Introduction to Stephe's Ada Library

Stephen Leake
stephen_leake@acm.org

Abstract

Stephe's Ada Library (SAL) containers are designed to support the widest possible range of item and container types (limited/non-limited, indefinite/definite, etc). SAL containers are defined by the operations required to implement algorithms. Issues that arise in the implementation of this design are discussed.

1 Introduction

Stephe's Ada Library (SAL) is a collection of stuff I've found useful in my projects. It is all available on line at `http://www.toadmail.com/~ada_wizard/`, under the GNAT-modified Gnu General Public License (the same license used for the runtime libraries of GNAT, the Gnu Ada compiler).

Some of the packages implement a container library, others implement a robotics and spacecraft mathematics library, and the rest don't fit in any particular category.

The math library provides operations for kinematics and dynamics of masses in 3 dimensional space. Cartesian vectors, rotation quaternions, orthonormal rotation matrices, moments of inertia, forces, acceleration, velocity are supported, in 3 and 6 degrees of freedom (translation and rotation). I've used this library for both robotics and satellite simulation. The core algorithms are documented in `http://www.toadmail.com/~ada_wizard/ada/spacecraft_math.pdf`.

This paper presents an overview of the container library design.

All examples in this paper are drawn from the tests for SAL, available with the SAL source. The examples have been simplified from the full test code, and as such will not compile directly; the full test code does of course compile and run.

2 Containers

2.1 Philosophy

A large part of SAL provides a set of container types, intended to be another entry in the "Standard Ada Library" discussion. My goal in this part of SAL

was to provide Ada packages that take full advantage of the flexibility and power of Ada generics, while at the same time making things relatively easy for the casual user.

In particular, I wanted to provide container types and algorithms that could work with the widest possible range of Ada types. I did not want to build a direct replacement for the C++ Standard Template Library; I wanted to see what I could accomplish by pushing the generics in Ada as far as I could. I did study the C++ STL for ideas, and used the same names for operations where that makes sense.

One basic decision is that there is no root container type, nor a root object type. Those are necessary in a language like Java that has no generics; they are not necessary in Ada.

Another basic decision is that algorithms define what operations a container needs; any data type that provides the operations needed by a particular algorithm is a "SAL container" as far as that algorithm is concerned. This differs from C++ STL, which first defines a hierarchy of containers, and then provides algorithms that work with that hierarchy.

On the other hand, there are not many algorithms implemented in SAL. So it may turn out that this is a bad idea :). The algorithms implemented so far are the ones I've needed in my projects.

Finally, since I implemented SAL in order to use it in other projects, and I have limited time, I designed each package to have maximum reuse. For example, there is only one tagged list package, not one for definite items and another for indefinite. This reduces the amount of testing and maintenance, at the cost of some run-time efficiency.

2.2 Item types

Ada provides a wide range of type categories; definite/indefinite, limited/non-limited, tagged/non-tagged, abstract/concrete. By contrast, C++ classes can only be abstract or concrete; there are no other choices. A C++ class corresponds to an Ada non-limited tagged type, either abstract or concrete. This range of Ada types complicates the declaration of container packages; we want to allow the item type for container packages to be any possible Ada type. In practice, this is not possible, but it is possible to allow the item type to cover most of the range.

One consequence of this design style is that we only need one implementation of each container type; it can handle any category of item type, with reasonable efficiency. This is a large benefit when the container library is being implemented by a single person, while working on other real projects. It allows more complete

testing of the containers, ensuring high quality.

An example of the SAL container package style is given by the doubly linked list package:

```
with Ada.Finalization;
with System.Storage_Pools;
generic
   type Item_Type (<>) is limited private;
   type Item_Node_Type is private;
   with function To_Item_Node (Item : in Item_Type) return Item_Node_Type;
   with procedure Free_Item (Item : in out Item_Node_Type);
   with function Copy (Source : in Item_Node_Type) return Item_Node_Type is <>;
   Node_Storage_Pool : in out System.Storage_Pools.Root_Storage_Pool'Class;
package SAL.Poly.Lists.Double is
   type List_Type is new Ada.Finalization.Controlled with private;

   procedure Add (List : in out List_Type; Item : in Item_Type);
   --  Add To_Item_Node (Item) to tail of List.

   ...

end SAL.Poly.Lists.Double;
```

The full package provides many more operations than Add, of course. It defines all operations that are reasonable for lists. See the source code for the complete list; this paper focuses on the issues around `Item_Type` categories.

