Abstract Data Types in Modular and Object-Oriented Programming

Ph. Narbel
LaBRI, Université Bordeaux I.
351, Cours de la Libération. 33405 Talence, France
e-mail: narbel@labri.fr

This paper attempts to pragmatically compare modular and object-oriented (OO) programming from the restricted point-of-view of user-defined abstract data types (ADT). In this respect, two main points are developed: first, the capability/obligation to nest or to collapse type components in ADT implementations is analyzed through a square of four relationships involving the types components making up an ADT. Second, different balances between type checking and type compatibility/flexibility are pointed up in the light of different implementations of the same ADTs using the two programming paradigms. The comparison will not yield a sharp distinction but will instead give precise hints about why, where and how modern modular programming tends to be more OO and vice versa.

1 Introduction

Modular programming is a way of organizing programs by building distinct name subspaces – *the modules* – with visibility control means. This very general definition leads to consider modern *object-oriented (OO) programming* as a particular kind of modular programming where name subspaces (*objects/classes*) are devoted to define and/or instantiating data types, with compatibility relationships through *subtyping*. OO programming has become a paradigm in itself. Thus, by a slight abuse of language, modular programming nowadays means one which is not object-oriented, i.e. modular programming through files, modules, packages, clusters, functors, etc...

Even if OO programming is generally thought of as an improvement over modular programming, the latter is still used, studied and developed. Some effective breakthroughs have been achieved and integrated in languages like the ones in the *ML* family : module typing and subtyping, coexisting multiple implementations, code-reuse schemes like inheritance through functors and module inclusions, redefinitions, and first-class modules are now offered by the module systems of these languages (see e.g. [MTHM97, HL94, Rus98, RRS00, LDG$^+$02]). Moreover, progress is still being made, e.g. *mixin modules* with recursive, deferred definitions and late-binding features (see e.g. [DC96, CHP99, AZ01, HL02]). In other words, much of what is often understood as specific of OO programming is already or is becoming features of modular programming. The situation is even more interwoven since in OO programming there is now a trend to include and develop features generally considered as modular ones, e.g. nested definitions (see e.g. [Szy92, Sma01, IP02b]) and various use of type parameters (see e.g [BOW98, TT99, IP02a]).

Taking these facts into account, what is really left out when comparing these two programming paradigms? The purpose of this paper is to attempt to answer this question by undertaking a pragmatic study between modular and typed OO programming based on the very basics, i.e. user-defined abstract data types (ADTs).

This paper is in the vein of most of the existing literature about this problem [Rey75, Coo90, Szy92, O'H98, Ler99, Pie02] but with the following distinctive features: first, we deliberately oppose the ADT concept in both programming paradigms. A distinction is indeed applied here between types and instances of types. According to the classical definition of ADTs, *objects* can be seen as a special kind of *instances* of ADTs where values go along with their operations. OO programming can be therefore considered as based on instantiating ADTs, in general by using *classes* (even if this is not a necessary condition). In practice, this explains for example why both modular and OO oriented languages often provide similar sets of definitions in their standard libraries.

Another characteristic of this report is that it does not intend to make an exhaustive comparison between the two programming paradigms. It strictly remains confined to ADTs and the immediate problems they arise, e.g. multiple coexisting implementations, relationships to their specifications, type compositions, type flexibility and type (un)safety. Moreover, it is strictly practical and based on examples given in existing multi-purpose languages. With this in mind, we had to fix an experiment protocol: First, we consider a given programming problem, and at least two classical implementations of it in the two different paradigms (what is classical must be e.g. checked in the mainstream literature and standard libraries of official language distributions). Second, specific properties of the two versions are analyzed to make their qualitative differences more precise. Third, in view of this analysis, we must try to emulate the first by the second and vice versa. Each satisfactory emulation indicates that the difference is essentially a modeling choice; each unsatisfactory one indicates either a true difference between the programming paradigms or a path for future developments. Of course, such a study does not have the same verification/refutation standard as a formalized method: what is really classical? what is really satisfactory? We shall however try to draw some conclusions in this loose context.

2 Abstract Data Types

An **abstract data type** (ADT) (see e.g. [Gut77, FG93, Pie02]) consists of three parts: a representation of the values of the type (called **the concrete representation**), an implementation of the operations associated with the concrete representation, and a way of applying an *information hiding* policy over these. The concrete representation and the definition of the associated operations make up the **implementation part** of the ADT, which is in general contained in a single syntactic encapsulating unit (sometimes called the *abstraction boundary* [Pie02]). Information hiding is what gives the abstractness of the type. It mainly relies on associated external sets of declarations forming **signatures** (or **interfaces**). Signature parts of ADTs are also necessary to provide information to clients and more generally to declare ADT during program design without commitment to the implementation.

This definition does not imply that there must be one single implementation for one signature/interface. One may indeed expect that several implementation parts share the same signature. This fact is of course true in OO programming, but we stress that this is also true in modern modular programming where it is becoming a very important feature.

