(This is a draft of a paper I am preparing. It merely presents an implementation technique which helps avoid using access types with their associated memory management and its opportunities for error. As such, I am not sure that it is worth a technical paper. On the other hand, this technique has many applications, so additional examples might be a good ides. For example, a binary tree can be implemented without access types. Comments are welcome.)

Self-Referential Data Types Without Access Types

Jeffrey R. Carter
PragmAda Software Engineering
jrcarter @ acm . org

Abstract

Self-referential types are complex and error-prone to implement. Traditionally they are implemented with access types with the opportunities for errors that they bring. An alternative technique that avoids access types is presented.

1. Introduction

Self-referential type are types that have themselves as components. As such their implementation tends to be error prone, especially in the area of access types and memory management. They are not common, but occur often enough that any technique to simplify their implementation and reduce the opportunities for error is welcome. An example of a self-referential type is presented, along with the traditional ways to implement it, that use access types and their associated memory-management problems. A desirable way to implement it, using containers and avoiding memory-management, but that does not work in Ada, is mentioned. From that a simple, safe way to implement such types is derived.

The technique presented here has previously been shown on the comp.lang.ada newsgroup [Carter 2018]. It is repeated here in hopes that it will reach a wider audience. While I derived it independently, I do not know if I am the first to do so, but I have not seen another presentation of it.

2. S-Expressions

An S-expression, often called a Sex(p[r]) for short, is defined as either an atom or a list of zero or more Sexes. LISP programs, for example, consist of Sexes. As such it is clearly self-referential and an excellent example of such a type. An implementation might be

```
package S_Expressions is
   type Sex (<>) is tagged private;
private -- S_Expressions
   ...
end S_Expressions;
```

3. Traditional Implementation

Traditionally, self-referential types are implemented using access types. The list, which is unbounded, may be implemented as a linked sequence:

```
type Node;
type Link is access Node;

type Sex (Is_Atom : Boolean) is new Ada.Finalization.Controlled with
record
   case Is_Atom is
   when False =>
      List : Link;
   when True =>
      Atom : Atom_Value;
   end case;
end record;
```

with Node completed in the package body as

```
type Sex_Ptr is access Sex;

type Node is record
   Next : Link;
   Data : Sex_Ptr;
end record;
```

or as an indexed sequence:

```
type Sex_Ptr  is access Sex;
type Sex_List is array (Positive range <>) of Sex_Ptr;
type List_Ptr is access Sex_List;

type Sex (Is_Atom : Boolean) is new Ada.Finalization.Controlled with record
   case Is_Atom is
   when False =>
      List : List_Ptr;
   when True =>
      Atom : Atom_Value;
   end case;
end record;
```

The problems with such approaches should be clear: We have to implement a complete unbounded linked or indexed data structure, which is a duplication of the effort already put into Ada's standard container library, and such implementations involve memory-management, which is error prone. The ability to use Ada's standard library would be much simpler and more likely to be correct.

4. An Obvious but Incorrect Approach

Those familiar with Ada's private types will not be surprised that the private part, before the full type definition, can define access types that designate private types, but not composite types with them as component types. This may lead to the thought that they could be used as the element type for an instantiation of an appropriate container:

```
package Lists is new Ada.Containers.Indefinite_Doubly_Linked_Lists
   (Element_Type => Sex):

type Sex (Is_Atom : Boolean) is record
   case Is_Atom is
   when False =>
      List : Lists.List;
   when True =>
      Atom : Atom_Value;
   end case;
end record;
```

An indefinite container seems appropriate since all we can do with the type is declare access types to it, which is what an indefinite container does internally. However, this is not legal Ada. This may be surprising, since a hand instantiation of the package would contain declarations very similar to those in the traditional implementation, but is the case.

5. A Workaround

What we need is a type that is legal to use in the instantiation, which can be used to hold a value of type Sex, but is not an access type. This calls to mind class-wide types, which can store values of any specific type in the class.

```
type Root is abstract tagged null record;

function Equal (Left : in Root'Class; Right : in Root'Class) return Boolean
is
   (Left = Right);

package Lists is new Ada.Containers.Indefinite_Doubly_Linked_Lists
   (Element_Type => Root'Class, "=" => Equal);

type Sex (Is_Atom : Boolean) is new Root with record
   case Is_Atom is
   when False =>
      List : Lists.List;
   when True =>
      Atom : Atom_Value;
   end case;
end record;
```

This is a legal, simple, and safe way to implement self-referential types. Function Equal is needed because class-wide types have no primitive operations, so there is no predefined "=" that matches the default for the generic formal "=". Note that Root cannot be an interface type.

6. The Price

There is no free lunch, and the simplicity and safety we have gained does come at a price. Sex is the only descendant of Root and the only type that will be stored in a list, but the compiler cannot know that, so any value obtained from a list will have type Root'Class, and must be explicitly converted to type Sex to be used.

```
Sex (Item.List.First_Element)
```

This seems like a small price to pay.

7. Conclusion

The technique presented here is simpler and more likely to be correct than the traditional approach to implementing self-referential types. It is a useful addition to the toolkit of any software engineer who has to create them.

Reference

Carter, Jeffrey R., Reply to "Recursive algebraic data types", comp.lang.ada 2018-03-06, https://groups.google.com/g/comp.lang.ada/c/ql9mqvWhnwA/m/NUK7lIWMAAAJ (visited 2021-07-08)