A Library of Generic Algorithms in Ada

David R. Musser
General Electric Company
Corporate Research & Development
P. O. Box 8
Schenectady, New York 12301

Alexander A. Stepanov
Polytechnic University
Computer Science Department
333 Jay Street
Brooklyn, New York 11201

Abstract

It is well-known that data abstractions are crucial to good software engineering practice. We argue that algorithmic abstractions, or generic algorithms, are perhaps even more important for software reusability. Generic algorithms are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms. We discuss this notion with illustrations from the structure of an Ada library of reusable software components we are presently developing.

1 Introduction

1.1 Purpose of the library

The purpose of the Ada Generic Library is to provide an Ada programmer with powerful generic packages for data structures such as lists, matrices, strings, trees, and graphs, along with numerical and combinatorial algorithms. Our main goal in this introduction is to explain both the structure of this particular library and the general principles we have followed in creating that structure. We believe these principles, which are quite different from those on which other libraries such as in [1] have been founded, have broad applicability to the goal of widely-usable software components in Ada.

The first phase of the library concentrates on a significant subset of the data structures problem: an extensive set of *linear data structure* manipulation facilities for different kinds of linked lists and vectors (one dimensional arrays). The data structures and algorithms included have been selected based on their well-established usefulness in a wide variety of of applications. Over 300 subprograms will be provided in the first phase of the library, in eleven Ada packages. (In the current release, eight packages containing over 150 subprograms are included.) This development is a part of the Reusable Ada Repository System, a joint project of GE's Western Systems (Sunnyvale, California) and Corporate Research and Development.

1.2 Principles behind the library

The main principles we have followed in building the library are the following:

1. Extensive use of generic algorithms, such as generic *sort* and *merge* algorithms that can be specialized to work for many different data representations and comparison functions.
2. Building up functionality in layers, separating, to as large an extent as possible, concerns about representations from those of algorithms.
3. Obtaining high efficiency in spite of the layering (using Ada's *inline* compiler directive).
4. Emphasis on careful selection and programming of highly efficient algorithms.
5. High quality documentation that makes it easy to find operations in the library and select the best algorithm and data structure for the application at hand.

The most important technical idea is that of generic algorithms, which are a means of providing functionality in a way that abstracts away from details of representation and basic operations. Instead of referring directly to the host language facilities, generic algorithms are defined in terms a few primitive operations that are considered to be *parameters*. By plugging in actual

operations for these parameters, one obtains specific instances of the algorithms for a specific data structure. By carefully choosing the parameterization and the algorithms, one obtains in a small amount of code the capability to produce many different useful operations. It becomes much easier to obtain the operations needed for a particular application by plugging components together than it would be to program them directly.

1.3 Related technology

The notion of generic algorithms is not entirely new, but there has not been any attempt to structure a general software library founded on this idea. Older program libraries, written in Fortran or other languages without the facilities for generic programming, could not take advantage of the algorithm abstractions that were known. But even the recent improvements in abstraction facilities in contemporary programming languages, such as Ada, have not precipitated widespread use of algorithmic abstraction. (Booch, for example, makes some use of generic algorithms for list and tree structures, but almost as an afterthought in a chapter on utilities.) For the benefits of this approach to be fully realized, great care must be exercised in selecting and structuring algorithms, especially in determining how they are parameterized and how they are used to develop more concrete levels of the library. Indeed, we view algorithm selection, abstraction, and structuring as being of far greater importance to software reusability than any language or other human-interface issues; experience with Unix tools provides ample evidence of this point.

2 Structure of the Library

The key structuring mechanism used in building the library is *abstraction*. We discuss four classes of abstractions that we have found useful in structuring the library, as shown in Table 1, which lists a few examples of packages in the library. Each of these Ada packages has been written to provide generic algorithms and generic data structures that fall into the corresponding abstraction class. (The packages marked with a * are not included in the current release of the library.) Brief definitions of the abstraction classes are given in the table and are illustrated in Figure 1.

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 in terms of Ada types such as arrays, records and access types. Three such packages, which we refer to as "low-level data abstraction packages," are included in the current library. 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 library there is a package of generic algorithms for linked-lists; in a future release there will be 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. 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 three operations: `Next` and `Set_Next` for accessing the next cell in a list, and `Is_End` for detecting the end of a list. 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.