As the first step of our experiment, we fix the programming problem we dwell upon: *implementing simple ADTs for ordered elements, like ordered integers and strings.* The two main representative languages will be *OCaml* (one of the powerful *ML* dialects) for modern modular programming, and *Java* for pure OO programming. The *OCaml* language will also allow us to express some intermediate cases since it implements an OO layer with some very recent features.

Now, the classical way of implementing these ADTs in *ML* following the mainstream literature (see e.g. [Pau97, App97, RRS00, LDG+02]) would be[1]:

```
module type ORDER =
    sig
      type element
      val cmp: element -> element -> int
    end

module Ordered_integers : ORDER =
  struct
    type element = int
    let cmp x y = x - y
  end

module Ordered_strings : ORDER =
  struct
    type element = string
    let cmp s1 s2 =   (* lexicographic order code *)
  end
```

(*Code 1*)

In *OCaml*, the two above ADTs implementations of the same signature can coexist in a single program. This coexistence will not be possible e.g. in *C* or *Ada* module layer. Nevertheless, to check the validity of the modeling view chosen for the above implementations, we show here that each single ADT would have a very similar structure in classical *C* (see e.g. [Han97] and its bibliography):

[1]For the sake of simplicity, most of our examples are not implemented with strict information hiding policies. In particular, constructors and accessors are skipped off. Adding them would not modify the conclusions we make here.

"ordered˙integers.h" file :

```
typedef struct element *element;
int cmp(element e1, element e2);
```

"ordered˙integers.c" file :

```
struct element { int val; };

int cmp(element e1, element e2){
  return (e1->val - e2->val);
}
```

(*Code 2*)

The classical way of implementing these ADTs in *Java* – again following the mainstream literature (see e.g. [Blo01, Sun02]) – would take a quite different form[2]. In fact, a *Java* programmer would probably make use of the abstract class (interface) `Comparable` which is included in the standard library of *Java* (the *Java Development Kit*). This interface gives a way of "*imposing a total ordering on the objects of each class that implements it*" [Sun02]. Most of the value classes in the JDK implement this interface. Thus, here is a possible code (recall that we skip the constructor codes):

"Comparable.java" file :

```
interface Comparable {
   public int cmp(Comparable object);
}
```

"OrderedInteger.java" file :

```
class OrderedInteger implements Comparable {
    int value;
    public int cmp (Comparable o) {
       return value - ((OrderedInteger) o).value;
    }
}
```

(*Code 3*)

"OrderedString.java" file :

```
class OrderedString implements Comparable {
    String s;
    public int cmp (Comparable o) {
       return  // lexicographic order code
    }
}
```

To ensure the validity of our modeling choice, here is the same ADTs realization in *Eiffel* (see [Mey92]). It takes a very similar form as above and there is also a similar abstract class named `COMPARABLE` in the standard library (*Kernel Library*). Here, we implemented only the ordered integers case (the one for the strings would be similar):

"comparable.e" file :

```
deferred class COMPARABLE
  feature {ANY}
    infix "<=" (other:like Current): INTEGER is
      deferred
    end;
end -- class COMPARABLE
```

(*Code 4*)

"ordered˙integer.e" file :

```
class ORDERED_INTEGER
 inherit COMPARABLE
 feature
   val : INTEGER;
   infix "<=" (i: ORDERED_INTEGER): INTEGER is
     do Result := Current.val - i.val; end
end -- ORDERED_INTEGER
```

[2]Note that, at this time, we do not attempt to exactly match the above modular solution. We just consider how an usual *Java* programmer would generally solve the given ADT implementation problem.

In order to check the similarity of the *Ocaml* and *Java* codes, here is how they are used when comparing two abstract integers (assume that both codes were completed to provide constructors which take a predefined integer as argument – assume it is named `new` for the *OCaml* version):

In OCaml:

```
let x = Ordered_integers.new 3
let y = Ordered_integers.new 4
Ordered_integers.cmp x y
```

(*Code 5*)

In Java:

```
OrderedInteger x = new OrderedInteger(3)
OrderedInteger y = new OrderedInteger(4)
x.cmp(y)
```

Thus, the modular and the OO versions of (*Codes 1 and 3*) are quite different though having *a priori* the same implementation purpose and being used in a similar way. In particular, the modular version models *ordered sets*, whereas the OO version models *elements of ordered sets*. These two implementations can therefore be assumed as effective candidates to yield differences between the two programming paradigms.

