Adding Inheritance to Ada

Jürgen F.H. Winkler

Siemens AG, Corporate Research
Otto-Hahn-Ring 6, D-8000 München 83, FRG
winkler@ztivax.siemens.com

The paper shows how Ada can be turned into an object oriented language by adding package types and an inheritance clause.

Keywords: Ada, Inheritance, Object oriented programming, Class, Subclass, Package type

1 Introduction

Ada provides some elements of object oriented programming (OOP) but it lacks the mechanism of inheritance, which is a very useful mechanism, and which can save a lot of recompilation and retesting effort. There are no good workarounds known, which could be used to simulate inheritance in native Ada. Perez [Per 88] presents one proposal for simulating inheritance in native Ada. His investigation shows that inheritance can only be simulated partially, and that, especially, the introduction of new data components in a subclass is not directly possible. It seems therefore worthwhile to propose language constructs which allow for full inheritance in Ada. This paper is based on a proposal submitted to the Ada9X project [Win 89]. Some ideas are also borrowed from [DSW 90].

The proposal for the introduction of inheritance into Ada is based on the concept of package types (in analogy to task types). A package type has quite the same properties as a class in an object oriented programming language.

Inheritance is introduced via a new clause, the inheritance clause. In this paper we present single inheritance only. Multiple inheritance could also be introduced.

2 Class Definition

We define classes as package types extending the syntax of Ada as depicted in the following example:

```
PACKAGE TYPE Figure IS . . . END Figure ;    (1)
```

This is quite the same syntax as that for task types. The definition (1) defines "Figure" as a private type. In terms of OOP "Figure" can also be called a class. We will use the term package type in the rest of the paper.

Inside the specification part of such a package type a new form of subprogram

declaration can be used:

```
CONSTR Figure (Pos PointTy);                    (2)
```

Such a declaration defines a constructor for the package type in whose specification part it is contained. Such a constructor must have the same identifier as the corresponding package type. Constructors may be overloaded in the same way as subprograms. A constructor is automatically called when a variable of the package type is created. Variables of package types are created in the very same manner as variables in Ada (which are actually called objects):

```
MyFigure : Figure (Pos => (10,20));             (3)
```

In the rest of the paper we call the variables objects as it is done in Ada. The difference between the object declaration (3) and an object declaration in Ada is the fact that the type name after the colon is interpreted as the name of a constructor subprogram. Since a constructor may have parameters we obtain the general form of an object declaration given in (3). This declaration declares an object named "MyFigure" of type "Figure". As part of the elaboration of this declaration the corresponding constructor is called with the given parameter.

Apart from the constructors the specification part of a package type may contain the same declarations as a package specification. Subprograms declared in the public part of a specification part of a package type are typically called methods in the OO world. We call them also methods because they are called in a slightly different way than other subprograms. A method is always called in the context of an object. The package type "Figure" may contain the following method declaration:

```
PROCEDURE Move (To: PointTy);                   (4)
```

The application of the method "Move" to the object "MyFigure" is expressed in the form generally used in the OO world:

```
MyFigure.Move (Vector => (40, 65));             (5)
```

In this form of call the object is a kind of additional and implicit parameter of the method; i.e. there is a special binding between a package type and its methods. This is different from the relation between a package and its public subprograms.

The complete specification part of the package type "Figure" may look like:

```
PACKAGE TYPE Figure
    TYPE Dimensions IS (X, Y);
    Lower_Bound : constant := -100;
    Upper_Bound : constant :=  100;
    TYPE Coordinate_Value IS
        Range Lower_Bound .. Upper_Bound;
    TYPE PointTy IS
        Array (Dimensions) Of Coordinate_Value;

    CONSTR Figure;
    CONSTR Figure (Pos: PointTy);

    FUNCTION Position Return PointTy;
        Pragma Inline(Position);
    PROCEDURE SetPosition (To: PointTy);
    PROCEDURE Move (Vector: PointTy);
PRIVATE
        Center: PointTy;
END  Figure;
```

The body of a package type declaration contains the bodies of the subprograms specified in the specification part and may contain further declarations but it must not contain any statements.

2 Subclass Definition

A package type C can be used as a basis to define further package types which are then a kind of extensions of C. The OO term for such extended package types is subclass.

A subclass of "Figure" can be defined as another package type using an inheritance clause:

```
PACKAGE TYPE Circle IS_A Figure  IS ...        (6)
```

The clause "IS_A Figure" is called the inheritance clause and indicates that "Circle" is a subclass of "Figure". I.e. an object of type "Circle" consists of two data components: "Center" and "Rad", and all subprograms in the visible part of "Figure" are also implicitly contained in the visible part of "Circle". The most important consequence of this property of subclasses is the fact that all methods defined in the superclass may also be applied to an object of the subclass. If we define an object "YourCircle" as follows:

```
YourCircle : Circle ...                        (7)
```

then we can apply the method "Move" also to this object:

```
YourCircle.Move(Vector => (5,5));              (8)
```

A complete definition of the specification of the subclass circle may look like:

```
PACKAGE TYPE Circle IS_A Figure  IS
    TYPE RadiusTy IS Range 0 .. Upper_Bound;
    CONSTR Circle
        (Pos: PointTy; Radius: RadiusTy);
    FUNCTION Radius Return RadiusTy;
    PROCEDURE SetRadius (Radius: RadiusTy);
    PROCEDURE Draw;
PRIVATE
        Rad: RadiusTy;
END  Circle ;
```

3 Assignment Statement

Package types can be used in the same way as other types. Since package types are private types the operations ":=", "=", and "/=" are available. The assignment operation is not only defined for the case where the type of the left hand side and the type of the right hand side are equal but also for the more general case in which the type of the right hand side is a subclass of the type of the left hand side. The effect of the assignment statement is the copying of the values of the data components of the object on the right hand side into the corresponding data components of the object on the left hand side. Therefore the following assignment is legal:

```
MyFigure := YourCircle;                        (9)
```

but not the other way round:

```
YourCircle := MyFigure;                        (10)
```

because there is no "Rad" component in an object of type "Figure".

Some examples for manipulating objects are given in the following procedure "Main".

```
    With Circle, Figure;
PROCEDURE Main IS
    Point1: Figure.PointTy;
    MyCircle: Circle ((5,5), 7);
    YourCircle: Circle ((8, -10), 3);
BEGIN
    MyCircle.Move ((10, 10));
    Point1 := MyCircle.Position;
    YourCircle.Draw;
    YourCircle := MyCircle;
    YourCircle.Draw;
END  Main;
```

3 Summary

By adding one generalization (package type) and one new construct (IS_A clause) Ada has has been turned into a fully object oriented language. Since Ada contains already some elements of object oriented programming the extensions fit into the general framework of the language.

We have not treated the integration of concurrency into the OO framework. The first idea to solve this problem is to introduce also an inheritance clause for task types. But this does not work because the internal structure of Ada tasks is quite different from the internal structure of objects in the sense of OOP. We will investigate this problem further in a future paper.

References

DSW 90 Dießl, Georg; Schulz, Georg; Winkler, Jürgen F. H.: Object-CHILL: The Road to Object Oriented Programming with CHILL. In: Palma, A. (ed.):Proceedings of the 5th CHILL Conference, Rio de Janeiro, March 1990, 118..125.

Per 88 Perez, Eduardo Perez: Simulating Inheritance with Ada. Ada Letters VIII, 5 (1988) pp. 37..46.

Ref 83 Reference Manual for the Ada Programming Language. ANSI / MIL-STD 1815 A. February 1983.

Win 89 Winkler, Jürgen F. H.: Ada9X , Proposal# 8906230125.