NOVI SAD J. MATH.
VOL. 30, NO. 3, 2000, 123-130

A TRULY POLYMORPHIC NUMBER CLASS HIERARCHY

Aleksandar Popović , Mirjana Ivanović
Institute of Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia
{ror, mira}@unsim.ns.ac.yu

Abstract

A class hierarchy for numbers has been implemented in Modula-3. The hierarchy starts with an abstract number class that encompasses common operations for all numbers and ends with concrete classes for integer, rational, real and complex numbers. All numbers have been implemented in ordinary and arbitrary precision. The paper, however, emphasizes the polymorphic nature of the hierarchy. Using the hierarchy it is possible to build procedures and functions that are truly independent of its argument types or function results. To achieve this, several additional procedures had to be implemented using low-level facilities.

AMS Mathematics Subject Classification (1991): 68N20
Key words and phrases: object-orientation, polymorphism, number representation

1. Introduction

The object-oriented methodology in programming is of significant and constantly increasing importance. Programs that are designed using object-oriented methodology, can be more easily maintained and extended.

Generally, the object-oriented methodology offers a lot of new opportunities and advantages in problem solving. However, in some domains disadvantages and limitations are noticeable. One of the obvious problems arises in the initialization of objects, and during the comparison of objects with constant values. Here is a naive and hypothetic example of function for evaluating the factorial of an integer (in this example, `Number` is an ancestor class for both integer class and long- integer class):

```
PROCEDURE Fact( n : Number ) : Number;
BEGIN
  IF Eq( n, 1 ) THEN                                    (*1*)
    RETURN 1;                                           (*2*)
  ELSE
    RETURN Mul( n, Fact( Sub( n, 1 ) ) );
  END;
END Fact;
```

In strongly typed languages [1,2], when n is compared with the constant a (line `(*1*)` in the example), both have to have the same type. Since the type of n is not known at compile time, the constant 1 of the appropriate type cannot be created. Similar holds for the function result type. In the line `(*2*)` of the example, the constant 1 should be returned. However, the constant of the appropriate type cannot be correctly stated, because it is not known in advance of what type (integer or long integer) the result should be.

If the two types of two constants in lines `(*1*)` and `(*2*)` should be known at compile-time, then the function `Fact` would not be polymorphic - the concrete types of argument and the result should be known in advance. Therefore, we would need two functions `Fact` - one for plain integers, and one for long integers.

This paper describes the object model and implementation of the library of routines for multiple precision arithmetic. The point of our work is not algorithmic optimum, but the object concept of sets of different numbers and their hierarchical organization. Beside the object-oriented number model, an effort is made to overcome the problem mentioned above by using low-level facilities of the programming language Modula-3.

In section 2 some basic concepts of number hierarchy are presented. Library models, their organization and global principles of implementation are presented in section 3. Some problems, possible solutions and an example are analyzed in section 4. Finally, some conclusions are drawn in section 5.

2. Basic concepts

The library of routines for multiple precision arithmetic contains 9 modules, which are mutually tightly connected. All together, they form an object hierarchy [4] (Fig.1):

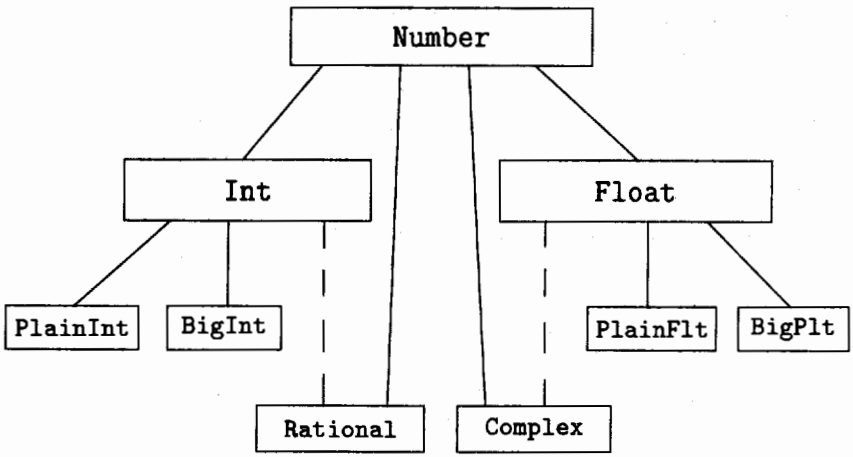

Figure 1.

In Fig. 1, object hierarchy is represented using the standard UML notation. The classes `Number`, `Int` and `Float` contain abstract methods.

Type `Number` contains operations that are common for all numbers:

`Copy`	- makes a copy of the number;
`ToString`	- converts the number to a string;
`SetToZero`	- initializes number's value to zero;
`SetToOne`	- initializes number's value to one;
`Equal`	- returns true if two numbers are equal;
`GEq`	- returns true if the first number is greater or equal to the second one;
`LEq`	- returns true if the first number is less or equal to second one;
`Positive`	- returns true if the number is greater than zero;
`Abs`	- returns an absolute value of the number;
`Minus`	- inverts the sign of the number;
`Zero`	- returns true if the number is zero;

`Add` - adds the first number to the second one;
`Inc` - increases the number;
`Sub` - subtracts the numbers;
`Dec` - decreases the number;
`Mul` - multiplies the numbers;
`Div` - divides the numbers.

The end user has the possibility to use integers, rational, real, and complex numbers, which are distributed in classes that have the same names. The above implementation supports two types of integers: common integers (`PlainInt`) and big integers (`BigInt`). By analogy with integers, this implementation supports two types of real numbers: common real numbers (`PlainFlt`) and big real numbers (`BigFlt`). Since rational and complex numbers contain integers and real numbers respectively, whether they will be "common" or "big" depends on the arguments which are passed during their initialization and use.