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

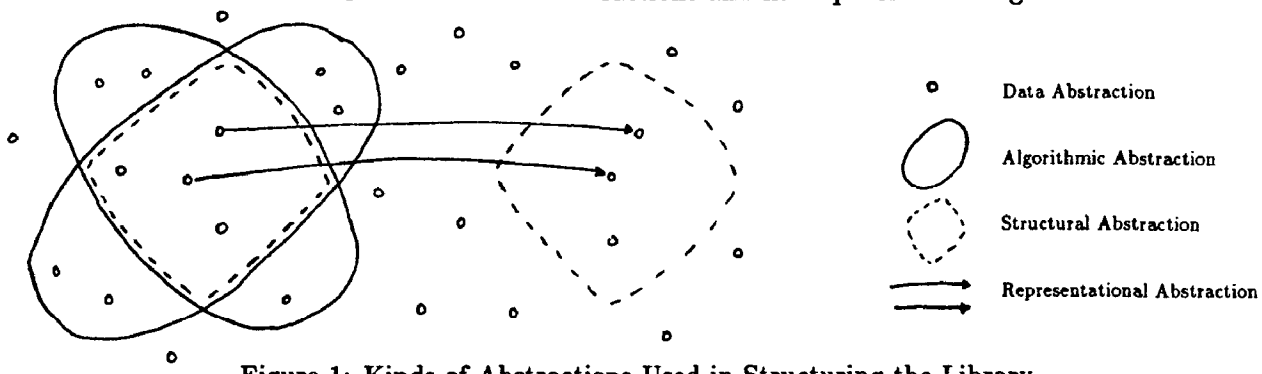

Figure 1: Kinds of Abstractions Used in Structuring the Library

Thus in Ada one programs generic algorithms in a generic package whose parameters are a small number of types and access operations—e. g.,

```
generic
  type Cell is private;
  with function Next(S : Cell) return Cell;
  with procedure Set_Next(S1, S2 : Cell);
  with function Is_End(S : Cell)
    return Boolean;
  with function Copy_Cell(S1, S2 : Cell)
    return Cell;
package Linked_List_Algorithms is
. . .
```

The subprograms in the package are algorithms such as `Merge` and `Sort` that are efficient when `Next`, `Set_Next`, etc., are instantiated with constant time operations.

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 singlylinkedlistalgorithmic 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"). (In the current release, the `Singly_Linked_Lists` package is actually programmed just in terms of the `Linked_List_Algorithms` package.)

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 `Linked_List_Algorithms` package. In Ada, to actually 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 a sequence structural abstraction, and this is carried out in Ada using generic packages in a manner that will be demonstrated in the Appendix. Note that what one obtains is really a family of stack data abstractions, whereas the usual programming techniques give only a single data abstraction.

3 Linear Data Structures

3.1 Low-level data abstractions

In the current release of the library we have provided three different low-level data abstractions using singly-linked list representations:

- The `System_Allocated_Singly_Linked` package provides records containing datum and link fields, allocated using the standard heap allocation and deallocation procedures.
- `Once_User_Allocated_Singly_Linked` provides more efficient allocation and deallocation by allocating an array of records as a storage pool, but is less flexible than the system allocated package since this array and the system heap are managed separately.
- `Auto_Reallocating_Singly_Linked` also uses an array of records for efficiency but automatically allocates a larger array whenever necessary; its disadvantage is that the parameters controlling the reallocation may need to be tuned to achieve optimum reallocation behavior.

3.2 Algorithmic, structural and representational abstractions

The current release of the library provides the following algorithmic, structural and representational abstraction packages:

- `Singly_Linked_Lists` is a structural abstraction package that provides over 60 subprograms for operations on a singly-linked list representation, including numerous kinds of concatenation, deletion, substitution, searching and sorting operations (the selection is based mainly on Common Lisp facilities [7]).
- `Linked_List_Algorithms` is a generic algorithms package that is the source of most of the algorithms used in `Singly_Linked_Lists`; many of the same algorithms will be used in implementing the `Doubly_Linked_Lists` package.
- `Stacks` provides the familiar linear data structure in which insertions and deletions are restricted to one end.
- `Double_Ended_Lists` employs header cells with singly-linked lists to make some operations such as concatenation more efficient and to provide more security in various computations with lists.
- `Output_Restricted_Deques` provides a data structure that restricts insertions to both ends and deletions to one end, making use of `Double_Ended_Lists`.