3 Type Components and Their Square of Relationships

In order to analyze the main differences between the two above implementations (*Codes 1 and 3)*, we go back to the definition of an ADT. This definition may require the inclusion of four distinct type-related components: (a) a type for the concrete representation; (b) an encapsulating unit containing both a concrete representation and its associated set of operations; (c) a signature/interface; (d) a declaration of the type of the concrete representation in the signature/interface (the so-called *existential type* [MP85, Pie02]). These four type components may be related by a square of four relationships:

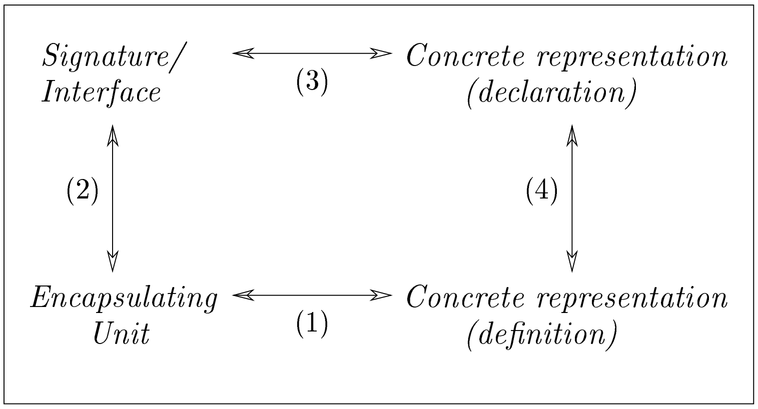
(1) The concrete representation with the encapsulating unit.

(2) The encapsulating unit with the signature/interface.

(3) The declaration of the concrete representation with the signature/interface.

(4) The concrete representation with its declaration.

These relationships are summed up in the following diagram and we henceforth refer to each of them as "*Relationship (.)*".



In view of this set of relationships, here is now the analysis of the ADT implementations given in (*Codes 1 and 3*).

3.1 Analysis of *Relationship (1)*

Let us consider the relationship between the concrete representation and the encapsulating unit in the modular programming version (*Code 1*) and the OO programming version (*Code 3*):

Description:

- In the modular programming version, this relationship is *an inclusion*: since modules can contain any kind of item, there is in general no restriction about *nesting the concrete representation* to abstract it (in (*Code 1*), the concrete representation `element` is included into the module `Ordered_integers`; note that the same kind of relationship holds in the *C* version in (*Code 2*)).

- In the OO programming version, this relationship is *an identification*: the concrete representation of the type is implicitly identified with the set of data variables (the *attributes*) contained in the encapsulating unit (the *class*) (in (*Code 3*), the implicit concrete representation `int val` is identified with the class `OrderedInteger`). Note that the collapse allows the arguments of the associated operations to be typed by the class itself (in (*Code 3*), the function `cmp` has an argument typed by `OrderedInteger`; the same kind of relationship holds in the *Eiffel* version in (*Code 4*)).

Modeling consequences:

- In the modular programming version, the nesting indicates a *hierarchical* relationship between *two* type definitions, i.e. between the concrete representation and the ADT itself. The ADT is indeed expressed as a type of values structured by operations, i.e. a specific *algebra*.

- In the OO programming version, the identification implies that no type definition hierarchy exists: the encapsulating unit directly represents the type itself.

Behavior of the instances of the ADTs:

- In the modular programming version, the instances of the ADT are built from the concrete representation only, and refer to the rest of the module for all their life: this is a kind of *centralized type organization*.

- In the OO programming version, instances can carry the full description (data + operations) of the type it belongs to: this induces a possible *decentralized type organization* where instances become independent.

This centralized/decentralized organization already appears in the use of the ADTs (see (*Code 5*)). Note that in the modular programming case, even if the concrete representation is given as some mutable structure or record, accessors would still be centralized in the module.

Elements of construction:

- In the modular programming version, the concrete representation and the encapsulating module belongs to distinct species with distinct name spaces (in (*Code 1*), the concrete representation `element` is included into the module `Ordered_integers`). This species distinction reflects the fact that modules are not given as constructions dedicated to type implementations.

- In the OO programming version, the identification does not generate distinct species (within the full type system of *Java*, there are however some rare exceptions related to primitive types as `int` in (*Code 3*); on the contrary, `String` is a class of the same species as `OrderedString`).

Composition between ADTs:

- In the modular programming version, one has to deal with the two species in order to compose types and ADTs. There are two cases: one can compose either full ADTs or concrete representations of ADTs. For the latter, one must directly extracts what is inside a module (contradicting masking). For the former, one has to deal with modules, provided that the language allows it in a sophisticated way (parametrized modules in *ML*, i.e. *functors*, play here their major role).

- In the OO programming version, most of the ADTs are at the same level which makes composition easier.

3.2 Analysis of *Relationship (2)*

Let us consider now the relationship between the implementation part of the ADT and its signature/interface part:

Description:

- In the modular programming version, this relationship is *an instantiation*: the signature is a declaration of the ADTs full encapsulating units which become instances of this signature. This defines a "has type" relationship at the level of ADTs (in (*Code 1*), the modules `Ordered_integers` and `Ordered_strings` are instances of `ORDER`).

- In the OO programming version, this relationship indicates a *compatibility link*: the interface, i.e. a fully abstract class, defines a type and the implementation parts of the ADT inherit from it. This defines a "has type" relationship at the level of the instances of the type (in (*Code 3*), the classes `OrderedInteger` and `OrderedString` define subtypes whose instances are compatible with the type defined by the interface `Comparable`). This construction is classical in OO programming since it is able to express the specification and at the same time, a potential compatibility between different implementations. Note that OO languages sometimes have ways of just specifying classes. For instance, in *Eiffel*, one can use the *short form* of the classes defined by `class interface`. But these are not compilable forms and mainly serve for documentation purpose. Within the OO layer of *OCaml*, one can specify classes by the more powerful construction `class type` which is used to apply *upcasts* and to explicitly restrict the type of some types parameters; they are also necessary to fully type a module containing a class. However, abstract classes are in general prefered because of the above reason.