Starting with the most complex case, here is a list of indefinite limited items:

```
package Puppets is
  type Puppet_Label_Type is (Muppet, Beanie);

  subtype Puppet_Parameters_Type is String;

  type Puppet_Type (Label : Puppet_Label_Type) is limited record
     case Label is
     when Muppet =>
        Arms    : Integer;
        Fingers : Integer;
     when Beanie =>
        Legs    : Integer;
     end case;
  end record;

  type Puppet_Access_Type is access Puppet_Type;
```

```
  procedure Initialize
     (Item       : in out Puppet_Type;
      Parameters : in     Puppet_Parameters_Type);

  function Allocate
     (Parameters : in Puppet_Parameters_Type)
     return Puppet_Access_Type;

  function Copy_Puppet
     (Item : in Puppet_Access_Type)
     return Puppet_Access_Type;

  package Lists_Aux is new SAL.Aux.Indefinite_Limited_Items
     (Create_Parameters_Type => Puppet_Parameters_Type,
      Limited_Type           => Puppet_Type,
      Item_Access_Type       => Puppet_Access_Type,
      Allocate_Item          => Allocate,
      Initialize_Item        => Initialize);

  procedure Free_Puppet is new Ada.Unchecked_Deallocation
     (Puppet_Type, Puppet_Access_Type);

  package Lists is new SAL.Poly.Lists.Double
     (Item_Type         => Puppet_Parameters_Type,
      Item_Node_Type    => Puppet_Access_Type,
      To_Item_Node      => Lists_Aux.To_Item_Node,
      Free_Item         => Free_Puppet,
      Copy              => Copy_Puppet,
      Node_Storage_Pool => ... );

end Puppets;
```

Obviously `Puppet_Type` is a toy type. But I have found this style useful in GUI applications, where `Item_Type` is a limited window type containing a display of items in a sorted list; `SAL.Gen.Alg.Find_Linear.Sorted` is used to maintain the sorted list.

Here is the body of `Add`:

```
procedure Add (List : in out List_Type; Item : in Item_Type)
is
  New_Node : constant Node_Access_Type := new Node_Type'
     (Item => To_Item_Node (Item),
      Prev => List.Tail,
      Next => null);
```

```
begin
  if List.Tail = null then
     List.Head := New_Node;
     List.Tail := New_Node;
  else
     List.Tail.Next := New_Node;
     List.Tail      := New_Node;
  end if;
end Add;
```

Note that this is quite simple; it deals only with the details of maintaining a doubly linked list. All the complexity of indefinite limited types is handled by the generic formal function `To_Item_Node`. This is a primary feature of the SAL container design; the containers themselves are simple to implement and test, yet they still support a very wide range of Ada types.

Here is the body of `Lists_Aux.To_Item_Node`:

```
function To_Item_Node
   (Parameters : in Create_Parameters_Type)
   return Item_Access_Type
is
  Temp : constant Item_Access_Type := Allocate_Item (Parameters);
begin
  Initialize_Item (Temp.all, Parameters);
  return Temp;
end To_Item_Node;
```

Again, a simple function; complexity is delegated to the generic formals `Allocate_Item` and `Initialize_Item`.

When the user calls `Add (List, ("Muppet 2 5"))`, `Lists_Aux.To_Item_Node` is called. It calls `Puppets.Allocate`, which reads the puppet label from the parameter string, and allocates an appropriately constrained `Puppet_Type` object. Then `Lists_Aux.To_Item_Node` calls `Puppets.Initialize`, which reads the rest of the parameters and initializes the fields of the `Puppet_Type` object.

Thus we initialize the limited object in place. The lists container provides access functions returning a pointer to the object (of type `Item_Node_Type`), allowing normal operations on the object (in the GUI case, showing the window, operating on child windows, etc).

