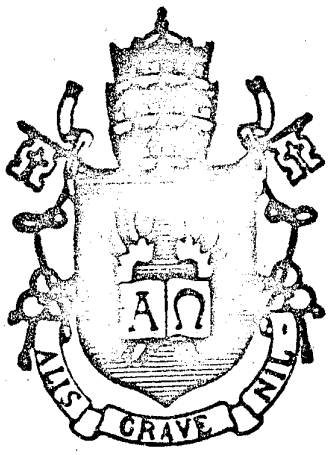

PUC

Series: Monografias em Ciência da Computação, No. 3/91

THE O=M PROGRAMMING LANGUAGE

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMATICA

Series: Monografias em Ciência da Computação, No. 3/91
Editor: Carlos J. P. Lucena March, 1991

THE O=M PROGRAMMING LANGUAGE *

Roberto Ierusalimschy **

* This work has been sponsored by the Secretaria de Ciência e Tecnologia of Presidência da República Federativa do Brasil.

** On leave at the University of Waterloo, Canada.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

Abstract

O=M is an object oriented programming language designed to achieve compatibility between Object Oriented Programming and Abstract Data Types. The main characteristic of the language is a complete identification between the concepts of Object and Module. This feature allows the language to have entities which, like objects, are dynamically created and manipulated, have inheritance and late-binding, and at the same time, like modules, can export types, have separate specifications and implementations, and support nesting facilities. Specifically, the dynamic manipulation of objects which export types gives the language a high degree of polymorphism.

Keywords: Object Oriented Programming, Programming Languages, Type Systems.

Sumário

O=M é uma linguagem de programação especialmente projetada para compatibilizar Programação Orientada a Objetos e Tipos Abstratos de Dados. A principal característica da linguagem é uma completa identificação entre os conceitos de Módulo e Objeto. Esta característica permite que a linguagem tenha entidades que, da mesma forma que objetos, podem ser criadas e manipuladas dinamicamente, têm herança e "late-binding"; ao mesmo tempo, estas entidades, assim como módulos, podem exportar tipos, têm especificações e implementações separadas, e oferecem suporte para aninhamento. Especificamente, a manipulação dinâmica de objetos que exportam tipos dá à linguagem um alto grau de polimorfismo.

Palavras-Chave: Programação Orientada a Objetos, Linguagens de Programação, Sistemas de Tipos.

1 Introduction

Since the beginning of the 80's Object-Oriented Programming (OOP) has received more and more attention as a new programming paradigm. Certainly, a main area (or The main area) in object orientation has been programming languages. From its origins with Simula-67, to its notoriety through Smalltalk-80, when characteristics like inheritance and data abstraction were combined, OOP has been guided by developments in object-oriented programming languages (OOPL).

In spite of all evolution, most OOPL have not incorporated many of the programming language technologies established in more traditional languages, such as those related to rigorous software development. More specifically, the famous equation *Objects = Abstract Data Types + Inheritance* is not indeed correct: besides the lack of a formal approach, which has always guided ADT facilities, OOPL do not present many features originated from ADT concepts, like module facilities, separate implementation and specification parts and generic (parametric) data types.

To overcome the limitations of OOPL for ADT support, we propose a new programming language, called O=M. From a programming perspective, O=M has been designed having as a basis a formal object-oriented development method, which incorporates nesting and inheritance in a VDM framework. From a linguistic point of view, O=M puts together features from OOPL as well as from ADT based languages, in an attempt to identify abstract data types with objects; as a design outcome we have the distinguishing mark of the language, which is a complete identification between the concepts of Object and Module.

Among other features, O=M has complete independence between specification and implementation hierarchies, a formal basis for late binding and multiple specification inheritance, nesting of specifications and implementations, a high degree of polymorphism, facilities for multiple implementations for each type, and a more strict visibility control over names and objects.

This paper presents the programming language aspects of O=M; the formal method behind it is presented in [9]. In the next section we take a closer look at the problems with OOPL for ADT support. An overview of O=M is presented in section 3. Section 4 discusses the relationship between nesting and inheritance. Section 5 shows the facilities of the language for visibility control. Finally, in section 6 we present some conclusions.

2 OOPL × ADT

In this section we give close attention to some incompatibilities between Object Oriented Programming Languages and Abstract Data Types. The first step here is a clear characterization of OOPL. We characterize OOPL by a set of language mechanisms which 1) are found in all languages undoubtedly recognized as Object Oriented, 2) usually are not found in other programming languages, and 3) are enough to explain most of the qualities assigned to this kind of language. These mechanisms are data abstraction with persistence, inheritance, late binding, instance visibility, and reference semantics. We will explain better these concepts on the course of the text, while we relate them with ADT.

2.1 Data Abstraction

We begin with Data Abstraction, which is a key point in OOPL. Despite the good support found on these languages, there are some drawbacks. One is caused by the reference semantics: as all variables actually keep the address of an object, the abstraction mechanism protects only these addresses, and not the real values of the variables (more on this topic in the discussion about reference semantics). Another drawback is the lack of a clear separation between specifications and implementations, like we have in Ada [1] and Modula-2 [19].[1] The distinction between specifications and implementations is a key point in data abstraction, and a specific language support for it seems very important for ADT.