The latter three packages are representational abstractions that produce different structural abstractions from different representations of sequences. In particular, any of the three different low-level representations of singly-linked-lists provided can easily be plugged together with any of these three representational abstractions, as well as with the Singly_Linked_Lists package, for a total of 12 different possible combinations. Each of these 12 combinations, called a *Partially Instantiated Package*, or *PIP* for short, is included in the library. To use one of them, one only has to instantiate the element type, and perhaps some configuration parameters, to specific values.

A later release will also include:

- `Sequences`
- `Doubly_Linked_Lists`
- `Simple_Vectors`
- `Extensible_Vectors`

packages, along with several low-level data abstraction packages that plug together with them.

4 Selection from the library

The first observation we would make is that proper classification of software components for maximum usability may well depend more on *internal structure* than on functional (input-output) behavior. In searching the library, the programmer needs to know not only

whether there is a subprogram that performs the right operation, but also what kind of data representation it uses (if it is not a completely generic algorithm), since in all but the simplest cases it will be used in a particular context that may strongly favor one representation over another.

Experienced programmers will sometimes want to use generic algorithms directly, instantiating the generic access operations to be subprograms accessing a particular data representation. Although generic, these algorithms are tailored to be used with data representations with particular complexity characteristics, such as linked-list- versus array-like representations, and the programmer must be aware of these issues.

This is not to say that intelligent use of the library necessarily requires the programmer to examine the bodies of the subprograms. If construction of the library is, as we have recommended, algorithmically-driven and draws upon the best books and articles on algorithms and data structures, then it should be possible to develop sufficiently precise and complete *selection criteria* based on the advice in those books and articles. Again, the preparation of these selection criteria and other documentation must be done very carefully and thoroughly to make later usage by programmers as simple as possible.

With the current Linear Data Structures library, there are, at a minimum, three kinds of selections to be made:

1. the choice of a low-level data abstraction package
2. the choice of a structural or representational abstraction package
3. the choice of operations within the structural or representational package

The fact that the structure of our library allows separate choices for 1 and 2 means that there are many more selections available than would be the case with more conventional organizations. However, it is not the case that these choices are entirely independent of each other or of the choices in 3. In fact, the programmer will often have to give careful consideration to the the combination of operations that he or she expects to use in an application, and make a package selection based on algorithmic issues of time and space efficiency of the subprograms as documented in the subprogram descriptions. Another issue that might dictate a choice would be the possible exceptions raised by the operations to be used.

5 Conclusions and future work

In summary, the main points we want to make are:

- that to achieve truly reusable software components, extensive use should be made of algorithmic abstraction;
- that, indeed, development of a software library should be algorithmically driven;
- and that careful development of algorithms, data structures, and selection criteria are essential to the success of the library.

The library we are developing in Ada is a significant attempt to implement this approach. We have, with Aaron Kershenbaum of Polytechnic University, also experimented extensively with the generic algorithms approach in Scheme, using higher order procedures; and we have implemented a number of useful generic subprograms in C.

We have only recently begun actual construction of the library discussed in this paper; thus it is difficult to predict the full scope of this work. A library of the nature we have discussed could be very large—eventually containing hundreds of *packages*, each containing from a few to perhaps a hundred subprograms. How, it may be asked, will it be possible to make effective use of such a large library?

We believe there is no easy answer to this question, but the organization by abstraction classes discussed herein should substantially reduce the size of the library and aid in making effective use of it. As programmers gain experience with use of even a few of the packages they can begin learning the general structure the library, which will greatly assist in intelligent selection from and proper use a wide range of library components. Among all of the technologies being explored today to improve software productivity, it may well be that a well-structured library of generic components offers the greatest benefits.

References

[1] Booch, G., *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] Brian W. Kernighan and P. J. Plauger, "Software Tools in Pascal," Addison-Wesley, 1981.

[4] Donald E. Knuth, *The Art of Computer Programming*, Vols. 1-3, Addison-Wesley, 1968, 1969, 1973.

[5] D. R. Musser and A. A. Stepanov, *Ada Generic Library Linear Data Structure Packages*, Vol. 1, to appear as a General Electric Corporate Research and Development Report, October 1987.

[6] R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.

[7] Guy L. Steele, *Common LISP: The Language*, Digital Press, 1984.