Behavior of the ADTs:

- In the modular programming version, the instantiation relationship indicates a coherence between the ADTs in a whole. But no *a priori* relationship exists between their instances. In (*Code 1*), the modules can both be used as instances of `ORDER`, but individual instances of ordered integers and strings are not compatible.

- In the OO programming version, the compatibility link allows subtyping to hold between the different implementations of the ADT (i.e. *inclusion polymorphism*). By the collapsed *Relationship (1)*, instances of the ADTs are able to become compatible with their unique interface. In (*Code 3*), the individual ordered integers and strings are federated under the interface `Comparable`. In particular, one can consider lists of `Comparable` elements containing instances of `OrderedInteger` and `OrderedStrings` at the same time.

Type checking:

- In the modular programming version, types can be statically checked within each implementation of the signature.

- In the OO programming version, the compatibility link implies possible dynamic upcasts into the interface, which must often be inverted into dynamic downcasts to recover the real type: a well-known unsafe type-oriented operation.

This situation appears in (*Code 3*) in a real bad form as the function `cmp` is binary. In the interface `Comparable`, this function is accordingly declared over instances of `Comparable`. But then, each implementation of `cmp` must also be typed over instances of `Comparable` which is what occurs in the `OrderedInteger` and `OrderedStrings` classes. Indeed, with regard to the compatibility link and the specification given in `Comparable`, there is indeed no *a priori* reason to preclude class comparisons. But then, comparing integers to strings can occur during execution with a type error. The code of the function `cmp` in `OrderedInteger` is therefore decorated by a downcast expression, i.e. `((OrderedInteger) o).value`. If comparing strings to integers does not make much sense, one can consider for instance the existence of another subclass of `Comparable` for ordered floats. The typing of `cmp` would make possible to implement coercion rules between integers and floats.

This inherent type unsafety is generally handled by using *type predicates* acting during execution to check the real identity of the instances: a very similar situation as with a *dynamic type system*. Indeed, the implementation of the comparison function in *Java* could take the following form if one wants to get the exceptions involved under explicit control:

```
public int cmp (Comparable o) throws NotComparableException {
   if (! (o instanceof OrderedInteger))
      throw new OrderedIntegerCompareException();
   return value - ((OrderedInteger) o).value;
}
```

Note that the Eiffel version (see (*Code 4*) uses a trick to instill some static constraints by using so-called *anchored types*: the construction "`like Current`" implicitly imposes a covariant link between the arguments types of a function. But then, covariance in pure inclusion polymorphism is well-known to leave the situation type unsafe.

3.3 Analysis of *Relationship (3) and (4)*

The relationships of the concrete representation declaration with the other type components are discussed together since reflecting what happens in *Relationship (1) and (2)*.

Description:

- In the modular programming version, these relationships are respectively an inclusion and an instantiation: the declaration of the concrete representation (the *existential type*) is nested into the signature part, which is then instantiated in each ADT (in (*Code 1*), the concrete representation declaration `element` is included into `ORDER` and is implemented as `int` in `Ordered_integers` and as `string` in `Ordered_strings`).

- In the OO programming version, since there is an identification between the concrete representation and the encapsulating unit, *Relationship (3)* is also an identification: the declaration of the type is the body of the abstract class (in (*Code 3*), `Comparable` represents the two facets of the declaration of an ADT at the same time). Thus, *Relationship (4)* becomes non-existent and mostly arbitrary. Note that this implicitness explains the use of adjectives for the names of interfaces (like `Comparable` of (*Code 3*) to stress the descriptive character of a list of declared functions.

Type checking:

- The nested declaration makes it possible to explicitly type the declarations of the associated operations (in (*Code 1*), the function `cmp` is typed over `element`). When instantiated, it statically fixes types for the implementation of `cmp`.

- In the OO programming version, since *Relationship (4)* is non-existent, the associated operations cannot be declared as to operate on a specific concrete representation. The associated operations necessarily apply to the ADT in the very general setting given by the interface. And, according to the identification of *Relationship (3)*, the name of the interface can be directly used to type the associated operations (in (*Code 3*), `Comparable` is directly used to type the argument of the declaration of the function `cmp`). The specification is therefore expressed at the level of the interface and not at the level of specific implementation of the ADT: typing behavior cannot be ensured.

4 Emulations

The analysis in the previous section allows us to draw the following main conclusions: First, the modular programming version induces a hierarchical relationship between two distinct types, i.e. the concrete representation and the ADT itself: This induces a particular modeling view (an **algebraic view** of type construction), stronger abstraction of the type and stronger type validity. However, composition is made more difficult and there is no compatibility at the level of single instances. The OO programming version smoothes out the hierarchical relationship between types to allow the notion of *object* to take over, i.e. instances go with their associated operations: This makes possible full compatibility links between different implementations (*inclusion polymorphism*) at the expense of weakening type validity.