For a less complicated example, here is a list of indefinite non-limited items:

```
package Symbols is
  type Symbol_Label_Type is (Floating_Point, Discrete);
```

```
  type Symbol_Type (Label : Symbol_Label_Type) is record
     case Label is
     when Floating_Point =>
        Significant_Digits : Natural;
     when Discrete =>
        First : Integer;
        Last  : Integer;
     end case;
  end record;

  type Symbol_Access_Type is access Symbol_Type;

  package Lists_Aux is new SAL.Aux.Indefinite_Private_Items
     (Item_Type        => Symbol_Type,
      Item_Access_Type => Symbol_Access_Type);

  procedure Free_Symbol is new Ada.Unchecked_Deallocation
     (Symbol_Type, Symbol_Access_Type);

  package Lists is new SAL.Poly.Lists.Double
     (Item_Type         => Symbol_Type,
      Item_Node_Type    => Symbol_Access_Type,
      To_Item_Node      => Lists_Aux.To_Item_Node,
      Free_Item         => Free_Symbol,
      Copy              => Lists_Aux.Copy,
      Node_Storage_Pool => ... );

end Symbols;
```

There are significantly fewer helper functions required. The list container manages the allocation and freeing of `Symbol_Type` objects. `Lists_Aux.To_Item_Node` simply allocates one `Symbol_Type` object, initialized by the input parameter.

For the simplest kind of list, here is a linked list of a definite non-limited item type:

```
package Lists_Aux is new SAL.Aux.Definite_Private_Items (Integer);
package Lists is new SAL.Poly.Lists.Double
 (Item_Type         => Integer,
  Item_Node_Type    => Integer,
  To_Item_Node      => Lists_Aux.To_Item_Node,
  Free_Item         => Lists_Aux.Free_Item,
  Copy              => Lists_Aux.Copy,
  Node_Storage_Pool => ... );
```

In this case, `Item_Type` is the same as `Item_Node_Type`, `To_Item_Node` just returns Item, and `Free_Item` does nothing. With proper in-lining, this design should impose little overhead compared to a less flexible package design.

To investigate this claim, I implemented a simple lists package that assumed definite non-limited types, and provides only Add and Delete operations. The times for the combination Add, Delete for the two packages, with and without optimization, are:

Design	Unoptimized	Optimized
Simple	546	514
Poly	586	559

The times are in nanoseconds, measured on a 2.0 GHz Intel Pentium 4M running Windows 2000. These results are somewhat ambiguous; optimization improved the simple design as much as it did the complex design, and the difference between the designs is about the same size as the optimization gain. In addition, it is difficult to get repeatable timing results on Windows. Since the gain in speed is small, I suggest these results support my decision that it is not worth writing several container packages, each for a particular category of Ada type.

2.3 Iterators

All container packages provide iterators, to allow processing the items in the container. The iterators in SAL are not "safe"; deleting an item via one iterator may invalidate other iterators, leading to dangling pointers and erroneous execution. In practice I rarely have more than one iterator open on a container, so this is not an issue. The overhead required to make iterators truly safe is not worth it.

Iterators are also known as "cursors". There are formal definitions of both terms that distinguish them; I won't go into that here. I used the term "iterators" following the C++ STL. The next version of the Ada language will most likely have a container library (see AI-302); the proposed library uses the term "cursors", following standard database usage.

Here's a simple example of using an iterator to visit all items in a list:

```
procedure Process_List (List : in List_Type)
is
   Iterator : Iterator_Type := First (List);
begin
   loop
      exit when Is_Null (Iterator);
      Do_Something (Current (Iterator));
```

```
      Next (Iterator);
   end loop;
end Process_List;
```

Iterators for other containers follow the same design pattern.

In general, iterators for container types are declared in the main container package. If the Poly iterators were in a child package, they would have to take class-wide arguments, and they would not match the profiles required by the generic algorithms packages. An exception is `SAL.Poly.Binary_Trees.Sorted`; iterators for binary trees are not often needed.

Iterators generally do not contain a reference to a container (one exception is `SAL.Aux.Enum_Iterators`, which provides iterators for plain Ada arrays). This greatly simplifies declaring iterators, which would otherwise almost always require `'Unchecked_Access`. Alternately, all containers would be forced to use a layer of indirection, which would be unnecessary overhead for applications that don't use iterators. On the other hand, it means the iterator operations cannot enforce the requirement that iterators be used with only one container. It also means the iterator cannot tell if the container has been deleted. Again, SAL favors flexibility over strict control.

Iterators are non-limited, so they can be copied. This can lead to dangling pointers if abused, but it greatly simplifies writing algorithms.

