ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Extensible Objects for Database Evolution: Language Features and Implementation Issues

Antonio Albano, Milena Diotallevi and Giorgio Ghelli

Published in collaboration with the
British Computer Society

BCS

Springer

Extensible Objects for Database Evolution: Language Features and Implementation Issues

Antonio Albano, Milena Diotallevi, Giorgio Ghelli
Università di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
e-mail: albano@di.unipi.it, ghelli@di.unipi.it

Abstract

One of the limitations of commercially available object-oriented DBMSs is their inability to deal with objects that may change their type during their life and which exhibit a plurality of behaviors. Several proposals have been made to overcome this limitation. An analysis of these proposals is made to show the impact of more general modeling functionalities on the object implementation technique.

1 Introduction

In the last decade many database programming languages and database systems have been defined which are based on the object paradigm. Some of these systems are based on designing from scratch an object data model, and a database programming language; for example, Gemstone, ObjectStore, Ontos, O2, and Orion. Other systems are based on the extension of the relational data model with object-oriented features, as in the Illustra and UniSQL systems, and in the forthcoming new SQL standard, called SQL3. The success of the object data model is due to the high expressive power which is obtained by combining the notions of object identity, unlimited complexity of object state, class inclusion, inheritance of definitions, and attachment of methods to objects. However, this data model is not yet completely satisfactory when entities need to be modeled which change the class they belong to and their behavior during their life, or entities which can play several roles and behave according to the role being played.

For example, consider a situation with persons classified either as generic persons, students or employees. This situation is modeled by three object types `Person`, `Student`, and `Employee`, in any object system. In this situation it is important to allow an object of type `Person` to become a `Student` or an `Employee`. However, this may lead to problems. Suppose that a `Code` field has been defined for both students and employees, with a different meaning and even a different type, integer and string respectively. Let a person John first become a student with code 100 and then an employee with code “ab200”. At least four choices are possible:

1. The new `Code` overrides the old one, which makes no sense.
2. The situation is avoided, either statically, by preventing the declaration of a `Code` field in two subtypes of `Person`, or dynamically by forbidding any object which already has a `Code` field to acquire a new `Code` field. These two approaches are unacceptable, since they create some form of dependency between two different object types, `Student` and `Employee`, which are “unrelated”, i.e. such that they don’t inherit from each other. In any object oriented methodology it is essential that a programmer defining a subtype only has to know about its supertypes, and must not fall into errors, either static or dynamic, which depend on the existence of another descendent of a common ancestor.
3. The situation can be prevented by stipulating that an object must always have a most specific type, i.e. that it can acquire a new object type only if this new type is a subtype of its previous most specific type. This solution is better than the previous one, since it does not link the possibility of extending an object to the rather irrelevant event of a common field name between two unrelated types, but it imposes too strong a constraint on the object extension mechanism.
4. A `Code` message to John gets a different answer depending on whether John is seen as a `Student` or as an `Employee`. This is the best solution.

The above problem is not just a consequence of using the same name `Code` for two different things, but is only one example of the fact that, when objects are allowed more than one most specific type, it is necessary to avoid interactions between these unrelated types, which can best be obtained by allowing objects to have a “context dependent” behavior. Context dependent behavior can be supported in two different ways:

1. By static binding: the meaning of a message sent to an object is determined according to the type that the compiler assigns to the object, or in some other static way. This solution produces efficient code, since no method lookup is needed, but heavily affects the features of code extensibility and reusability which characterize object-oriented programming and which are due to the combined effect of inheritance with dynamic binding of methods to messages.
2. By dynamic binding (also called “late binding” or “dynamic lookup”): in this case an object may have several “entry points”, which we call “roles”; for example, when a Student is extended to become an Employee, it acquires a new role (or entry point) which will be used when it is seen as an Employee, without losing its old Student role. Messages are always addressed to a specific role of an object, and method lookup starts from the addressed role. An object which is always accessed through its most specific role behaves exactly like an object in a traditional object oriented language.

The languages proposed to deal with extensible objects may be classified as follows [ABGO93]:

- languages with *dynamic binding* and *uniform behavior* (e.g., Galileo [ACO85], [AGO91]);

- languages with *static binding* and *context dependent behavior* (e.g., Clovers [SZ89], Views [SS89], IRIS [FBC$^+$87] and Aspects [RS91]);
- languages with both *dynamic binding* and *context dependent behavior* (e.g., Fibonacci [AGO95, ABGO93]).