2.2 Inheritance

As a consequence of the lack of separation between specifications and implementations, OOPL normally present another pitfall, the lack of distinction between both hierarchies.[2]. Implementation inheritance, as shown in Smalltalk [7], is the ability of an implementation (or class) to (re)use code and data structures written for other implementations as its own, with complete freedom to modify these features for any new purpose. On the other hand, specification inheritance, as shown in Emerald [2], is a mechanism to classify types and to increment the language polymorphism. Since Simula-67 these two concepts have been mixed, although there is no reason to suppose

[1] The OOPL Duo-Talk [14] has this feature.

[2] Again, Duo-Talk is an exception; see also [3]

that similar behaviors must be implemented by similar structures and vice versa. For the user of a class, the implementation hierarchy is useless and must be hidden, in the same way as the implementation itself, while the specification hierarchy has to be shown, as the user needs to know the type compatibility rules over this class.

Using the inheritance classification proposed by Wegner & Zdonik [17], we can say that specification inheritance must be *signature compatible* (or even behavior compatible, if the compiler could check it), while implementation inheritance is free to be *cancel compatible*. With distinct hierarchies, we can reuse code the way we need, changing or deleting features, and at the same time have static type checking and a clear concept of a type as a specification.

2.3 Late Binding

Binding is the association between a name and its semantics, or its implementation. Late binding is the facility to bind operation names with operation bodies only at call time, instead of compile or link time, as is usually done. There is a close relationship between binding and abstraction, because it is at binding time that abstractions must be instantiated. When binding takes place at compile time, as in Pascal, the compiler must known all data implementations. With binding at link time, as in Modula-2, we have much more freedom, but still implementations are fixed during run time. Only with late binding a language achieves complete representation independence, and a program using a data type needs to know nothing about its representation.

On the other hand, late binding presents some difficulties for reasoning about programs. Without further restrictions, an operation can have not only many implementations, but many specifications as well. A partial solution to this is presented in typed languages. In these languages, each operation specification is attached to a type, so only objects with this type (or some subtype of it) give implementations for the operation. The idea is that all these implementations must satisfy at least the given specification, so it does not matter which binding is done during run time. A sound type system can restrict the ways an operation can be (re)defined on subtypes, in order to keep its syntactic specification.[3] Anyway, only the types are

[3]The language Eiffel suggests a stronger verification, with the use of pre and post conditions for operations. Unfortunately, it has not incorporated this verification facility into the compiler. (see [16])

verified, and not the whole specification.

2.4 Instance Visibility

By Instance Visibility we mean the property that objects can access only their own internal variables, i.e., the abstraction boundary surrounds each object individually. On languages based on ADT, the abstraction boundary is set surrounding the whole type, so the type operations can access internal variables of any instance of the type. The benefits of Instance Visibility include facilities for distributed systems, when the parameters of an operation can be on different machines, and the possibility of multiple implementations of a type, as one can have a different operation implementation for each type implementation. As subtyping is a special case of this situation, Instance Visibility is an important mechanism to support inheritance.

On the other hand, Instance Visibility has some drawbacks. One is efficiency; operations over two or more parameters of the same type can be more efficiently implemented if they can access details of all parameters. As a consequence, sometimes a class exports internal details in order to achieve efficiency, sacrificing abstraction. A good example is the class Rational, in Smalltalk. As the addition operation does not have direct access to the other operand, the class has to export functions to access the numerator and the denominator of a rational number, two meaningless functions from an abstract point of view.

2.5 Reference Semantics

In OOL, all variables are actually pointers. This property, which has consequences both in language semantics and implementation, is what we call Reference Semantics. Some ADT languages, as CLU, also have this property, although it brings some problems for data abstraction. The advantages of this kind of semantics are the separation between the lifetime of variables and objects, and good support for a Database style of programming. From the implementational point of view, it simplifies memory allocation for variables, as the actual objects can have variable size due to inheritance and polymorphism. On the negative side, we have minor problems with efficiency, as all objects must be dynamically allocated. But the real problems are with data abstraction and modularity.

For instance, suppose an ADT Stack implemented by an Array. Internally, the Stack has a protected variable pointing to the array, but the array

itself "lives" outside the stack. If any other object gains access to the array, the abstraction is broken. This problem is worsened by what we call Capture. In procedural languages, when a procedure is called with some arguments, these arguments can be used by the procedure in many ways, but usually when the procedure returns the sharing vanishes. In OOL, there is no way to distinguish between parameters sent to the method or to the receiver object. So, any parameter of a method call can be stored (captured) inside the receiver object for later use. In our example, if the Stack calls a procedure passing the array as a parameter, the array can be captured and later accessed.

3 The O=M Programming Language

This section provides a general overview of the programming language O=M. The languages with major influence over O=M have been Modula-2 [19], CLU [13] and Eiffel [15]. From Modula-2 we have borrowed the lexis, the syntactic style and most of the module concepts, like nesting, visibility rules and a clear separation between specifications and implementations. From CLU we have incorporated most of the semantics, like creation of objects, parameter passing and assignment. Eiffel has given the object-oriented flavor, with multiple inheritance, late-binding, renaming and redefinitions.