2.4 Implementation

Since Ada allows abstract data types that are tagged or not, and in some applications tagged types are not desired, I started out designing SAL to support both kinds of container types. The child package tree rooted at `SAL.Gen` (for generic) provides non-tagged container data types, while the tree rooted at `SAL.Poly` provides tagged (polymorphic) container data types. However, I only implemented a doubly-linked list in the `SAL.Gen` tree; it quickly became clear that there was little reason not to use tagged containers.

Here are the tagged (polymorphic) container data types provided by SAL:

`SAL.Poly.Binary_Trees.Sorted` Sorted binary trees for indefinite limited types. These are simple binary trees, not AVL or Red-Black. So far, I've found that the applications for which the extra efficiency matters actually want a hash table anyway. The child package `Iterators` provides in-order traversal of the tree.

`SAL.Poly.Lists.Double` Doubly-linked lists, using Ada.Finalization to manage allocated memory.

`SAL.Poly.Lists.Single` Singly-linked lists, using Ada.Finalization to manage allocated memory. Provided for completeness, and to explore the details of what operations are possible on singly versus doubly linked lists. There is very little reason to use a singly-linked list instead of a doubly-linked list; if you really need to save that amount of storage, you probably need a custom container anyway.

`SAL.Poly.Unbounded_Arrays` Unbounded arrays of indefinite types. The arrays grow and shrink as items are added and deleted. They can grow at either end.

One obvious container type missing is a hash table. I do have one that works with SAL, but it is not licensed under the GMGPL (it was written at work), so it is not part of SAL. One of these days, I'll get around to rewriting it under the GMGPL.

SAL also provides stacks and queues, which are implemented with containers or plain arrays.

`SAL.Gen.Stacks.Bounded_Nonlimited` Stacks of non-limited types, with a fixed maximum size. Implemented with a plain array. Used in `SAL.Poly.Binary_Trees.Sorted.Iterators` to keep a stack of node pointers.

`SAL.Gen.Stacks.Bounded_Limited` stacks of limited types (matching the standard SAL container style), with a fixed maximum size. Implemented with a plain array.

`SAL.Poly.Stacks` Root tagged stack type, allowing indefinite element types.

`SAL.Poly.Stacks.Unbounded_Array` polymorphic stacks of indefinite types, implemented with an unbounded array so there is no fixed maximum size.

`SAL.Gen.Queues.Gen_Bounded_Nonlimited` Fixed-size queues of definite items; does not follow the standard SAL style (it was written for a special purpose). Implemented with a plain array.

2.5 Storage Pools

Each package that does allocation takes a generic `Storage_Pool` parameter. This allows users to implement their own storage pools, and still use SAL. I use this capability in the tests to show that deallocation is done properly. Ada 95 does not allow a default parameter for `Storage_Pool`, so even novice users must provide a storage pool parameter. Unfortunately, this will not be fixed in the upcoming revision of Ada 95 (AI-300 proposed a solution, but was not adopted).

However, there is a simple way to provide the required parameter: `System.Storage_Pools.Root_Storage_Pool'Class (<some_global_access_type>'Storage_Pool)`. `<some_global_access_type>` can be `Ada.Strings.Unbounded.String_Access`, or some user-defined access type.

3 Algorithms

One purpose of a coherent set of abstract data types is to provide algorithms that can be used with them. SAL has a small set of algorithm packages, rooted at `SAL.Gen.Alg`. The root package takes generic formal parameters that are common to all the algorithms in the tree; child packages may require additional functions (such as comparison). In one sense, it is this root algorithm package that defines a SAL container; any abstract data type that can be used with `SAL.Gen.Alg` is a SAL container. This contrasts with the C++ Standard Template Library approach of defining a class hierarchy, and algorithms that work with various levels in the hierarchy.

I started a parallel set of polymorphic algorithm packages, allowing abstract container types, but they turned out not to be necessary; the SAL.Gen.Alg packages meet all my needs so far.