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

A Appendix: Examples of library structure

At the algorithmic abstraction level, our library includes a package called Linked_List_Algorithms containing numerous algorithms for basic operations on linked lists that are efficient provided that the representation satisfies certain requirements. It can be described as a collection of many useful algorithms expressible in terms of Next, Set_Next, Is_End, and Copy_Cell operations that are assumed to take constant time. Using Linked_List_Algorithms, other packages provide structural abstractions that assume more details about particular representations: Singly_Linked_Lists and Doubly_Linked_Lists. We will examine a small number of the subprograms at these levels and see how they are used to build more concrete packages.

Specification of Linked_List_Algorithms package

```
generic
  type Cell is private;
  with function Next(S : Cell) return Cell;
  with procedure Set_Next(S1, S2 : Cell);
  with function Is_End(S : Cell) return Boolean;
  with function Copy_Cell(S1, S2 : Cell)
    return Cell;
package Linked_List_Algorithms is

      function Reverse_Append(S1, S2 : Cell)
        return Cell;
      -- Returns a sequence consisting of the
      -- elements of S1, in reverse order,
      -- followed by those of S2 in order.

      generic
            with function Test(X : Cell)
                  return Boolean;
      function Count(S : Cell) return Integer;
      -- Returns a non-negative integer, the
      -- number of elements E of S such that
      -- Test(E) is true.

      generic
            with function Test(X, Y : Cell)
                  return Boolean;
            with procedure Free(X : CELL);
      function Delete_Duplicates(S : Cell)
        return Cell;
      -- Returns a sequence of the elements
      -- of S but with only one occurrence of
      -- each, using Test(X,Y) as the test for
      -- equality.
        . . .

end Linked_List_Algorithms;
```

We have shown only three of the 31 subprograms in this package.

Body of Linked_List_Algorithms package

In the package body an auxiliary function called Advance is introduced and used in many of the algorithms.

```
package body Linked_List_Algorithms is

   procedure Advance(S : in out Cell) is
   begin
     S := Next(S);
   end Advance;

   pragma Inline(Advance);

   function Reverse_Append(S1, S2 : Cell)
     return Cell is
     Result     : Cell := S2;
     To_Be_Done : Cell := S1;
   begin
     while not Is_End(To_Be_Done) loop
       Result := Copy_Cell(To_Be_Done, Result);
       Advance(To_Be_Done);
     end loop;
     return Result;
   end Reverse_Append;

   function Count(S : Cell) return Integer is
     Result     : Integer := 0;
     To_Be_Done : Cell    := S;
   begin
     while not Is_End(To_Be_Done) loop
       if Test(To_Be_Done) then
         Result := Result + 1;
       end if;
       Advance(To_Be_Done);
     end loop;
     return Result;
   end Count;

   function Delete_Duplicates(S : Cell)
     return Cell is
     Tail, To_Be_Done, I : Cell := S;
   begin
     if not Is_End(To_Be_Done) then
       Advance(To_Be_Done);
       while not Is_End(To_Be_Done) loop
         I := S;
         while I /= To_Be_Done and then
               not Test(I, To_Be_Done) loop
           Advance(I);
         end loop;
         if I = To_Be_Done then
           Tail := To_Be_Done;
           Advance(To_Be_Done);
         else
           I := To_Be_Done;
           Advance(To_Be_Done);
           Set_Next(Tail, To_Be_Done);
           Free(I);
         end if;
       end loop;
     end if;
     return S;
   end Delete_Duplicates;

   . . .

end Linked_List_Algorithms;
```