In this paper we discuss some possible linguistic and implementative issues which arise when both dynamic binding and context dependent behavior are supported. We draw on the experience gained in the design and implementation of the Galileo and Fibonacci object-oriented database programming languages.

The linguistic model we present is not essentially new, as it is based on the role mechanism of Fibonacci. The focus of the paper is not on the mechanism itself, but on its effect on object representation, a point which is not usually discussed in the literature. In particular, we show how the various parts of an object representation are related to the language features by showing how object representation changes when new features are introduced after each other. We study a basic representation model, which is neither unrealistic nor optimized, but this study also gives information about optimized object representation techniques. In fact, the basic model only contains the information which is needed to implement the operational semantics of the object operations. Hence, every time some information must be added to the structure of the basic model to deal with a new linguistic feature, the same amount of information must show up, in some way, also in every optimized object representation. By giving an object representation model, and an implementation of the basic object operations on that model, we also define an informal semantics for the role mechanisms we present.

The paper is organized as follows. Section 2 gives a basic linguistic and implementative model for an object-oriented language without extensible objects. Section 3 extends the linguistic model with extensible objects with uniform behavior, and shows how extensibility affects the implementation model. Section 4 studies how the possibility of shrinking objects affects the implementation model. Section 5 further extends the language with a role mechanism, i.e. with context dependent behavior with dynamic binding, showing the effects on the implementation model. Section 6 draws some conclusions.

2 Non-extensible Objects

In this section we define a basic object-oriented language, without extensibility, with the associated object representation model. To fix a notation, throughout the paper we adapt the syntax and semantics of the Galileo 95 language [AAG95].

2.1 The language

2.1.1 Object Types

We assume that an object type specifies three pieces of information:

- the object type interface, i.e. (a) the set of messages which can be sent to the object, with the parameter and result types for every message, and (b) the object instance variables, i.e. the components of the object state, which can be accessed from outside the object;
- the structure of the object state, i.e. the name, type, and mutability of the instance variables;
- the method implementation, i.e. the code that an object of that type executes when it receives a message.

In other languages, such as Fibonacci, an object type only specifies the interface of objects of that type, while every object can have, in principle, its own implementation, i.e. its own state structure and method set. We also assume here that all the components of the state of an object can be accessed from outside the object; these assumptions have some consequences on object representation, which we cannot address here for space reasons.

The following example shows the definition of the object type `Person`, with a method `Introduce`:

```
let rec type
   Person = object [Name :string;
                    BirthYear :int;
                    Phones :[House :string];
                    Introduce:= fun () :string is
```

```
                                              implode({"My name is  ";
                                                        self.Name })
                                  ];
```

The declaration `let type T = object [F;M]` defines a new object type `T` and a function `mkT` which, given a tuple of type `[F]`, builds an object of type `T`. The signature `F;M` is composed by a set `F` of label-type pairs `(`a_i `:`T_i`)` which introduce the components of the object state (the identifiers a_i are called *attributes*), and by a set `M` of label-function pairs (m_i `:= fun(`...`)` ...`)` which define the methods used by the objects of that type to answer to the corresponding messages. Field selection and message passing are expressed, respectively, as `obj.a` and `obj.m(`p_1`,...,`p_n`)`. A method can recursively access the object which received the message using the predefined identifier `self`. Finally, `[House :string]` is a tuple type, {`"a";"b"`} is a sequence of string, `implode` concatenates a sequence of strings. Each application of the `mkT` constructor returns an object of type `T` with a different identity.