We shall now try to obtain some of the properties of each implementation through the other programming paradigm.

4.1 Algebraic View in OO Programming

First, note that the hierarchical modeling view induced by the nesting of the type components in the modular programming version of (*Code 1*) has a general interest. Here are some other examples (some are *multi-sorted*, i.e. they use more than one type for their concrete representations):

- A type for points together with a distance function makes a type for metric space.
- A type for pairs of float together with the operations related to the imaginary value i implemented over polar coordinates makes a type for the complex numbers (or more generally, a type for a commutative field).
- A type for containers and a type for data together with specific managing/requests functions make a type for data bases.

- A type for virtual sheets and a type for sets of windows with a set of managing/decorating functions make a type for work environments

Now, in order to obtain this hierarchical relationship in the framework of OO programming, one must make explicit the distinction between the concrete representation and the encapsulating unit, i.e. *Relationship (1)*. The classical solution in *Java* is to define two or more ADT's at the same level, which are then composed together. This technique leads to the following revisited version of our ADT problem (implemented for the ordered integers ADT case only)[3]:

```
interface Element { }

interface Order {
   int cmp (Element e1, Element e2);
}

class OrderedInteger implements Element {
    int value;
    OrderedInteger (int v) { value = v; }
}

class OrderedIntegers implements Order {
    public int cmp (Element i1, Element i2) {
       return ((OrderedInteger) i1).value - ((OrderedInteger) i2).value;
    }
}
```

(*Code 6*)

However, the result does not reflect the algebraic view: no nesting takes place here. For sure, compositions using these types is made easier, but first, it remains essentially type unsafe for the same reasons as for (*Code 3*)[4], and second, the implementation of the ADTs is now scattered. Nevertheless, nested definitions are possible in some OO languages. Nested classes were already present in the Simula standard library class `Simset` [BDMN75, Szy92] and they are also available in *C++* and *Java*. Another version of the above implementation could then become:

```
interface Order {
    interface Element{};
    int cmp (Order.Element e1, Order.Element e2);
}

class OrderedIntegers implements Order {
    class OrderedInteger implements Order.Element {
        int value;
        OrderedInteger (int v) { value = v;}
    }
    public int cmp (Order.Element i1, Order.Element i2) {
        return ((OrderedInteger) i1).value - ((OrderedInteger) i2).value;
    }
}
```

(*Code 7*)

The hierarchical relationship has been obtained, at least syntaxically, i.e. *Relationships (1) and (3)* became inclusions. However, *Java* does not infer many constraints from this structure (see e.g. [IP02b]). The type relationships remain loose. In particular, the comparison function `cmp` is still unsafe since any instance of type `Order.Element` would be accepted by `cmp`. However, a defensive type checking policy can be organized from this structure by inserting specific constructors dedicated to produce instances of the concrete representation of the ADT. Consider for example an implementation of binary search trees over ordered elements using a similar interface of ordered sets as above (see also [Sma01]): since a whole instance of ordered set can be transmitted, one can impose homogeneous typed trees more easily. With respect to the algebraic view, this is also what would be done in modern modular programming, i.e. defining a binary search tree ADT parametrized by an ordered set ADT (see e.g. [Pau97]). Note however a difference: an instance of `OrderedSet` in the above (*Code 7*) has the same status as any other object, i.e. it is a *first-class citizen*. Thus,

[3] Of course, since `Element` has no specific property, one can replace it by the maximal type in *Java*, i.e. `Object`.
[4] See also [Sun02] about the interface `Comparator` of the *JDK* which is similar to the above interface `Order`.

one can for instance build lists of different ordered sets. In order to obtain the same property in modular programming, one must consider extensions where modules are also *first-class citizens* (see for instance the *ML* dialect *Moscow ML*).

Within OO programming, we can also obtain versions of ADTs with safer type checking by using constructions including some type parameters: a property generally called *genericity*. This mixing of polymorphism mecanisms is possible e.g. in *C++* with *templates*, in *Eiffel* by *generic classes*, and in the OO layer of *OCaml*; *Java* has also non-standardized extensions including genericity (see e.g. *GJ* [BOSW98]). To illustrate this kind of construction, here is a revisited version of the ordered element ADT implemented with the OO layer of *OCaml* (the identifier `'element` denotes the type parameter):

```
class virtual ['element] order =
  object
    method virtual cmp : 'element -> 'element -> int
  end;;

class ordered_integers =
  object
    inherit [int] order
    method cmp x y = x - y
  end;;

class ordered_strings =
  object
    inherit [string] order
    method cmp x y = (* lexicographic order code *)
  end;;
```

(*Code 8*)

In this version, each instance of `ordered_integers` and `ordered_strings` will have a specific comparison method `cmp` with a statically fixed type: the `'element` parameter is instantiated within each class. As a result, there is no type checking weakness. But the price to pay is that compatibility now is constrained by the type parameter instantiation: in particular, instances of `ordered_integers` will no longer be compatible with instances of `ordered_strings`. More generally they will not be compatible with `'element order` but only with `int order`.