Specification of Singly_Linked_Lists

This structural abstraction is expressed in terms of primitive operations for list access that will be provided by a low-level data abstraction package (they are named with the character 0 appended so that it is possible to use renamings to export these operations.)

```
with Linked_Exceptions;
generic
  type Element0  is private;
  type Sequence0 is private;
  Nil0 : Sequence0;
  with function First0(S : Sequence0)
    return Element0;
  with function Next0(S : Sequence0)
    return Sequence0;
  with function Construct0(E : Element0;
    S : Sequence0) return Sequence0;
  with procedure Set_First0(S : Sequence0;
    E : Element0);
  with procedure Set_Next0(S1, S2 : Sequence0);
  with procedure Free0(S : Sequence0);
package Singly_Linked_Lists is

  function Invert_Copy(S : Sequence)
    return Sequence;
  -- The result is a new sequence containing
  -- the same elements as S, but in reverse order.

  function Reverse_Append(S1, S2 : Sequence)
    return Sequence;
  -- Returns a sequence consisting of the
  -- elements of S1, in reverse order,
  -- followed by those of S2 in order.

  generic
  with function Test(X, Y : Element) return Boolean;
  function Count(Item : Element; S : Sequence)
    return Integer;
  -- Returns a non-negative integer, the number
  -- of elements e of S such that Test(Item,E)
  -- is true.

  generic
  with function Test(X : Element) return Boolean;
  function Count_If(S : Sequence) return Integer;
  -- Returns a non-negative integer, the number
  -- of elements e of such that Test(E) is true.

  generic
  with function Test(X, Y : Element)
    return Boolean;
  function Delete_Duplicates(S : Sequence)
    return Sequence;
  -- Returns a sequence of the elements of S but
  -- with only one occurrence of each, using
  -- Test(X,Y) as the test for equality.
  -- S is destroyed.

      . . .

end Singly_Linked_Lists;
```

Here we have listed only a few of the 66 subprograms in the package.

Body of Singly_Linked_Lists

The package body illustrates how many of the subprograms can be built with different instantiations of basic algorithms from Linked_List_Algorithms:

```
with Linked_List_Algorithms;
package body Singly_Linked_Lists is

  function Copy_Cell(S1, S2 : Sequence)
    return Sequence is
  begin
    return Construct(First(S1), S2);
  end Copy_Cell;

  pragma Inline(Copy_Cell);

  package Algorithms is new
    Linked_List_Algorithms(Cell => Sequence,
      Next => Next, Set_Next => Set_Next,
      Is_End => Is_End, Copy_Cell => Copy_Cell);

  generic
    Item : Element;
    with function Test(X, Y : Element)
      return Boolean;
  function Make_Test(S : Sequence) return Boolean;

  function Make_Test(S : Sequence)
    return Boolean is
  begin
    return Test(Item, First(S));
  end Make_Test;

  pragma Inline(Make_Test);

  generic
    with function Test(X : Element)
      return Boolean;
  function Make_Test_If(S : Sequence)
    return Boolean;

  function Make_Test_If(S : Sequence)
    return Boolean is
  begin
    return Test(First(S));
  end Make_Test_If;

  pragma Inline(Make_Test_If);

  generic
    with function Test(X : Element)
      return Boolean;
  function Make_Test_If_Not(S : Sequence)
    return Boolean;

  generic
    with function Test(X, Y : Element)
      return Boolean;
  function Make_Test_Both(S1, S2 : Sequence)
    return Boolean;

  function Make_Test_Both(S1, S2 : Sequence)
    return Boolean is
  begin
    return Test(First(S1), First(S2));
  end Make_Test_Both;

  pragma Inline(Make_Test_Both);
```

```
    function Invert_Copy(S : Sequence)
        return Sequence is
    begin
      return Reverse_Append(S, Nil);
    end Invert_Copy;

    function Count(Item : Element; S : Sequence)
          return Integer is
      function Test_Aux
        is new Make_Test(Item, Test);
      function Count_Aux
        is new Algorithms.Count(Test_Aux);
    begin
      return Count_Aux(S);
    end Count;

    function Count_If(S : Sequence)
          return Integer is
      function Test_Aux is new Make_Test_If(Test);
      function Count_Aux
        is new Algorithms.Count(Test_Aux);
    begin
      return Count_Aux(S);
    end Count_If;

    function Delete_Duplicates(S : Sequence)
          return Sequence is
      function Test_Aux is new Make_Test_Both(Test);
      function Delete_Aux is new
        Algorithms.Delete_Duplicates(Test_Aux,Free);
    begin
      return Delete_Aux(S);
    end Delete_Duplicates;

        . . .

end Singly_Linked_Lists;
```