The following piece of code builds an object of type `Person`, accesses one of its fields, and sends it a message.

```
  let John := mkPerson ([Name := "John Smith";
                         BirthYear := 1967;
                         Phones := [House := "06 222444"] ]);

  john.BirthYear;
  john.Introduce();
```

2.1.2 Subtyping and Inheritance

Subtyping and inheritance are two different mechanisms which are often related in object oriented languages. Subtyping is an order, or preorder, relation among types such that whenever T' is a subtype of T, written $T' \subseteq T$, any operation which can be applied to any object of type T' can also be applied to any object of type T. Inheritance is a generic name which describes any situation where an object type, object interface, or object implementation, is not defined from scratch but is defined on the basis of a previously defined entity of the same kind. For example, in our situation defining an object type T' by inheritance from T means defining T' by only saying which new attributes and methods must be added, and how the methods and attributes of T must be modified. More precisely, it is only possible to specialize the type of an attribute (specializing means substituting with a subtype), and a method can be substituted by any other implementation, but its type can only be specialized.[1] The constraint that attributes and methods can only be added or specialized is called *strict inheritance*, and implies that an object type which is defined by inheritance from T is also a subtype of T.

We adopt the following syntax for the definition, by inheritance, of a subtype `T` of an object type `T'`:

```
type T := object is T' and H
```

`H` specifies the properties (attributes and methods) to add or redefine in `T`; below is an example.

```
  let rec type Student := object is Person and
                          [Code :string;
                           Faculty :string;
                           Introduce := fun () :string is
                                implode({super.Introduce();
                                         " I am a student of  ";
                                         self.Faculty}) ];
```

It is generally possible to define an object type by inheritance from several object types: `T := object is T`$_1$`, ..., T`$_n$ `and [F;M]` (multiple inheritance). The `mkT` function expects, in this case, a record with the attributes in `F` and with all the other attributes inherited from `T`$_1$`, ..., T`$_n$.

If the supertypes have a property with the same name and different types, the property is inherited from the last (w.r.t. the `T`$_1$`, ..., T`$_n$ order) supertype which defines it.[2] In this case, strict inheritance means that every property

[1]The type $S' \rightarrow U'$ of a method m is a subtype of the type $S \rightarrow U$ when $S \subseteq S'$ and $U' \subseteq U$; the inversion of the direction of the comparison between S and S' is explained in [Car88].

[2]In other languages, in this situation the property must be explicitly redefined in `F;M`.

which is either redefined *or* inherited from more than one type must have a type which is a subtype of the type of the same property in *all* the ancestor types $\mathtt{T}_1, \ldots, \mathtt{T}_n$.

2.1.3 Method Lookup and Semantics of Self

When a message `m` is sent to an object `O`, two problems must be solved: (a) which method is used to answer the message, and (b) which is the semantics of the pseudo-variable `self` which may appear in the selected method.

In traditional object-oriented languages, with objects that cannot change their type dynamically, the run-time type `T` of an object `O` is fixed when the object is created. This run-time type is generally only a subtype of the compile-time type of any expression whose evaluation returns `O`. When a message is sent to `O`, the method is first searched for in its run-time type `T`. If none is found, the method is searched for up the supertype chain of `T`. The search will stop, since static typechecking ensures that the method has been defined in one of the super-types. The fact that the method lookup starts from the run-time type `O`, rather than from the compile-time type of the expression which returns `O`, is called *dynamic binding*, while the specific algorithm used to look for the method (depth-first upward search, in this case) is called the *lookup algorithm*.

Consider now a `self.msg(...)` invocation found inside a method defined for the message *msg2* inside type T, and suppose that the method is executed by an object with a run-time type T', which inherits the method for *msg2* from T. Two choices are possible, in principle, for the semantics of `self.msg(...)`:

- method lookup for *msg* may start from the statically determined type T (static binding of self)
- method lookup for *msg* may start from the dynamic type T' of the object which has received the message *msg2* (dynamic binding of self).

The second choice is the one adopted in all object-oriented languages, and is essential in many typical object-oriented applications. Hence, when the method where `self.msg(...)` is found is type-checked, the type checker can only assume that *self* will be bound to an object whose type inherits from T. This is not a problem in languages which only allow strict inheritance, such as the one we are describing, and this is the main justification for the strict inheritance constraint.

The pseudo-variable `super` can also be used in a method expression. `super` is statically bound, i.e. the method search for a message sent to `super` begins with the supertype of the type where the method is defined.

2.2 An Implementation Model

We now describe an implementation model for the basic language described so far. We only focus on the information that must be present in the run-time representation of an object to support the described functionalities.

