Avoiding Access Types

Jeffrey R. Carter
PragmAda Software Engineering; email: jrcarter@acm.org; https://github.com/jrcarter

Abstract

This paper is a summary of the presentation at the 2024 Ada-Europe International Conference Ada Developers Workshop. Access types and their associated memory management are a constant source of errors, so it is desirable to avoid them. Access types are very rarely needed. Some techniques for avoiding their use are presented.

Keywords: access types, memory management, Ada.

1 Introduction

Ada has two types of access types: access-to-object types and access-to-subprogram types. Here we are dealing with access-to-object types and will call them "access types" for short.

Access types and their associated memory management are a constant source of errors, so it is desirable to avoid them. In Ada, access types are never needed (valid as a first-order approximation, and probably as second). Most uses can be replaced with unbounded components from the standard library; most remaining cases can be encapsulated and hidden within a custom component. Encapsulation and hiding make it easier to get the memory management right.

There are three main cases where access types can be avoided:

- Returning large objects with unknown constraints
- Making private types private
- Self-referential types

2 Returning large objects

Sometimes it is necessary to return an object with constraints that are not known at the point of the call. Typically, this is provided by a function that returns an unconstrained type:

```
type T is array
   (Positive range <>, Positive range <>) of Thing;
function Read (File_Name : in String) return T;
```

Depending on what T represents, the result may be arbitrarily large, so it must be allocated on the heap. But then there is a memory-management issue: how to return the result value and still deallocate the allocated memory.

A common solution is to return the access value, but using access types in the visible part of package specifications is poor design, and forcing the client to do the memory management, as this would, is very poor design.

We could use a smart pointer internally, which would automatically deallocate the memory after the return expression has been evaluated, but then there is still the problem of what the caller can do with such a large object, often leading to the use of access types by the client. We would like to avoid that.

A solution which avoids all these problems is to return a holder:

```
package T_Holders is new
   Ada.Containers.Indefinite_Holders
      (Element_Type => T);
procedure Read
   (File_Name : in String;
    Item : in out T_Holders.Holder);
...
declare
   type T_Ptr is access T;
   Ptr : T_Ptr := new T (...);
begin
   Item.Replace_Element (New_Element => Ptr.all);
   Free (Ptr);
   -- Operate on Item using Update_Element
exception
when others =>
   Free (Ptr);
end;
```

This involves declaring an access type, allocating the necessary space, copying that space into the holder, and freeing the access value. It is easy to get this memory management correct since the allocation and deallocation occur within a few lines. If the copy proves to have unacceptable consequences, one can create a custom holder to avoid it.

3 Making private types private

In Ada 83, the full type of a private type was only visible within the package. Ada 95 added child packages as a form of programming by extension, allowing visibility of the full type in descendant packages. While this is sometimes desirable, there are also cases when it is not [2].

Consider a large record type with constraints between various components that are difficult to enforce automatically. The package would like all modifications of objects of the type to be done by subprograms provided by the package, which ensure that the constraints hold. If the full type is visible to child packages, even the best-intentioned developer of such a child package might forget and assign directly to a component of the object, violating

the constraints. It is necessary to make the type invisible to child packages to avoid this.

A question on Stack Overflow a few years ago [5] addressed just this situation. The first solution posted (by Jere) used access types, and included memory management. It is instructive to note that the memory management is incorrect, thus demonstrating yet again why access types should be avoided if possible.

My solution used holders, again:

```
private – Parent
  type Root is abstract tagged null record;
  function Equal
    (Left : in Root'Class; Right : in Root'Class)
  return Boolean is
    (Left = Right);
  package Class_Holders is new
    Ada.Containers.Indefinite_Holders
      (Element_Type => Root'Class, "=" => Equal);
  type Item is record
    Value : Class_Holders.Holder;
  end record;
end Parent;
package body Parent is
  type Real_Item is new Root with record
    Value : Boolean;
  end record;
```

The language prevents instantiating Indefinite_Holders with an incomplete private type, leading to the use of a class-wide type. The function Equal is needed because class-wide types have no primitive operations. Since the full type extends Root, it can be stored in the holder; values have to be converted to their actual type when retrieved:

```
R : Real_Item;
V : Item;
...
R.Value := True;
V.Value.Replace_Element (New_Item => R);
...
R := Real_Item (V.Value.Element);
```

4 Self-referential types

A self-referential type is a type that contains components of itself. Access types are often used to implement them; however, even here access types can be avoided.

A common example of a self-referential type is the binary tree, which contains

- A Value
- Left and Right subtrees

Conceptually

```
type Tree is record
  Value : Element;
  Left  : Tree;
  Right : Tree;
end record;
```

which is clearly not valid Ada. Again, holders can be used [1]:

```
type Root is abstract tagged null record;
function Equal
  (Left : in Root'Class; Right : in Root'Class)
return Boolean is
  (Left = Right);
package Tree_Holders is new
  Ada.Containers.Indefinite_Holders
    (Element_Type => Root'Class, "=" => Equal);
type Tree is new Root with record
  Value : Element;
  Left  : Tree_Holders.Holder;
  Right : Tree_Holders.Holder;
end record;
```

Again, the values obtained from the holders must be converted to type Tree.

All of the examples presented here have used holders, but there are cases, especially of self-referential types, for which another container would be better. For example, S-Expressions (usually shortened to SEXes) can contain a list of SEXes. Implementing that using access types would involve implementing a complete list abstraction. Far better to reuse the implementation in Ada.Containers.Indefinite_Doubly_Linked_Lists, using the techniques presented here. I present this in a draft paper [4] and on comp.lang.ada [3].

5 Summary

Access types are often used unnecessarily. Avoiding access types enhances the correctness of programs, since it prevents the errors associated with memory management. Avoiding access types is often less effort than implementing correct memory management.

References

[1] J. Carter, Binary-tree implementation, https://github.com/jrcarter/Binary_Trees.

[2] J. Carter, “Breaking the Ada Privacy Act”, *Ada Letters*, Volume XVI, Number 3 [1996 May/June] [https://github.com/jrcarter/Papers].

[3] J. Carter, Reply to “Recursive algebraic data types”, comp.lang.ada, https://usenet.ada-lang.io/comp.lang.ada/ p7mv5srig1@dont-email.me/ [2018-03-06].

[4] J. Carter, “Self-Referential Data Types Without Access Types”, https://github.com/jrcarter/Papers/ [2021-09-13].

[5] user15552120, “Hiding record from child packages”, https://stackoverflow.com/questions/68838455/hiding-record-from-child-packages/ [2021-08-19]