Here is the root `SAL.Gen.Alg` package specification:

```
generic
   type Item_Node_Type is private;
   type Container_Type (<>) is limited private;

   type Iterator_Type (<>) is private;
   -- Iterator_Type is indefinite, because some require a discriminant
   -- of Container'access. Initializing one with None should be fast.

   with function Current (Iterator : in Iterator_Type)
      return Item_Node_Type is <>;
   with function First (Container : in Container_Type)
      return Iterator_Type is <>;
   with function Last (Container : in Container_Type)
      return Iterator_Type is <>;
   with function None (Container : in Container_Type)
      return Iterator_Type is <>;
   with function Is_Null (Iterator : in Iterator_Type)
      return Boolean is <>;
   with procedure Next_Procedure (Iterator : in out Iterator_Type);
   with function Next_Function (Iterator : in Iterator_Type)
```

```
      return Iterator_Type;
package SAL.Gen.Alg is
   pragma Pure;

   procedure Do_Nothing (Container : in Container_Type);
   pragma Inline (Do_Nothing);
   -- For algorithm defaults.

end SAL.Gen.Alg;
```

Here is the spec of the count algorithm, which simply counts the items in a container:

```
generic
function SAL.Gen.Alg.Count (Container : in Container_Type) return Natural;
```

Instantiating `SAL.Gen.Alg.Count` with the integer `Lists` package above is straightforward:

```
package Algorithms is new SAL.Gen.Alg
  (Item_Node_Type => Integer,
   Container_Type => Lists.List_Type,
   Iterator_Type  => Lists.Iterator_Type,
   Current        => Lists.Current,
   First          => Lists.First,
   Last           => Lists.Last,
   None           => Lists.None,
   Is_Null        => Lists.Is_Null,
   Next_Procedure => Lists.Next,
   Next_Function  => Lists.Next);

function Count_Integers is new Algorithms.Count;
```

Here are the algorithms provided by SAL:

`SAL.Gen.Alg.Count` Counts the items in a container.

`SAL.Gen.Alg.Find_Binary` Finds items in a container, using binary search. The container must have an ordering that permits computing an iterator that is halfway between two other iterators.

`SAL.Gen.Alg.Find_Linear.Sorted` Searching and sorting in a linear container (such as a linked list). Only merge sort is supported.

`SAL.Gen.Alg.Find_Linear` Searching in an unordered container (just compare to every item). Useful for small containers that are searched rarely.

`SAL.Gen.Alg.Process_All_Constant` Apply a user-defined procedure to every item. Useful for printing out containers.

`Gen.Alg.Container_Type` is indefinite, to allow more general container types. This means some algorithms must take an extra parameter to use for temporary storage. Consider `Gen.Alg.Find_Linear.Sorted.Sort`:

```
procedure Sort
  (Container      : in out Container_Type;
   Temp_Container : in out Container_Type)
is begin
   Remove_Out_Of_Order (Container, Temp_Container);
   Merge (Container, Temp_Container);
end Sort;
```

`Remove_Out_Of_Order` makes one pass through `Container`, removing items that are out of order, and storing them in `Temp_Container`, in order. Then `Merge` merges the two sorted containers. This is efficient if the original container is mostly sorted.

If `Container_Type` was not indefinite, `Temp_Container` could be declared as a local variable. An alternate design would be to ask the user to provide an appropriate initial value for the temporary container. The current design gives the user better control over memory usage.

`Gen.Alg` supports non-limited containers, and thus requires a Copy operation for all items. When instantiating with limited items that truly should not be copied, Copy can raise `SAL.Invalid_Operation`. An alternate design would be to provide both `Gen.Alg_Limited` and `Gen.Alg_Non_Limited`; that requires maintaining two copies of each algorithm, which is too high a burden.

In sorted containers, the key is assumed stored in `Item_Type`, accessed via the generic formal function `To_Key` and generic formal comparison functions. This allows for keys that are complex functions of `Item_Type`, and allows for caching keys.

`SAL.Gen.Alg.Find_Linear.Key_Type` is indefinite limited private to support keys like:

```
type Keys_Type (Label : Label_Type) is record
   case Label is
   when Muppet =>
      Arms    : Integer;
      Fingers : Integer;
   when Beanie =>
      Legs : Integer;
```

```
   end case;
end record;
```

This means we also require several versions of `Is_Equal`, rather than a single `To_Key`.

4 Conclusion

I've presented some of the design philosophy behind SAL containers and algorithms, and discussed some of the implementation decisions.

SAL has proved invaluable in several applications, saving implementation time by providing ready-to-use containers.