Abstractly, an object contains three pieces of information: (a) the values of its instance variables (the state), (b) an attribute lookup structure to map each attribute to its position in the state, and (c) a method lookup structure to map each message name to the corresponding method. In some situation, say single inheritance, the compiler can precompute the attribute and method positions for an object from its static type; we will not consider this possibility any further, since it disappears as soon as we add object extensibility. The simplest method lookup structure is obtained by building, for any object type defined as `object is T1,T2 and [F;M]`, a structure containing a lookup table for the methods in `M` plus a list of references to `T1,T2`; we call this structure an "Local Method Table" (LMT). A method is then searched in the LMT of the run-time type of the receiving object, and, if it is not found there, in the LMT's of the ancestor types.

Alternatively, for each object type a table can be built which directly maps every message, owned or inherited, to the corresponding method, to avoid the graph search; we call this structure a "Full Method Table" (FMT). This approach consumes some more space, but makes method lookup more efficient, and is especially convenient in a database language, where the space taken by the FMT structures is negligible with respect to the space taken by the objects. As for the attribute lookup structure, the possible structures are the same as for method lookup, but the efficiency tradeoff is different; in particular, we will always assume a flat structure mapping attributes to positions, which we call the Full Attribute Table (FAT).[3]

Hence, in the basic model an object is represented by a reference to a structure which contains its Full Method Table and its Full Attribute Table, (we call it the Full Object Type Descriptor), and by its state, as depicted in Figure 1. Notice that the Full Object Type Descriptor is shared by all the object with the same run-time type.

[3]Real language implementation may vary between the the fastest solution, where positions are statically computed (when possible), to the

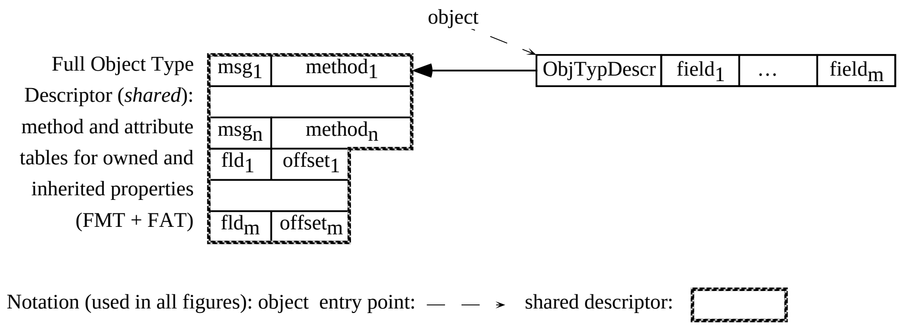
object
Full Object Type Descriptor (*shared*): method and attribute tables for owned and inherited properties (FMT + FAT)
msg1
method1
msgn
methodn
fld1
offset1
fldm
offsetm
ObjTypDescr
field1
...
fieldm
Notation (used in all figures): object entry point:
shared descriptor:

Figure 1: The structure of an object which does not change type.

3 Extensible Objects with Uniform Behavior

In this section we add, to the basic model, the possibility of extending objects, but without introducing the notion of a context dependent behavior. We then show the linguistic and implementative effect of this first extension. This section is based on the linguistic and implementative model which underlies the first version of Galileo [ACO85].

We first extend the basic language by stipulating that, when an object type `T′` is defined by inheritance from type `T`, two functions are automatically generated: `mkT′` to construct directly new instances of type `T′`, and the function `inT′` to transform an instance of type `T` into an instance of the new type `T′` without affecting the object identity; we call this operation "extension".

The function `inT′` has two parameters: the value of the object `O` to be extended and a record which gives the values of the `T′` attributes which are not inherited from `T`.

To solve the problem created by the presence of two properties, in two independent subtypes, with the same name but a different type, the following *property type specialization rule* is adopted: when an object `O` with a set of properties $\mathcal{A}$ is extended with a new type `T`, for every property `P` which is both in $\mathcal{A}$ and in `T`, the type of `P` in `T` (the new type of `P`) must be a subtype of the type of `P` in $\mathcal{A}$ (the old type of `P`).

For example, the object `john` may be extended with the type `Student` as follows:

```
let rec type Student = object is Person and
                   [Code :string;
                    Faculty :string;
                    Phones :[House :string; GuestHouse :string];
                    Introduce := fun () :string is
                                  implode({super.Introduce();
                                           " I am a student of  ";
                                           self.Faculty})
                   ];
let johnAsStudent := inStudent(john,
                             [Code := "0123";
                              Faculty := "Science";
                              Phones := [House := "06 222444";
                                         GuestHouse := "552244"]]);
```

The extension operator does not change the object identity.