A program in O=M is syntactically similar to one in Modula-2. It is composed by many syntactic modules, some with specifications and others with implementations for these specifications. On the other hand, the semantic behavior is close to Eiffel and CLU. In the same way as in CLU, every value is an object. Objects are the basic and unique active entities in the language, which can be stored in variables, passed as parameters and so on. The variables always store references. There are explicit commands (called constructors) to create new objects, while the deletions are automatically done by a garbage collector.

The main distinguishing mark of O=M is a complete identification between modules (in the sense of Modula-2) and objects (in the sense of Eiffel). These entities (called objects from now on), like modules, are described by a specification part and a separate implementation part, can export new types (which in O=M are the same as specifications), and can be nested. At the same time, like conventional objects, these entities can be dynamically created, passed as parameters, and stored in variables. Specifically, as objects can export types and be dynamically manipulated, the language has a high

degree of polymorphism, as we will discuss in the next section.

3.1 Specifications

Objects are described in the language by their types and classes, or specifications and implementations. A class is a kind of static generic recipe used to create new objects with a desired behavior. A type, on the other hand, is an external description of this behavior. This description declares all the features that objects of this type must have. These features include functions, procedures and (nested) specifications.

A specification can be declared as a heir from other previously declared specifications. This statement makes all features declared (or inherited) by the parents to be available in the specification, and stands the new type as a subtype of its ancestors. It is important to note that, unlike some other languages (e.g. Quest [4] and Emerald [2]), a type is a subtype of another one only if it is so declared. As what we call a specification is only a syntactic specification, the language can not constrain compatible specifications to have compatible implementations, and so the programmer must assume the responsibility for every subtype relationship. For the same reason, name collisions between inherited features are also usually considered errors, even when the definitions are compatible, in order to avoid undesired coincidences. Only in the event of repeated inheritance, when the same type is inherited more than once through different paths, the features inherited repetitively are unified.

When a type inherits from another one, it can *rename* and *redefine* the inherited features. The rename mechanism is a way to change the name of an inherited feature, with the old name loosing its meaning. Its most common use is to avoid clashes between features with the same name inherited from different parents; it can also be used to give to the feature a name more appropriate to the new environment. A redefine declaration list means that the specification will present new definitions for the listed features. It also allows controlled unifications: if two or more homonym features, inherited from different ancestors, are redefined, only one new definition is needed. Obviously, this new definition must be compatible with all the old ones. The next example shows the use of these facilities.

Example 1:

Specification A;

```
  Procedure Print;
  Function Replicate ():A;
End A.

Specification B;
  Procedure Print;
  Function Copy ():B;
End B.

Specification AB;
  Subtype of A
    rename Print as PrintA, Replicate as Copy;
    redefine Copy;
  Subtype of B
    redefine Copy;
  Function Copy ():AB;
End AB.
```

Here, the functions `Replicate` (renamed as `Copy`), from `A`, and `Copy`, from `B`, are unified in a new definition (called `Copy`) inside `AB`. The new definition matches the old ones, as it returns an `AB` object, which is a subtype of both `A` and `B`.[4]

Objects, like modules, can export types. This gives the programmer an option to program in a way more close to ADT languages, as shown in the next example.

Example 2: This example shows a possible specification for integer numbers, in an ADT fashion.

```
Specification IntegerSpec;
  Specification Integer; -- exported type
  End Integer;
  Function Add (a,b:Integer):Integer;
  Function Sub (a,b:Integer):Integer;
  ...
End IntegerSpec.
```

[4]As usual (and correct) in type systems with subtyping, a function type $A \rightarrow B$ can redefine (i.e., is a subtype of) a function type $C \rightarrow D$ iff C is a subtype of A, and B is a subtype of D.

This module exports a type and functions to manipulate objects of this type. As the type and the functions are inside the same module, the functions can see details of objects of type Integer that are not visible outside an IntegerSpec object.

Please note that the language also allows to build this specification in a more object-oriented way, putting the operations inside the integer objects. In fact, with some duplication we can have both (OO and ADT) styles, as in the next fragment.

```
Specification IntegerSpec;
  Specification Integer;
    Function Add (a:Integer):Integer;
    Function Sub (a:Integer):Integer;
    ...
  End Integer;
  Function Add (a,b:Integer):Integer;
  Function Sub (a,b:Integer):Integer;
  ...
End IntegerSpec.
```

Having an implementation Int for IntegerSpec, together with variables a and b of type Int.Integer, we can perform the addition either as

```
Int.Add(a,b)
```

or as

```
a.Add(b)
```

In those cases, the two Add operations do not have to have the same meaning.

It is important to note that the type is exported by the object, and not by the specification. A program can have many objects of the type IntegerSpec, maybe tied to different implementations, and each of these objects defines a different Integer type. A conversion function between these types could be declared, inside the IntegerSpec type, as follows:

```
Function Convert (I:IntegerSpec ; i:I.Integer):Integer;
```

This function receives as arguments an integer implementation and a value in this implementation, and returns a value with the receiver implementation. O=M uses the dot notation to access exported features, so `I.Integer` denotes the type `Integer` exported by the object bound to the parameter `I`.