Apart from the operations mentioned above, common to all number types, some other operations specific to concrete number types are also implemented. For example, the module `Int` contains methods for calculating the remainder of division the one integer by another (`Mod`), for calculating greatest common divisor and least common of two integers (`GCD` and `LCM`), and others. The module `Float` [5] contains the method for calculating the square root of a real number (`Sqrt`). THe module `Rational` contains the method that reduces a rational number, if possible (`Reduce`).

3. Number class hierarchy and polymorphism

Every object `x` of the class `Number` is now the assignment compatible with every object of the descendant classes. However, as we have explained in the introduction, this is not sufficient for achieving polymorphic procedures of the type `Number` in strongly typed languages.

The use of low-level facilities of the programming language Modula-3 has solved the problem. Modula-3 has two library functions: `ISTYPE` and `TYPECODE`. `ISTYPE` checks if two pointers or object variables have the same type, while `TYPECODE` returns the number which is the internal representation of some pointer or object type. By combining these two functions, it is possible to find the type of some passed argument, if we need it, and to make the required initializations.

Instead of initialization of a concrete object of the known type, an empty object of some type is created, or the copy of the existing object is made. To enable this, two functions are implemented: `EmptyObjectLike` and `Clone`. The first one returns identical copy of an object that was passed to it, while the second one returns an empty object that has identical type like the passed argument. The former calls the library routine `RTHeapAllocate`, which returns the copy of the passed variable, and which has the same type like the variable. The latter one creates first an empty object using `EmptyObjLike`, and then copies the object that is passed to it.

```
PROCEDURE EmptyObjLike( x : T ) : T =
BEGIN
  RETURN RTHeap.Allocate( TYPECODE( x ) );
END EmptyObjLike;

PROCEDURE Clone( x : T ) : T =
VAR
  temp : T;
BEGIN
  temp := EmptyObjLike( x );
  temp.Copy( x );
  RETURN temp;
END Clone;
```

Next to the functions mentioned above, the end user can initialize an object by using methods `SetToZero` and `SetToOne`. The former initializes an object and sets its value to zero, whereas the latter one initializes an object and sets its value to one. These methods are declared at the top of the object hierarchy, because they are heavily used during the programming of any kind.

3.1. An example: Ackerman's Function

The usage of the implemented methods and procedures will be illustrated on a concrete example of Ackerman's function. A general function is implemented whose work does not depend on the type of arguments passed to it. It returns a result of the same type as the type of one of its arguments.

Ackerman's function is defined in the following way:

$$
\begin{aligned}
A(n,x,y) &= x+1, && n=0;\\
A(n,x,y) &= x, && n=1, y=0;\\
A(n,x,y) &= 0, && n=2, y=0;\\
A(n,x,y) &= 1, && n=3, y=0;\\
A(n,x,y) &= 2, && n>3, y=0;\\
A(n,x,y) &= A(n-1, A(n,x,y-1), x), && \text{otherwise}
\end{aligned}
$$

The function arguments x and y have the type Number, and n has the type PlainInt, because that the argument is an integer by its definition. The arguments x and y can be the integers, rational, real or complex numbers. Of course, if the complex numbers are passed, and if their imaginary part is a non-zero value, the function will never finish.

```
PROCEDURE Ack(n:PlainInt; x,y:Number): Number =
VAR
  k1, k2 := NEW( PlainInt );
  t := NEW( Number );
BEGIN
  t := Clone( x );
  IF n.Zero() THEN
    t.Inc();
    RETURN t;
  ELSIF y.Zero() THEN
    k1.Init( 2 );
    k2.Init( 3 );
    IF n.Eq( k1 ) THEN
      t.SetToZero();
    ELSIF n.Eq( k2 ) THEN
      t.SetToOne();
    ELSIF n.GEq( k2 ) THEN
      t.SetToOne();
      t.Inc();
    END;
    RETURN t;
  ELSE
    k1.Init( 1 );
    t := Clone( y );
    t.Dec();
```

```
      RETURN Ack( Sub(n,k1), Ack(n,x,t), x);
    END;
  END Ack;
```

At the beginning of the procedure, the object **x** is cloned because the concrete type of the object **x** is not known. When it is necessary to initialize the value of the temporary variable **t**, it is done by using one of the methods **SetToOne** or **SetToZero**. Of course, when the end user needs to initialize its value to two, or some other value different from one or zero, he/she can combine the methods **SetToOne** and **Inc**.

This example clarifies the advantages of this implementation. The function does not have to be altered for different argument types, it will simply return the result that has the same type as the argument **x**.

4. Conclusion

This paper describes an object-oriented number class hierarchy and implementation of the algorithms for multiple precision arithmetic. The point of this research is not algorithmic optimum, but the application of the object-oriented methodology.

A great advantage of this implementation lies in the possibility to write general procedures which are capable of working with any number type. If all procedure arguments are declared to have the type **Number**, a procedure written in that way will not have to be rewritten, depending of its arguments. Any number is **Number**, so the procedure will return a result no matter what the number type is passed to it.

References

[1] Harbison, S. P., Modula-3, Prentice Hall, Engelwood Cliffs, New Jersey, 1992.

[2] Mössenbock, H., Object-Oriented Programming in Oberon-2, Springer-Verlag, Berlin, 1992.

[3] Budimac, Z., Ivanović, M., Paunić, Dj., Programming Language Modula-2, University of Novi Sad, 1998 (in Serbian).

[4] Popović, A., An Implementation of Number Class Hierarchy in Programming Language Modula-3, B. Sc. Thesis, University of Novi Sad, 1997 (in Serbian).

[5] Popović, A., Multiple precision real numbers in object-oriented environment, Student's Paper, University of Novi Sad, 1997 (in Serbian).