Suppose now that the following types are also defined:

```
let rec type Athlete = object is Person and
                            [Code:int;
                             Sport: string;
                             Introduce:= fun () :string is
                                implode({super.Introduce();
```

simplest solution where the object state is represented by a sequence of name-value pairs.

```
                                                " I practice ";
                                                self.Sport}) ];

let rec type Employee = object is Person and
                            [Code:string;
                             Company: string;
                             Introduce:= fun () :string is
                                implode({super.Introduce();
                                         " I work at ";
                                         self.Company}) ];
```

An object of type `Person` which has never been extended to a `Student` can be extended to become an `Athlete`, but the property type specialization rule prevents the extension of a `Student` to an `Athlete`, since the type of the `Code` field in the new type `Athlete` is not a subtype of the type of the same field in the old type `Student`. However, the same rule allows a `Student` to be extended to an `Employee`.

Other operators defined on extensible objects in this language are:

- `Expr isalso T`, to test whether an object denoted by the expression `Expr` also has the type `T`; for example both `john isalso Student` and `johnAsStudent isalso Student` are true.
- `Expr As T`, to coerce an object denoted by the expression `Expr` to one of its possible types `T`; for example `john As Student` returns the object with type `Student`. This operation raises a run-time failure if the object never acquired type `T`, but has no other run-time effect in this language.

3.1 Method Determination

In Galileo, method lookup cannot only depend on the minimal type of an object, since, thanks to object extension, an object may have more than one minimal type. In Galileo, method lookup depends on the whole *object type history*, which is defined as the ordered set of types $\{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$ such that $\mathtt{T}_1$ is the type where the object has been built, and every extension operation adds a new type at the end of the history.

When an object with a type history $\{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$ receives a message `m`, the method to execute is searched for in two steps:

1. first, the method is looked for among the methods that belong to (i.e. are not inherited) the last type $\mathtt{T}_n$ acquired; if it is not found there, the search goes on in the type history, in the inverse temporal order $\mathtt{T}_{n-1}, \mathtt{T}_{n-2}, \ldots, \mathtt{T}_1$ (*history lookup*);
2. then, if the method is not even found in the construction type $\mathtt{T}_1$, the search goes up the supertype chain of $\mathtt{T}_1$ as in the basic language. Static typechecking ensures that the search will eventually find the appropriate method (*upward lookup*).

For example, an object `john` created with type `Person`, and then extended with the subtypes `Student` and `Athlete`, and finally with `Student` subtype `GraduateStudent`, will answer the message `Introduce` using the method defined in the type `GraduateStudent`.

The semantics of `self` and `super` is the same as in the basic model: when a message is sent to `self`, it is dynamically looked up starting from the last acquire type, while a message sent to `super` is statically bound to the corresponding method.

3.2 The Implementation Model

The simplest run-time representation of objects in this language contains the object type history, represented as a modifiable sequence of references to *type descriptors*, the attribute table, and the object state. A type descriptor, in this case, contains the Local Method Table, as defined above, and the type name, which is needed to implement the `isAlso` and `As` operations. Method search is performed in the type descriptors graph, with the two-phase algorithm described above. When an object is extended with a new subtype, a new type descriptor is added to its history and the new fields are added to its state and to its attribute table; if an attribute of the supertype is redefined, its value is directly replaced by the new value.

This simple representation is shown in Figure 2. Note that the object is accessed indirectly to allow the object to be extended without modifying its identity, so that any external reference to the object is preserved; any other technique to allow identity preserving extensibility (e.g., concatenating new fields to the object tail) would work.

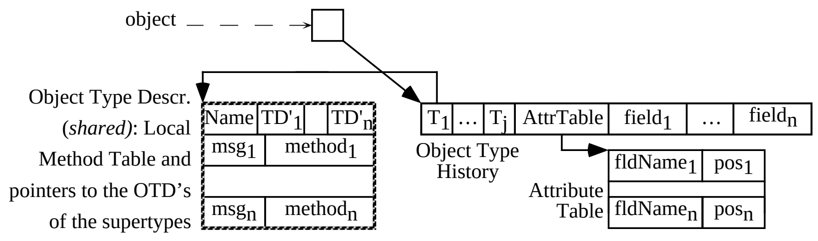

Figure 2: The structure of an extensible object.