This transfer from type flexibility to type validity has been made possible by the introduction of the type parameter `'element`. In fact, this parameter just makes *Relationship (1)* and *(3)* no longer types identifications ! The relationships in (*Code 8*) are almost isomorphic to the ones in (*Code 1*). However, there is still a problem with this version: no real type abstraction takes place. In particular, the type parameters involved must be instantiated by types defined externally to the classes. Definitions of types can therefore be scattered as in the *Java* version of (*Code 6*). Nesting could be again a solution, though not available in the current version of the OO layer of *OCaml* (version 3.06). In this respect, it is interesting to note that a programming idiom appeared in *OCaml* to solve this problem (see e.g. [BHR00, RV02]). One can indeed embed classes into modules to allow one to express a full abstraction of the concrete representation of an ADT . Thus, here is another version of the ordered elements ADT (implemented for the ordered integers only):

```
class virtual ['element] order =
    object
      method virtual cmp : 'element -> 'element -> int
    end;;

module Ordered_integers =
  struct
  type element = int
  class ordered_integers =
    object
      inherit [element] order
      method cmp x y = x - y
    end
  end;;
```

(*Code 9*)

This time, the four type components are present even if the square of relationships has been broken by a new level of encapsulating unit.

4.2 Type Safety in OO Programming

The last versions of our ADTs illustrated the fact that the use of type parameters induces new balances between type flexibility and type validity. Structurally, *Relationship (1)* and *(3)* were again inclusions. In fact, instantiations of type parameters can be computed by an *inference mecanism*, a possibility offered by the OO layer of *OCaml*. Thus, for instance, *Relationship (1)* can appear as an identification though being implicitely realized as an inclusion of a parameter whose instantiation is automatic. Here is another implementation of the ordered elements ADT illustrating this fact (see e.g. [CMP00, RV02]):

```
class virtual comparable =
  object (_ : 'element)
      method virtual cmp : 'element -> int
  end;;

class ordered_integer x =
  object (self)
    inherit comparable
    val int_value = x       method get_val = int_value
    method cmp i = self#get_val - i#get_val
  end;;

class ordered_string x =
  object (self)
    inherit comparable
    val string_value = x    method get_val = string_value
    method cmp i =  (*lexicographic order with self#get_val and i#get_val*)
  end;;
```

(*Code 10*)

This version demonstrates that a type nesting inducing a non-trivial *Relationship (3)* (here, the parameter `'element` is included in the interface `comparable`) does not necessarily induce an explicit *Relationship (1)*. As a result, the algebraic view is seen to be no longer necessary while the typing is safe. However, as for (*Code 8*) the type comparison method `cmp` will be statically fixed for each instance, and accordingly, this will restrain their compatibility (*Relationship (2)* is not anymore a "has type" relationship). In particular, no compatibility is this time available with `comparable`. Compatibility holds only between instances whose type parameters have been instantiated similarly. Also, no type abstraction is available here, inducing again a scattering of the type definitions.

4.3 Collapsing Types in Modular Programming

First, note that the non-hierarchical modeling view induced by the identifications of the type components in the OO programming version (*Code 1*) has a general interest. Often, it simply makes the modeling simpler. As for the algebraic view, here are some other examples:

- A type for metric points defined by a distance function acting on it.
- A type for stacks defined by pop and push functions acting on it.
- A type for mobile windows defined by moving functions acting on it.

In each example, the ADT is not the instance of another type as in the algebraic case. However, to explicitly type the declared elements of a signature in modular programming, one needs a third type: no identification is in general available (*Relationship (3)* remains an *inclusion*). In practice, and because of the above remarks, it is very often simulated. But in general, the only tool to smooth out the level distinction is a naming convention ! The first idea is then just to make a multiple use of the same name to simulate the type identification. For instance (*Code 1*) can simulate (*Code 3*) as follows:

```
module type COMPARABLE =
    sig
      type comparable
      val cmp: comparable -> comparable -> int
    end;;
```

(*Code 11*)

```
module Ordered_integer =
  struct
    type comparable = int
    let cmp x y = x - y
  end;;
```

The result is that the function `cmp` has a meaningful type, but its full type is redundant:

```
Ordered_integer.cmp :
   Ordered_integer.comparable -> Ordered_integer.comparable -> Ordered_integer.comparable
```

A traditional technique to alleviate this problem is instead to use neutral and non-informative names for the concrete representations of the ADTs. In the *ML* literature, single letters like "`t`" are often used [Pau97, Oka98, LDG+02]:

```
module type COMPARABLE =
    sig
      type t
      val cmp: t -> t -> int
    end;;
```

(*Code 12*)

```
module Ordered_integer =
  struct
    type t = int
    let cmp x y = x - y
  end;;
```

The result is a loss of naming precision within the encapsulating units, but the full names and types still include the needed information, e.g.:

```
Ordered_integer.cmp : Ordered_integer.t -> Ordered_integer.t -> Ordered_integer.t
```

Note that this kind of naming convention is also classically applied in other modular-oriented languages. For instance, it is discussed in the case of *C* in [Han97] where the redundancy naming problem is also tackled. Nevertheless, we stress that the above codes are isomorphic to the first ones (see (*Code 1*)). As a matter of fact, the concrete relationships of usual modular programming impose a level jump to a set representation. In other words, no real new property is obtained through this naming trick, only a superficial simplification of the underlying model.