Specifications can have parameters, as illustrated in the next example.

Example 3:

```
Specification RationalSpec (I:IntegerSpec);
  Specification Rational;
    Function Round:I.Integer;
  End Rational;
  Function Add (r,q:Rational):Rational;
  Function Create (num,den:I.Integer):Rational;
  ...
End RationalSpec.
```

This specification allows us to indicate which integer implementation to use to interface with rational numbers. Parameterized specifications also give the possibility of creating generic types. This facility will be explained in the next section.

3.2 The Concept of Type in O=M

The concept of type commonly used in programming languages is something like "Types are sets of data values" [10], or a type is a *function abstraction*, or a collection of functions, which gives a meaning to a, otherwise untyped, value [6]. Under these assumptions, to type check a program is to verify that no operation is applied to an invalid value (i.e, from a wrong type), or that one never uses the wrong abstraction function over a value. In both cases, the result is the assurance that the semantics of a correctly typed program is independent of representation , i.e., its semantics does not depend on the representation chosen for each data type.

These concepts are not very useful in object-oriented languages. As the operations are semantically attached to the objects (via late binding), there is no way to apply an operation over an object of the wrong type[5]. With

[5]In run-time checked languages, like Smalltalk, one can send an invalid message to an object. But this is caught by the language, that generates a *message not understood* error. So, the program stops before the wrong operation is performed.

these meanings, all object-oriented languages are strongly (not necessarily statically!) typed. Another complicating factor is subtyping, which would imply that the set of values of different types were not necessarily disjoint, or that a given value could have multiple abstraction functions. Nevertheless, few object-oriented programming languages give a precise meaning for type. In the OOL jargon, what is considered a (statically) typed language is one that the compiler can assure the absence of *message not understood* errors. We think we could get more from types.

A direct consequence of the identity Object=Module in O=M is the identification between the concepts of type and specification. So, any object of a given type satisfies the associated specification. With this interpretation, type declarations are seen as (partial) program specifications, and type checking as (partial) verification. Subtyping presents no problem: a subtype is just a stronger specification. So, any object of a given type is also of any supertype of it. Again, we note that only the specification hierarchy needs this kind of behavior compatibility; the implementation hierarchy has a freer structure.

O=M has been developed in order to support a formal development method ([9]), which is an extension of the VDM based method proposed by Jones in [11]. The type rules of the language have been all derived from verification rules, and not from random examples or other ad hoc motivations. Besides the formal ground, there has been an extra rule: everything that can not be verified by the compiler must be explicitly stated by the programmer. This explains why the subtype relationship must be declared (the compiler does not know the complete semantics of each type), and name clashes between inherited features are treated as errors unless otherwise stated (via redefinitions).

3.3 Implementations

For each specification, there can be zero or more implementations. These implementations are referred to in commands for creation of new objects, so one can choose dynamically which implementation to use for each object. Implementations define the internal structure of objects and, like Modula-2's implementation modules, are composed of a set of declarations. These declarations include variables (called *instance variables* in the OO jargon), constants, functions, procedures, specifications, (nested) implementations, and constructors.

When we declare a variable, we state only its type; the variable can

handle any implementation of this type. Only when we create a new object must we give an implementation. In order to create an object, we apply a constructor over an implementation. A constructor is a kind of function that creates and initializes objects of the class, returning the new object. The semantics of these constructors is quite similar to the `Create` procedure in Eiffel, but instead of having a fixed (and only one) name we may use any (and as many as desired) identifiers for this purpose, just declaring them with the **Constructor** reserved word. It is important to note that a constructor is not an object feature, but a kind of implementation feature. Different implementations of the same type can have different constructors.

As a specification can have no implementations, the language does not need a mechanism like Simula's virtual (or Eiffel's deferred) features. A virtual class in O=M is just a specification with no implementations. Moreover, an implementation may have no constructors, so we can build partial implementations, only to be inherited; the absence of constructors assures that they can have no instances.

An implementation can be declared as a heir of other previously written implementations, in order to reuse their declarations. The hierarchy created this way has no relationship with the specification hierarchy. As we have already said, the main purpose of the specification hierarchy is to reuse specifications and to create subtype compatibility from a behavioral perspective, while the main purpose of the implementation hierarchy is to reuse code and data structures. The only compromise between them is that each final implementation must have at least all features of its specification.

When an implementation inherits declarations from others, it can also rename these declarations. There is no need for a redefine facility, as a subimplementation has no compatibility compromise with its ancestors. Instead, an implementation has the freedom to list which declarations it wants from each parent. Usually, the binding between names inside the same object is done at compile time. When a feature is inherited, it keeps all its bindings, despite renames. Any attribute linked to the inherited one is also inherited, nameless if needed, in order to maintain consistency. If the programmer wants late-binding, s/he may use the `Self` constant, which has the usual meaning. In other words, if there is a direct reference to a name, we get early-binding, while if there is an access to an attribute of `Self`, we get late-binding. It is worth to note that only exported names can be referred by `Self`, as its type is given by the specification. As an implementation does not need to have the same internal features of its parents, this rule avoids invalid references.