To obtain a more efficient execution of message passing, an object representation can be used which closely resembles the one in Figure 1. In this case, each object only contains its state and a reference to a Full Type History Descriptor. The Full Type History Descriptor contains the object history (a list of references to the corresponding Object Type Descriptors) and the full method table and the full attribute table which correspond to that history. The system maintains a pool of Full Type History Descriptors, so that they can be shared among objects with the same history, and creates a new FTHD only when an object is created whose history is new (Figure 3).

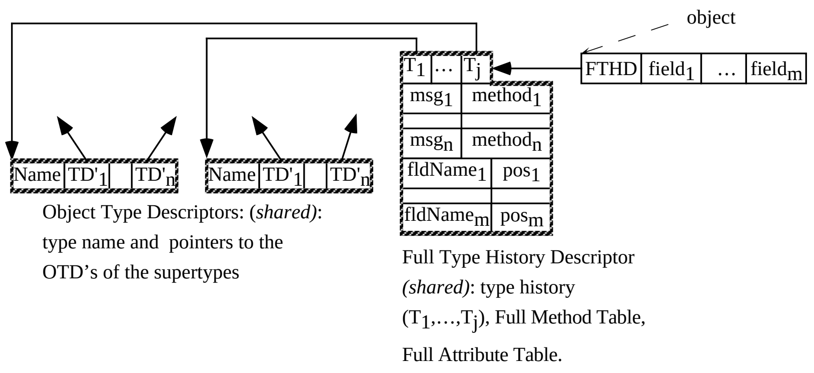
object
FTHD
field1
...
fieldm
T1
...
Tj
msg1
method1
msgn
methodn
fldName1
pos1
fldNamem
posm
Name
TD'1
TD'n
Name
TD'1
TD'n
Object Type Descriptors: (*shared*): type name and pointers to the OTD's of the supertypes
Full Type History Descriptor *(shared)*: type history (T1,...,Tj), Full Method Table, Full Attribute Table.

Figure 3: A better structure for an extensible object.

Hence, we may conclude that adding extensible objects with uniform behavior, to a language with multiple inheritance, costs one indirection level for each object, which allows it to preserve its identity when it is extended, and some space for the Full Type History Descriptors pool.

4 Extensible and Shrinkable objects

As a further generalization step, we now add an operator `dropT(Expr)` to the language, to cancel the type `T`, and all its subtypes, from the object denoted by the expression `Expr`. `dropT(Expr)` is a function which is declared automatically when a subtype is defined, as it happens with `mkT` and `inT`.

In our linguistic model, object shrinkability adds a first kind of context-dependent behavior. Let $\mathtt{Ide}_1$ be an identifier bound to an object of type $\mathtt{T}_1$ (e.g., $\mathtt{Ide}_1$ `:= mkPerson(...)`), and $\mathtt{Ide}_2$ an identifier bound to the same object extended with the subtype $\mathtt{T}_2$ (e.g., $\mathtt{Ide}_2$ `:= inStudent(`$\mathtt{Ide}_1$`, ...)`). If type $\mathtt{T}_2$ is removed from the object, by executing either $\mathtt{dropT}_2\mathtt{(Ide}_2\mathtt{)}$ or $\mathtt{dropT}_2\mathtt{(Ide}_1\mathtt{)}$, then:[4]

[4]Alternative, reasonable, semantics, may be defined, but we do not explore here this issue.

- if the object is accessed through the identifier $\texttt{Ide}_2$, a run-time failure will arise when a message is sent to it, either to execute a method or to extract the value of an attribute, irrespective of whether the property is defined in $\texttt{T}_2$ or is inherited from $\texttt{T}_1$;
- if the object is accessed through the identifier $\texttt{Ide}_1$, a message to execute a method or to extract the value of an attribute defined in $\texttt{T}_1$ is normally executed. Note that, in a well typed program, it is not possible to extract a property which is only defined in $\texttt{T}_2$ by going through $\texttt{Ide}_1$.
- the `isalso` and `As` operators can still be applied to $\texttt{Ide}_2$, to verify whether the object still belongs to some type and to send messages to the part of the object that is still valid.