Specification of System_Allocated_Singly_Linked

Now we introduce an actual data representation for singly-linked lists, by first creating a package defining just a simple record structure and corresponding subprograms for creating and accessing this structure. In this package the standard heap allocation and deallocation procedures are used, but our library also includes two other packages defining data representations that provide a more elaborate allocation method for the same representation.

```
  with Linked_Exceptions;
  generic
    type Element is private;
  package System_Allocated_Singly_Linked is
    type Sequence is private;
    Nil : constant Sequence;
    function First(S : Sequence) return Element;
    function Next(S : Sequence) return Sequence;
    function Construct(The_Element : Element;
      S : Sequence) return Sequence;
    procedure Free(S : Sequence);
    procedure Set_First(S : Sequence; X : Element);
    procedure Set_Next(S, X : Sequence);
    pragma Inline(First, Next, Construct, Free,
      Set_First, Set_Next);
    First_Of_Nil : exception
        renames Linked_Exceptions.First_Of_Nil;
    Set_First_Of_Nil : exception
        renames Linked_Exceptions.Set_First_Of_Nil;
    Next_Of_Nil : exception
        renames Linked_Exceptions.Next_Of_Nil;
    Set_Next_Of_Nil : exception
        renames Linked_Exceptions.Set_Next_Of_Nil;
    Out_Of_Construct_Storage : exception
        renames
        Linked_Exceptions.Out_Of_Construct_Storage;
  private
    type Node;
    type Sequence is access Node;
    Nil : constant Sequence := null;
  end System_Allocated_Singly_Linked;
```

Body of System_Allocated_Singly_Linked

```
  with Unchecked_Deallocation;
  package body System_Allocated_Singly_Linked is
    type Node is record
      Datum : Element;
      Link  : Sequence;
    end record;

    function First(S : Sequence)
      return Element is
    begin
      return S.Datum;
    exception
      when Constraint_Error =>
        raise First_Of_Nil;
    end First;

    function Next(S : Sequence)
      return Sequence is
    begin
      return S.Link;
    exception
      when Constraint_Error =>
        raise Next_Of_Nil;
    end Next;

    function Construct(The_Element : Element;
      S : Sequence) return Sequence is
    begin
      return new Node'(The_Element, S);
    exception
      when Storage_Error =>
        raise Out_Of_Construct_Storage;
    end Construct;

    procedure Free_Aux is new
      Unchecked_Deallocation(Node, Sequence);

    procedure Free(S : Sequence) is
      Temp : Sequence := S;
    begin
      Free_Aux(Temp);
    end Free;

    procedure Set_First(S : Sequence;
      X : Element) is
    begin
      S.Datum := X;
    exception
      when Constraint_Error =>
        raise Set_First_Of_Nil;
    end Set_First;
```

```
  procedure Set_Next(S, X : Sequence) is
  begin
    S.Link := X;
  exception
    when Constraint_Error =>
      raise Set_Next_Of_Nil;
  end Set_Next;
end System_Allocated_Singly_Linked;
```

Partially Instantiated Package (PIP)

Next comes the step of plugging these packages together, in a Partially Instantiated Package that has only the Element type as a generic parameter:

```
with Singly_Linked_Lists;
with System_Allocated_Singly_Linked;
generic
  type Element is private;
package System_Allocated_Singly_Linked_Lists is
  package Records is new
    System_Allocated_Singly_Linked(Element);
  use Records;
  package Lists is new
    Singly_Linked_Lists(Element, Sequence, Nil,
      First, Next, Construct, Set_First,
      Set_Next, Free);
end Simple_Singly_Linked;
```

Instantiation with type Integer

Note that we have a package that is still generic in the type of elements stored as data in the lists. To illustrate how to make use of the list processing capabilities that have been built up, we next carry out an instantiation of the element type as Integer.

```
with System_Allocated_Singly_Linked_Lists;
package Integer_Linked is new
  System_Allocated_Singly_Linked_Lists(Integer);
```