An implementation can declare nested implementations. These internal implementations are used to reify exported types and to describe classes with restricted use. Like in Modula-2, nested classes can access internal features of their external object, and can export features visible only inside this external object. The following example shows these facilities.

Example 4: This example is an implementation (Rat) for the specification RationalSpec shown in example 3.

```
Import RationalSpec , IntegerSpec;
Implementation Rat:RationalSpec (I:IntegerSpec);

  Specification Rational;
    Function Round:I.Integer;
    Function num:I.Integer;
    Function den:I.Integer;
  End Rational;

  Implementation R:Rational;
    Var n,d:I.Integer;
    Function num:I.Integer;
      Begin Result := n End;
    Function den:I.Integer;
      Begin Result := d End;
    Function Round:I.Integer;
      Begin ... End;
    Constructor new (num,den:I.Integer);
      Begin n := num ; d := den End;
  End R;

  Implementation RInt:Rational;
    Subimplementation of R with num , n as value;
    Function den:I.Integer;
      Begin Result := 1 End;
    Function Round:I.Integer;
      Begin Result := value End;
    Constructor new (num:I.Integer);
      Begin value := num End;
  End RInt;
```

```
  Function lcm (i,j:I.Integer):I.Integer;
    Begin ... End;

  Function Add (r,q:Rational):Rational;
    Declare d := lcm(r.den,q.den);
            n := r.num*(d/r.den)+q.num*(d/q.den);
    Begin Result := Create(n,d) End;

  Function Create (num,den:I.Integer):Rational;
  Begin
    If den = 1 then Result := RInt.new(num)
    else Result := R.new(num,den)
  End;
  ...
End Rat.
```

This example illustrates many points. The functions num and den, exported by the type Rational, are not visible outside the implementation Rat, because the type Rational in the specification does not export those features (example 3). As any other specification, internal ones also can have multiple implementations; in the example, we have two different implementations for the type Rational, an usual one (R) and another to deal with the special case of integers (RInt). Implementations of parameterized specifications must have the same parameters; these parameters act like constants inside the implementation. That implementation uses the integer implementation, given as parameter, not only in its interface but also in its internal structure (see the declaration of the variables n and d). The visibility rules allows the implementations R and RInt to access the parameter I.

4 Inheritance × Nesting

Although sometimes they are considered redundant or antagonistic (p.e. [16], p. 323), the mechanisms of inheritance and nesting can be well integrated, and their union gives rise to some powerful facilities of O=M. In this section we will see how specification inheritance and nesting are used to achieve parametric polymorphism, and how implementation inheritance and nesting allow the definition of private classes without sacrificing reusability.

4.1 Parametric Polymorphism

Polymorphism in a programming language is the facility of an entity (object, function, etc) to have multiple types. An always present form of polymorphism in OOL is subtyping: an object of a given type T also belongs to every supertype of T. Another form of polymorphism, presented in non OOL such as Ada and CLU, enables one to define generic entities with actual types depending on parameters; this form is called parametric polymorphism. (Cardelli & Wegner [5] call these two forms as Vertical and Horizontal polymorphism).

The first thing to note is the difference between both kinds of polymorphism. For some situations they are equivalent, as shown in the following routine declarations:

```
Function Area (x:Figure):Integer;
Function Area (t:Figure_Type ; x:t):Integer;
```

In the former declaration, x can be of any subtype of Figure; in the latter, t can be any subtype of Figure, and x has this type. But this equivalence is not valid when the routine has a collection of parameters of the polymorphic type, as below:

```
Function F (x:array of T):Boolean;
Function F (t:T_Type ; x:array of t):Boolean;
```

Now, the former means that each array element can be of any subtype of T, so x[1] can be of one subtype and x[2] of another one, while in the latter all elements must have the same given subtype (t). No form is better than the other, as each one has its applications. For instance, if we have a function Max over the type Magnitude, with subtypes Integer and Char, the former would admit (erroneous) comparisons between integers and characters. On the other hand, if the function is Empty_Intersection, over the type Collection, there is no reason to avoid comparisons between sets and bags (or other kinds of collections).

The same difference happens with generic data types. So, a declaration like Stack(Magnitude) means a stack where each element can have a different subtype of Magnitude, while something like Stack(t:Magnitude) means that all elements have the same subtype.

Usually, parametric polymorphism presents some semantic difficulties in programming languages, due to the need to assign a formal meaning to

type values (for instance, the t parameter of the examples above), and the difficulties for static type checking. Some languages, like Ada [1] and CLU [13], avoid these problems dealing with parametric polymorphism as a static facility, constraining actual type parameters to be manifest constants (type names) and forbidding any kind of computation with these values other than parameter passing. In this context, parametric polymorphism is equivalent to a macro facility, and has no effect in the dynamic semantics of a language.

Other languages offer more elaborate solutions. In Russel [6], for instance, a data type is identified with the set of operations which gives meaning to values of this type. As operations (functions and procedures) are values (as in many other languages), so are data types. To achieve static type checking, the language has some constraints in order to have the *substitution property*[6]; this way, the compiler decides type equality via equality of the expression texts describing the types. Another solution is presented by Quest [4], which uses a two level type system: usual values, such as integers, stacks, and functions over these values have types with level 1, while the types (or *kinds*) of level 1 values such as `Integer` and `Stack→Integer`, belong to level 2. Static type checking is achieved avoiding loops and recursion in functions over level 1 values, so the compiler can perform all computations on this level.