Shrinkable objects thus have a behavior which depends on the context they are accessed through, their *role* in our terminology. For this reason, the implementation model must be extended to take into account the fact that an object can be accessed through many different roles. Every role contains the following information: the creation type (e.g., $\texttt{Ide}_1$ is associated with the `Person` type while $\texttt{Ide}_2$ is associated with the `Student` type), the validity (e.g., $\texttt{Ide}_1$ is valid while, after the $\texttt{dropT}_2$ operation, $\texttt{Ide}_2$ is not valid any more), and a reference to the object. The object itself must contain the state and a reference to all of its roles, both to implement `As` and `isalso` and to find every role associated with a subtype of `T` when `dropT` is executed.

This representation is shown in Figure 4, where an object with two valid roles and one removed role is represented.

In the previous section we have noticed that an indirection level is needed to allow identity preserving object extension; in this case, the indirection level which is given by the roles can be exploited to this aim.

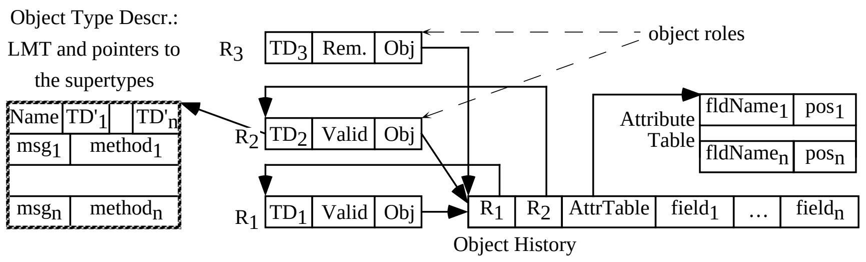

Figure 4: The structure of a shrinkable object.

Every value of type object is actually represented by a reference to one of its roles. When an object is extended a new role is added, and when an object loses the type $\texttt{T}_i$, the following actions are executed:

- the status of the $\texttt{T}_i$ role becomes `removed`;
- the $\texttt{T}_i$ role is removed from the object type history (R_1, R_2 in Figure 4);
- the first steps are repeated for every role of the object whose type is a subtype of $\texttt{T}_i$.

As before, we may modify this implementation by substituting object type descriptors with full message and attribute tables, and by sharing them between objects with the same history, but in this case the definition of history sameness is slightly more complex, since `drop` operations must be taken into account.

5 Extensible Objects with Context Dependent Behavior

The most general solution to support objects which can dynamically acquire new types and exhibit a plurality of behaviors was first given in Fibonacci [ABGO93], and then adapted to Galileo 95 [AAG95]. The Fibonacci proposal has the following main features:

Objects with roles An object has an immutable identity and is organized as an acyclic graph of *roles*. Methods and fields are associated with the roles. Every message is addressed to a specific role of an object, and the answer may depend on the role addressed (*context dependent behaviors*);

Independence of extensions An object can be extended with unrelated subroles without interference;

Plurality of dynamic bindings A message can be sent to a role with two different notations to request a different lookup method:

- *upward lookup*: the message is sent with the exclamation mark notation, and the method is looked for in the receiving role and in its ancestors;
- *double lookup*: the message is sent with the dot notation, and the method is first looked for in all the descendants of the receiving role, visited in reverse temporal order, then in the receiving role, and finally in its ancestors.

Note that a traditional object oriented language can be seen as a role language where no object is ever extended and every message is always sent to the most specific role of the object. In this situation, upward lookup and double lookup coincide, and both coincide with the standard method lookup technique.

Role casting and role inspection Operators are provided to inspect the roles of an object and to dynamically change the role through which an object is accessed.

Multiple implementations An object type only describes the interface of the corresponding objects, while the implementation (i.e., method implementation and state structure) is defined, for every object, when the object is built.