Test suite for Singly_Linked_Lists

We show a small part of an extensive test suite we have developed for Singly_Linked_Lists, using the Integer_Linked instance.

```
with Integer_Linked; use  Integer_Linked;
with Text_Io; use Text_Io;
procedure Examples is
  use  Lists;
  Flag : Boolean := True;

  function Shuffle_Test(X, Y : Integer)
    return Boolean is
  begin
    Flag := not Flag;
    return Flag;
  end Shuffle_Test;

  function Iota(N : Integer) return Sequence is
    Result : Sequence := Nil;
  begin
    for I in reverse 0 .. N - 1 loop
      Result := Construct(I, Result);
    end loop;
    return Result;
  end Iota;

-- I/O functions

  procedure Print_Integer(I : in Integer) is
  begin
    Put(Integer'Image(I));
    Put(" ");
  end Print_Integer;

  procedure Print_List(S : Sequence) is
    procedure Print_List_Aux
      is new For_Each(Print_Integer);
  begin
    Print_List_Aux(S);
  end Print_List;

  procedure Show_List(S : Sequence) is
  begin
    Print_List(S);
    New_Line;
  end Show_List;

  procedure Show(The_String : String) is
  begin
    Put(The_String);
    New_Line;
  end Show;

-- Little functions needed to construct examples

  function Divides(A, B : Integer)
    return Boolean is
  begin
    return B mod A = 0;
  end Divides;

  function Odd(A : Integer) return Boolean is
  begin
    return not Divides(2, A);
  end Odd;

 begin
    Show("Invert(Iota(6))");
    Show_List(Invert(Iota(6)));
    Show("Reverse_Append(Iota(5),Iota(6))");
    Show_List(Reverse_Append(Iota(5), Iota(6)));

    declare
      function Count_When_Divides
        is new Lists.Count(Test => Divides);
      function Count_If_Odd
        is new Count_If(Test => Odd);
    begin
      Show("Count_When_Divides(3, Iota(10))");
      Print_Integer(Count_When_Divides(3, Iota(10)));
      New_Line;
      Show("Count_If_Odd(Iota(9))");
      Print_Integer(Count_If_Odd(Iota(9)));
      New_Line;
    end;

    declare
      function Delete_Duplicates_When_Divides
        is new Delete_Duplicates(Test=>Divides);
    begin

Show("Delete_Duplicates_When_Divides(Next(Next(Iota(20))))");

Show_List(Delete_Duplicates_When_Divides(Next(Next(Iota(20)))));
      New_Line;
```

```
  end;
end Examples;
```

Output from the tests

```
Invert(Iota(6))
 5  4  3  2  1  0
Reverse_Append(Iota(5), Iota(6))
 4  3  2  1  0  0  1  2  3  4  5

Count_When_Divides(3, Iota(10))
 4
Count_If_Odd(Iota(9))
 4

Delete_Duplicates_When_Divides(Next(Next(Iota(20))))
 2  3  5  7  11  13  17  19
```

Though limited to a thin vertical slice of the library structure, these examples show many of the features of our approach and the potential for a programmer to make use of different abstraction mechanisms; e.g., to use the `Singly_Linked_List` package in conjunction with his or her own more complex record structure to produce a large collection of useful algorithms operating on that structure, just by plugging them together as we have done to produce the `System_Allocated_Singly_Linked_Lists` package.

Representational abstraction example

In order to illustrate representational abstractions, we give the following treatment of stacks:

```
generic
  type Element  is private;
  type Sequence is private;
  with function Full(S : Sequence) return Boolean;
  with function Empty(S : Sequence) return Boolean;
  with function First(S : Sequence) return Element;
  with function Next(S : Sequence) return Sequence;
  with function Construct(E : Element; S : Sequence)
    return Sequence;
  with procedure Free_Construct(S : Sequence);
package Stacks is
  type Stack is limited private;
  procedure Push(The_Element : in Element;
                 S : in out Stack);
  procedure Pop(The_Element : out Element;
                S : in out Stack);
  function Top(S : Stack) return Element;
  function Is_Empty(S : Stack) return Boolean;
  pragma Inline(Push, Pop, Top, Is_Empty);
  Stack_Underflow, Stack_Overflow : exception;
private
  type Stack is new Sequence;
end Stacks;

package body Stacks is

  procedure Push(The_Element : in Element;
                 S : in out Stack) is
  begin
    if Full(Sequence(S)) then raise Stack_Overflow;
    end if;
    S := Stack(Construct(The_Element, Sequence(S)));
  end Push;

  procedure Pop(The_Element : out Element;
                S : in out Stack) is
    Old : Sequence := Sequence(S);
  begin
    The_Element := Top(S);
    S := Stack(Next(Sequence(S)));
    Free_Construct(Old);
  end Pop;

  function Top(S : Stack) return Element is
  begin
    if Is_Empty(S) then raise Stack_Underflow;
    end if;
    return First(Sequence(S));
  end Top;

  function Is_Empty(S : Stack) return Boolean is
  begin
    return Empty(Sequence(S));
  end Is_Empty;

end Stacks;
```

Here we have created a stack structural abstraction by a simple mapping that allows the operations of a sequence structural abstraction to be used to implement those of stacks. Again, we emphasize that this approach yields a whole family of stack data abstractions, one for each possible sequence data abstraction, including all vector as well as linked list representations.