In O=M, parametric polymorphism is a consequence of the mechanisms for specification nesting and inheritance. This solution has some similarities with the one in Russel. An object is a dynamic entity composed by a set of operations and a set of values. Some (or all) of these operations can be used to characterize a type exported by the object. So in O=M, "type" values and common values are unified under the Object concept.[7] An example is in order now, to show how to use this facility.

Example 5: This example shows a specification for types with an order relation. The specification Any is used to give more structure to the hierarchy.

```
Specification Any;
  Specification T;
  End T;
End Any.
```

[6] Which means that function calls with identical arguments return identical results.
[7] This characterization is used in the denotational description of the language.

```
Specification MagnitudeSpec;
  Subtype of Any rename T as Magnitude;
  Function LessThan (a,b:Magnitude):Boolean;
  Function GreaterThan (a,b:Magnitude):Boolean;
  ...
End MagnitudeSpec.
```

Objects of type MagnitudeSpec are implementations of magnitude types. Objects of type M.Magnitude, for some M of type MagnitudeSpec, are magnitude values from the implementation M; for each different M we have a different type. Now we can use MagnitudeSpec as a "type" parameter to declare the following polymorphic function:

```
Function Max (M:MagnitudeSpec ;
        a,b:M.Magnitude):M.Magnitude;
Begin
  If M.LessThan(a,b) then Result := a
  else Result := b
End;
```

Now, if we declare the two types below, the function Max can be used with both.

```
Specification IntegerSpec;
  Subtype of MagnitudeSpec
    rename Magnitude as Integer;
  Function Add (a,b:Integer):Integer;
  ...
End IntegerSpec.

Specification CharSpec;
  Subtype of Magnitude
    rename Magnitude as Char;
  Function IsDigit (a:Char):Boolean;
  ...
End CharSpec.
```

It is worth noting that there is no inheritance relationship between the types Magnitude, Integer and Char; after all, they are exported by different objects. The relationship only holds for the types CharSpec, IntegerSpec

and MagnitudeSpec. From a Russel perspective, the Spec objects can be identified with type values.

The same technique can be used for specifications. Again, we use an example to illustrate the point.

Example 6: Here we declare a generic sorted collection, that is, a collection with an order relationship among its elements. The function LeastElement returns the smallest element inside the collection.

```
Specification SortedCollection (m:MagnitudeSpec);
  Procedure Insert (e:in m.Magnitude);
  Function LeastElement:m.Magnitude;
  ...
End SortedCollection.
```

Some remarks are in order here: the type of the parameter m imposes the restriction that elements in a SortedCollection must have an order, or from an implementational point of view, m has to have a function LessThan to compare these elements, since all subtypes of MagnitudeSpec must have such a function. This facility is equivalent to the **Where** clause in CLU or **With** in Ada. Another remark is that the real parameter for this specification is an object, not another specification. So, it is meaningless to declare something like

```
X:SortedCollection (CharSpec); -- Wrong!
```

but, given an object C of type CharSpec, which is a concrete implementation for characters, we can declare

```
X:SortedCollection (C);
```

In order to make the type system checkable, we rely on constants. To know if types exported by objects are equal, one must know if the exporters and parameters are the same. O=M only recognizes the equality if these objects are denoted by the same constant (through parameter passing). This frees the language from the need of the substitution property. So, if one needs to declare objects with a type (actually the object which exports the type) that is given by a function, one has to assign the function result to a new constant. As function and procedure **In** parameters are constants, they can be used directly to declare types.

4.2 Nesting × Public Implementations

A common critique against nesting is that nesting is prejudicial for code reuse. This is due to two factors. First, the visibility rules do not allow a nested feature to be used from the outside world. Second, a nested routine usually access details of its environment, and this tends to make the feature too specific to be useful elsewhere. On the other hand, nesting is a practical mechanism to avoid the proliferation of classes at the global (and unique, without nesting) level, and to control visibility.

In this situation, inheritance offers an interesting solution which allows the use of nesting without its drawbacks. With the implementation inheritance mechanism, a programmer is able to declare a generic feature on the global level, while at the local level s/he fits a subimplementation of it for the specific need. If later the nested feature shows itself useful in other places, the programmer can declare it as a new global class; in order not to change the structure of its old scope, the programmer can keep the nested class, which now only needs to import the global one.

Before finishing this section it is important to note some restrictions when mixing inheritance and nesting. According to the visibility rules, a nested class can access features of its external object (i.e., the one which surrounds it). If such class is inherited by another one not surrounded by the same external object, the subclass wold be able to access features outside its visibility limits. In order to keep consistency, O=M poses the restriction that a superclass must either have the same external object of its subclass (i.e., be defined inside the same scope) or be a global class. The second possibility is due to the fact that global classes can not access any external feature.

For similar reasons, this restriction also applies to nested specifications. So, a supertype must either have the same external object of its subtype or be a global type.

