginning of the second inner loop.) Assertion 1, $\texttt{First} \neq \texttt{Last}$, and $\texttt{Test}(S, \texttt{First})$ is false.

3. (Just before the Swap call.) Assertion 2 and `Test(S, Last)` is true.

These assertions along with entry and exit assertions obtained from the Formal Semantics of the algorithm are sufficient for carrying out an inductive assertions proof of partial correctness: any path from the beginning of the procedure to the exit is composed of a finite number of path segments between two assertions, and one can verify that for each such path segment the assertion at the beginning combined with the semantics of the statements along the path implies the assertion at the end. We omit these proofs.

To prove total correctness, we need to show also that the procedure always terminates. Actually, the result we prove is conditional on the termination of `Test` and `Swap`. Note that

1. it can be shown inductively that, at all times, $\texttt{First} \leq \texttt{Last}$;

2. every path segment (as defined in the partial correctness proof) that can be repeated in any execution of the procedure contains an assignment that increases `First` or decreases `Last` (on one path both are changed, but not without violating 1);

Therefore, any execution consists of only a finite number of path segments. Since each path segment contains only assignment statements, equality tests, or calls to `Test` and `Swap`, we have

Partition Lemma 6 *If the generic parameters* `Test` *and* `Swap` *always terminate, then* `Partition` *always terminates.*

6 Conclusion

In this paper we have attempted to develop a framework sufficient to encompass the key aspects of generic programming, with illustrations from our experience in building a library of generic software components in Ada. Although the documentation of the initial library in [7] is informal, and we have not yet carried out formal specification and verification of the library components, we believe that this task would be both mathematically very interesting and practically very useful.

On the mathematical side, the correctness of generic algorithms offers greater challenges and less tedium than concrete algorithms, for often one must create the appropriate abstract concepts in terms of which one can effectively express and reason about the behavior of an algorithm or collection of algorithms. The nature of the problem of verifying generic algorithms should be attractive to researchers in computer science and mathematics, whereas the problem for concrete algorithms is often regarded as so tedious as to be worth doing only if most of the work can be done with an automated reasoning system.

On the practical side, the considerable work of composing a formal specification and carrying out a detailed proof of correctness at a generic level is compensated by the ease with which one is then able to deal with the correctness of many distinct instantiations. While it is often difficult to justify the amount of effort required for formal verification of concrete programs, except in the case of software used in life-critical systems, the possibility of verifying components in generic software libraries may open the way for the benefits of this technology to become much more widely available.

References

[1] G. Booch, *Software Components in Ada.* Benjamin/Cummings, 1987.

[2] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[4] J. Goguen, "Parameterized Programming," *Transactions on Software Engineering*, SE-10(5):528-543, September 1984.

[5] A. Kershenbaum, D. R. Musser and A. A. Stepanov, "Higher Order Imperative Programming," Computer Science Dept. Rep. No. 88-10, Rensselaer Polytechnic Institute, Troy, New York, April 1988.

[6] D. R. Musser and A. A. Stepanov, "A Library of Generic Algorithms in Ada," *Proc. of 1987 ACM SIGAda International Conference*, Boston, December, 1987.

[7] D. R. Musser and A. A. Stepanov, *The Ada Generic Library: Linear List Processing Packages*, Springer–Verlag, 1989. (This book supercedes General Electric Corporate Research and Development Reports 88CRD112 and 88CRD113, April 1988).

[8] G. L. Steele, *Common LISP: The Language*, Digital Press, 1984.

[9] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.