In fact, actual type collapse is often concomitant with the notion of *objects*, i.e. the identification in *Relationship (1)* leads to associate values with their associated operations. This can be approximated in modular programming when modules themselves can represent objects. This is possible in modular languages where modules functions exist (this technique appears in [Pau97]). For instance, in *OCaml* one can express the ordered integers ADT as follows:

```
module type INT_COMPARABLE =
  sig
    val value : int
    val cmp: int -> int -> int
  end;;
```

(*Code 13*)

```
module Ordered_integer (I: sig val value: int end) : INT_COMPARABLE =
  struct
    let value = I.value
    let cmp x y = x - y
  end;;
```

Each application of the functor `Ordered_integer` gives a new module instance which encapsulates a specific integer. This method makes up difficult to obtain generic versions since there are no longer any type parameters available. Moreover, this version is impaired as soon as modules are not fully first-class citizens. Thus, in the current version of *OCaml*, instances of the above integers could not be contained in data structures. Also, no compatibility is available between different comparable instances. This property is discussed further in the next section.

4.4 Instances Compatibility in Modular Programming

One of the shortcomings of the modular programming version in (*Code 1*) compared to the OO programming version in (*Code 3*) is that instances of the ADTs are not compatible. However, here we show a possible technique to obtain this property.

The idea is to transform each instance of a modular ADT into a particular record containing all the necessary information, i.e. emulating a simple *record model* for OO programming (an idea e.g. used in [Mag01]). Instances of records in *OCaml* are indeed first-class citizens possibly containing functions. Here, we show how this technique works in the case of our ADT example with a fully implemented abstraction. A distinction must then be made between the masked concrete representation type `element` and its external representation – call it `extern_element`. This distinction allows us to generically type the constructor `make` as "`extern_element -> element`". Two constraints must then hold: first, the ADT must propose an inverse of `make` called `show`. Second – an effect of the static type checking –, compatibility is only ensured for external representations whose type is the same. Thus, first, here is another version of our ordered elements ADT with a full realized abstraction:

```
module type ORDER =
  sig
    type element                          type extern_element
    val make : extern_element -> element  val show : element -> extern_element
    val cmp : element -> element -> int
  end;;

module Ordered_integers :
    ORDER with type extern_element = string =
  struct
    type element = int                     type extern_element = string
    let make s = int_of_string s           let show i = string_of_int i
    let cmp = compare
  end;;

module Ordered_strings :
    ORDER with type extern_element = string =
  struct
    type element = string                   type extern_element = string
    let make s = s                          let show s = s
    let cmp = compare
  end;;
```

(*Code 14*)

Then the two implementation parts can be made compatible by the following adapter module which contains a functor generating a specific adapting function:

```
module Order_Record =
  struct
    type 'a r_element = { elt : 'a; cmp : 'a r_element -> int }

    module Make_Adapter (O: ORDER) =
      struct
        let rec adapter e1  =
          { elt = O.show e1; cmp = (fun {elt=e2} -> O.cmp e1 (O.make e2))}
      end
  end;;
```

(*Code 15*)

This module can then be used to create specific adapters which can be used to transform the instances of the ADTs into their record forms:

```
module MR_integers = Order_Record.Make_Adapter (Ordered_integers)
module MR_strings  = Order_Record.Make_Adapter (Ordered_strings)
```

Thus, this laborious technique makes up possible to obtain instances of ordered integers and ordered strings which are first-class citizens and which are compatible up to the typing of the elements of their respective records.

5 Conclusion

The emulation experiments with modular programming have not been very satisfactory as far as obtaining similar properties as the ones of the OO programming versions. First, simple naming conventions to simulate identifications have been shown to be useful but only for superficial simplifications of the underlying model. These conventions were essentially symptomatic of the need of building ADTs in terms of the relationships used in (*Code 3*). They induce no actual effect on the behavior of the produced instances of and ADT. Second, notion of objects and compatibility between instances were awkwardly obtained in (*Codes 13 and 15*). As a result, the ADT modular versions are mostly left out with their initial qualities, i.e. good type abstraction, well-expressed algebraic view and complete type safety. However, the discussions about the ADT versions of (*Code 7*) and (*Code 13*) indicate that this situation could be improved by using sophisticated *first-class modules* (see [Rus98, RRS00]). Also, the difficulties of composing ADTs are also expected to be improved in the future, e.g. by *mixins* (see e.g. [DC96, CHP99, AZ01, HL02]).