5 Visibility Control

Before the discussion about visibility control, we would like to talk a little more about the concept of capture, which is a concept introduced by Algol, via the block mechanism. Capture is the act of a routine to assign an input parameter value to a non-local variable, so that this value can still be accessed after the return of the routine. This facility creates a parting between the scope (and life time) of a variable and the scope of its contents.

Capture is also present in OOPL, via the nesting of methods inside objects.

OOPL do not possess most of the semantic problems caused by capture in Algol-like languages, as that separation of scopes is already present in object orientation. On the other hand, OOPL have more pragmatic problems. Although capture has been introduced by Algol-like languages, the programming style usually adopted in these languages does not stimulate its use, and in fact many languages impose restrictions on it (p.e. Euclid [12] and Algol-68 [18]). Quite to the opposite, in OOPL capture is a must. The only way to link two objects is sending one as a parameter for a method of the other. The problem is that any object used as a parameter for a method can be captured by the receiving object, making a permanent link between them. There is no way to restrict statically the links between objects: all objects are global.

In order to increase the granularity of control over links between objects in O=M, we make use of the External Object concept. As already stated, the external object (EO) of a given object is the one that exports its type; from an implementation point of view, the EO is the surrounding module of a given object. External Objects make a partition on the object space. Each object defines a new universe where all its internal objects live, in a way similar to Euclid's *collections*. Objects of classes declared at the most external level of a declaration have no EO, but we assume a pseudo-object named Global which surrounds them.

Besides that partition, EO presents another facility that is used to restrict capture. In order to capture an object we must assign it to a variable of the right type, that is, a supertype of its own type. As we have already seen, a supertype of a given type must either have the same external object or be Global. To limit capture, O=M poses the restriction that variables can not have Global types. The consequence of these rules is that in order to capture a given object we have to have already captured its EO; otherwise there is no way to declare a captor variable with the proper type. (It is important to note that constants and Input parameters can have Global types.) The above facility allows the differentiation between parameters sent to methods and to objects. Parameters sent to methods can have global types, so the receive object has no way to capture it. An example will help to clarify these ideas.

Example 7: Suppose we have the following definitions:

```
Specification A;
End A.
```

```
Specification B;
  Specification B1;
    Subtype of A;
  End B1;
  Specification B2;
    Subtype of B1;
  End B2;
End B.

Specification C;
  Procedure P1 (x:in A);
  Procedure P2 (x:in B ; xx:in x.B1);
End C.
```

In this example, it is legal to call P1 with any object of type b.B1, as B1 is a subtype of A (for any b of type B). But, as A is a global type, it is impossible to declare any variable with this type, and capture is impossible. When this routine returns, we are sure the parameter is not being shared. Procedure P2, when called, can declare variables that are type compatible with its second parameter (xx), using the type exported by the first one (x.B1). Anyway, capture is still impossible, as the scope of these variables is limited by the scope of the parameter x.

So far we have found a way to avoid capture. But many times one needs to do it, i.e., one needs to send a parameter to an object, and not only to the method. A trivial example is a Push operation on a Stack, where the pushed object remains on the stack after the end of the routine. As already stated, in order to capture a parameter an object has to have (permanent) access to its EO. This permanent access is achieved via the parameters of specification. When we use parameterization, both the specification and the implementations include the parameters. The specification can use them as external objects of types it uses, while the implementations can use them to declare variables with these types, thus allowing capture. This facility integrates smoothly with parametric types, as shown in the next example.

Example 8: This is the classical example of a parametric stack.

```
Import Any; -- from example 5
Specification Stack (a:Any);
```

```
  Procedure Push (e:in a.T);
  Procedure Pop (e:out a.T);
  Function Empty ():Boolean;
End Stack.

Implementation LinkedStack:Stack (a:Any);
  Var head:Node;
  Specification Node;
    Function value:a.T;
    Function next:Node;
  End Node;
  Implementation N:Node;
    Var v:a.T;
    Var n:Node;
    Function value:a.T;
      Begin value := v End;
    Function next:Node;
      Begin next := n End;
    Constructor new (value:a.T ; next:Node);
      Begin v := value ; n := next End;
  End N;
  Procedure Push (e:in a.T);
    Begin head := N.new(e,head) End;
  Procedure Pop (e:out a.T);
    Begin e := head.value() ; head := head.next() End;
  Constructor new;
    Begin head := Nil End;
End LinkedStack;
```

Now, if we have an implementation for integer numbers called Int (with type IntegerSpec, which is a subtype of Any), we can declare an integer Stack as:

```
Var IntStack:Stack(Int);
```

and create it as follows:

```
IntStack := LinkedStack(Int).new;
```

As Int is an implementation parameter, this stack object has permanent access to it. Using Int as external object, the stack is able to declare

variables with type `Int.Integer`, and then capture parameters with this type.

A distinctive feature of this mechanism is that all dependencies between objects are documented in the type declarations. In the example above, the declaration `IntStack:Stack(Int)` states clearly that object `IntStack` has permanent access to object `Int`. Another characteristic is that any object of a type that is not exported (like `Node`, in the example above) can never be captured, and so the only references to it are inside the EO. This fact does not preclude this objects to be used as parameters for outside methods, as their types can be heirs of global types.

The mechanism presented above implements what we call dynamic visibility control, as its purpose is to restrict access to dynamic entities (objects). O=M also presents mechanisms for static visibility control, i.e. to restrict access to static entities (identified by names). These facilities are similar to those presented by Modula-2, and we will give only a brief description of them.

An implementation can declare attributes not defined in the specification. These attributes are private, and can be accessed only inside the implementation. A nested object reifying an exported type can export more features than stated on the exported type; these features are visible only inside its external object. The example 4 (Rational numbers) has used this facility. Nested objects can access features of their surrounding objects; constants and specifications are automatically visible inside nested objects, while functions and procedures must be explicitly imported. As implementations can not be exported, the EO of an object is always visible at the place of its creation, and could be passed as an implementation parameter. So, the fact that objects can have direct access to external features can be viewed as a kind of syntactic sugar.

The last point to note is that all these facilities are not ad hoc. On the contrary, they are in conformity with the notion that an internal object is a constituent of its external object, and so an object state depends on the state of all its internal objects. This notion, only outlined here, is formalized in [9], which presents the design principles behind O=M.

6 Conclusions

We have presented a new programming language, called O=M, which intends to achieve a better compatibility between Object Oriented Program-

ming Languages and Abstract Data Types. The design of the language has been supported by a formal object oriented development method, which introduces the concepts of inheritance and nesting in a VDM framework.

The main novelty presented by O=M is the complete unification between objects and modules. Among the outcomes of this unification, we underline the following features:

- Independent specifications and implementations, favoring data abstraction and multiple implementations for a type.
- Independent specification and implementation hierarchies. The specification hierarchy enforces signature (behavior) compatibility between subtypes, in order to favor formal reasoning about programs, while the implementation hierarchy allows complete freedom for code reuse.
- Parametric polymorphism (and generic types); as dynamic objects can export types, the language is able to manipulate types during run-time. Despite this feature, O=M is statically (and strongly) typed.
- Nesting of implementations, which allows a better control of name visibility. Together with inheritance, this mechanism does not prejudice reusability.
- Dynamic visibility control. O=M gives ways to part the *object space* of a program. All links between different spaces are documented in the type declarations.

As the main purpose of O=M has been to show new language concepts, it still has not an implemented compiler. Instead of it, the language has a formal description by means of denotational semantics[8]. This description includes all type checking, and has been built in a way such that it simulates static checking: a program only has a valid denotation (different from Error) if it is free of static errors, like type errors, declaration errors, etc.

References

[1] ANSI *Ada Programming Language*, ANSI/MIL-STD 1815A – 1983.

[2] A. Black et alii. Distribution and Abstract Types in Emerald, *IEEE Transactions on Software Engineering SE-13*(1), 1987.

[8] This semantics is presented in [8].

[3] P. Canning et alii. Interfaces for Strongly-Typed Object-Oriented Programming, *Sigplan Notices 24*(10), 1989, pp. 457–67. (OOPSLA'89 Proccedings)

[4] L. Cardelli. *Typeful Programming*, notes of IFIP Advanced Seminar on Formal Description of Programming Concepts, Petropolis-Brazil, 1989.

[5] L. Cardelli & P. Wegner. On Understanding Types, Data Abstraction and Polymorphism, *ACM Computing Surveys 17*(4), 1985.

[6] J. Donahue & A. Demers. Data Types are Values, *ACM Transactions on Programming Languages and Systems 7*(3), 1985.

[7] A. Goldberg & D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[8] R. Ierusalimschy. *O=M: Uma Linguagem Orientada a Objetos para Desenvolvimento Rigoroso de Programas*, PhD thesis, Dep. Informática, PUC-Rio, 1990.

[9] R. Ierusalimschy. *A Method for Object-Oriented Specifications with VDM*, Monografias em Ciência da Computação 2/91, Dep. Informática, PUC-Rio, 1991.

[10] K. Jensen & N. Wirth. *PASCAL – User Manual and Report*, Springer-Verlag, 1976.

[11] C. B. Jones. *Systematic Software Development Using VDM*, Prentice-Hall International, 1986. (Series in Computer Science)

[12] B. Lampson et alii. Report on the Programming Language Euclid, *Sigplan Notices 12*(2), 1977.

[13] B. Liskov et alii. *CLU Reference Manual*, Springer-Verlag, 1981. (LNCS 114)

[14] C. Lunau. Separation of Hierarchies in Duo-Talk, *Journal of Object-Oriented Programming 2*(2), 1989, pp. 20–26.

[15] B. Meyer. Eiffel – A Language and Environment for Software Engineering, *The Journal of Systems and Software 8*(3), 1988, pp. 199–246.

[16] B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall International, 1988. (Series in Computer Science)

[17] P. Wegner & S. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, *ECOOP'88 Proceedings*, Springer-Verlag, 1988, pp. 55–77. (LNCS 322)

[18] van Wijngaarden et alii. *Report on the Algorithmic Language Algol-68*, Mathematisch Centrum, Amsterdan, 1969.

[19] N. Wirth. *Programming in Modula-2*, Springer-Verlag, Berlim, 1985. (third edition)