The emulation experiments of OO programming were a little more satisfactory with respect to type nesting and type safety. Nesting classes has been shown to be a possible way of obtaining hierarchical type relationships and type abstractions. However, the *Java* version of (*Code 13*) shows inherent weakness, mostly because of its liberal semantics with respect to inner classes (see [IP02b]), and also because nesting is available under pure inclusion polymorphism so that the implementation remains as unsafe as before. It seems indeed to be a difficult task to mix the *nesting dimension* with the *subtyping dimension*. Nevertheless, this mixing seems a promising way of making up more sophisticated OO systems (see e.g. [Szy92, Sma01, IP02b]. Genericity has been shown to be a better alternative in terms of type safety with (*Codes 8, 9 and 10*). However, it induces some weakness in terms of type abstraction. Research is on its way to find solutions to this problem, for instance with *virtual types* (see e.g. [BOW98, TT99, IP02a]) or with *dependent types* (see [OCRZ02]).

Thus, as announced, this paper does not describe a sharp distinction between the two programming paradigms. It just gives a way of pointing up what seems to be easier in each one. The emulations were not really satisfactory, but they gave hints about the paths each paradigm could follow to improve their states. With this respect, the square of the four relationships has been our main tool to describe the similarities and the differences between all the presented versions of the same ADT. Of course, the chosen programming example has been a very simple one (except the not-so-innocent use of a binary comparison function), making our conclusions necessarily partial. However, the fact that this simple ADT could take so many effective forms is an indication of the complexity of the comparison question.

References

[App97] A.W. et al. Appel. *The Standard ML Basis Library*. Bell Labs, Lucent Technologies, 1997.

[AZ01] D. Ancona and E. Zucca. A calculus of module system. *Journal of functional programming*, 2001. in press.

[BDMN75] Graham M. Birtwistle, Ole-Johan Dahl, Bjoern Myhrhaug, and Kristen Nygaard. *SIMULA begin*. Petrocelli/Charter, New York, 1975.

[BHR00] S. Boulmé, Th. Hardin, and R. Rioboo. Polymorphic data types, objects, modules and functors: is it too much? Technical Report 14, Paris VI, LIP6, 2000.

[Blo01] J. Bloch. *Effective Java (Programming Language Guide)*. The Java Series. Addison-Wesley, 2001.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamine, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98*, 1998.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–549, 1998. FOOL 5.

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec O'Caml*. O'Reilly, 2000. (English translation available in the O'Caml URL).

[Coo90] W.R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object Oriented Languages: Proceedings of REX School/Workshop*, number 489 in Lecture Notes in Computer Science, pages 151–178. Springer, 1990.

[DC96] Duggan D. and Sourelis C. Mixin modules. In ACM press, editor, *Intnl. Conf. on Functional programming*, pages 262–273, 1996.

[FG93] A. E. Fisher and F. S. Grodzinsky. *The Anatomy of Programming Languages.* Prentice-Hall, 1993.

[Gut77] J.V. Guttag. Abstract data types and the development of data structures. *Comm. ACM*, 1977.

[Han97] D.R. Hanson. *C Interfaces and Implementations.* Professional Computing Series. Addison-Wesley, 1997.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 123–137, 1994.

[HL02] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP*, volume 2305 of *Lecture Notes in Computer Science.* Springer, 2002.

[IP02a] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 1(175):34–49, 2002.

[IP02b] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 1(177):56–89, 2002.

[LDG+02] X. Leroy, Doligez D., J. Garrigue, Rémy D., and Vouillon J. *The Objective Caml system. Documentation and user's manual (3.06).* INRIA, August 2002.

[Ler99] X. Leroy. Objects and classes versus modules in Objective Caml. Intl. Conf. on Functional Programming (slides of invited talk), 1999.

[Mag01] L. Magdelaine. Maquettage d'un simulateur en O'Caml. Technical report, University Bordeaux I and EDF (département M.O.S), 2001.

[Mey92] B. Meyer. *Eiffel, the Language.* Prentice-Hall, 1992.

[MP85] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. In *Proc. of the ACM Symp POPL*, 1985.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[OCRZ02] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. Technical Report IC/2002/070, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, 2002.

[O'H98] P.W. O'Hearn. Polymorphism, objects and abstract types. *SIGACT News*, 29(4):39–50, 1998.

[Oka98] Ch. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[Pau97] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1997.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Rey75] J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In A. (editor) Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. INRIA, 1975. IFIP Working Group 2.1 on Algol.

[RRS00] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Owner's Manual*, june 2000.

[Rus98] C. Russo. *Types for modules.* PhD thesis, Univ. of Edinburgh, 1998.

[RV02] D. Rémy and J. Vouillon. *Objects in Caml.* INRIA, 2002. part of [LDG+02].

[Sma01] Y. Smaragdakis. Interfaces for nested classes. Eighth Workshop on Foundations of Object-Oriented Languages Workshop (FOOL 8), January 2001.

[Sun02] Sun. *Java 2 Platform, Standard Edition*, 2002. Version 1.4.1, API spec.

[Szy92] Clemens A. Szyperski. Import is not inheritance: Why we need both: modules and classes. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 19–32, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[TT99] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized types. In Springer-Verlag, editor, *European Conference on Object-Oriented Programming*, pages 186–204, 1999. LNCS 1628.