We only describe here the Galileo 95 model, which adopts the single implementation approach for objects.
Let us consider again the definitions given above of the `Person` subtypes `Student` and `Athlete`:

```
let rec
type Student := object is Person and
                [Code :string;
                 Faculty :string;
                 Introduce := fun () :string is
                          implode({(self As Person)!Introduce();
                                   " I am a student of  ";
                                   self.Faculty}) ];
```

```
let rec type Athlete = object is Person and
        [Code:int;
         Sport: string;
         Introduce:= fun () :string is
            implode({(self As Person)!Introduce();
                     " I practice ";
                     self.Sport}) ];
```

`(self As Person)!Introduce()` invokes the `Introduce` method defined for `Person`; its semantics is detailed in the next section.

In this model, an object with a role `john` of type `Person` may now be extended with the types `Student` and `Athlete` as follows:

```
let johnAsStudent := inStudent(john, [...]);
let johnAsAthlete := inAthlete(john, [...]);
```

The answer to the message `Code` sent to `johnAsStudent` is a string while the answer to the same message sent to `johnAsAthlete` is an integer. The answers to the message `Introduce` sent to `johnAsStudent` or to `johnAsAthlete` are also different. We say that `john`, `johnAsStudent` and `johnAsAthlete` are three roles of the same object, of type `Person`, `Student`, and `Athlete`, respectively.

The `mkT`, `inT`, `dropT` and `Expr isalso T` operations are the same as in the previous section. `Obj As T` fails if `Obj` has no `T` role, and returns a reference to the `T` role of `Obj` otherwise. This is slightly different from the version described in the previous section, since in that case, if `Obj As T` is defined, then `Obj` and `Obj As T` only differ in their type, while here they refer to two different roles of the same object. Finally, an operation `Expr isexactly T` is added, which tests the run-time role type of the role `Expr`; for example, `john isexactly Athlete` is false while `johnAsAthlete isexactly Athlete` is true.

5.1 Method Determination

When a role `r` with run-time type `T` receives a double lookup message `r.m`, the corresponding method is looked for in two steps:

1. first, the method is looked for in the object roles whose type is a subtype of `T`, in the inverse acquisition time order, i.e. starting from the last acquired role and going backward (*downward lookup phase*);

2. if the method is not found, the search proceeds in the role type `T`, and finally goes up the supertype chain of `T` until the root type is reached (*upward lookup phase*). Static typechecking ensures that the search will eventually find the appropriate method.

When a role `r` receives an upward lookup message `r!m`, only step 2 is performed.

For example, the answer to the double lookup message `john.Introduce` changes once the object has been extended with the role type `Student`, and once again after its extension with the role type `Athlete`. To receive always the same answer from `john`, irrespective of any extensions, the message must be sent with the `john!Introduce` notation.

The combination of double lookup with role casting allows *static binding*, and the *super* mechanism, to be simulated. For example, let us consider the following function:

```
let foo := fun(x:Person) :{string} is
          {x.Introduce;
           x!Introduce;
           (x As Person)!Introduce}
```

Let `johnAsStudent` be bound to a value of type `Student`, which has been later extended with a role of type `ForeignStudent`, subtype of `Student` which redefines the method `Introduce`. The value returned by `foo(johnAsStudent)` is a sequence of three answers produced by the methods defined in type `ForeignStudent` (*double lookup*), in type `Student` (*upward lookup*), and in type `Person` (*static binding*).

5.2 Self-reference Semantics

When a method containing a `self.msg` invocation is executed, and the original message was sent to a role `r` with run time type `T`, the interpretation of `self` is determined as follows:

- if the method was found in the downward lookup phase, hence in a type `T′` which is a subtype of the type `T` of `r`, then `self` is bound to the `r As T′` role;

- if the method was found by a search in the supertype chain, then `self` is bound to the `r` role.

Hence, `self` behaves as if it were statically bound for methods found during the downward lookup phase, and dynamically bound for methods found during the upward lookup phase. In fact, if `self` were bound to the `T` role inside a method found in a type `T′` $\subseteq$ `T` during the history search phase, run-time type errors may arise, since, when the `T′` method has been compiled, the `self` type was assumed to be a subtype of `T′`, which is not true for `T`. On the other hand, this choice does not affect the language's expressive power because the method lookup mechanism is equivalent to the one adopted in the basic language for non extended objects. This means that this approach can represent every classical object-oriented construction based on the dynamic binding of self for non-extensible objects.

5.3 The Implementation Model

In this language, when an object is extended, the old methods and attributes are not deleted, since they can still be accessed using the `obj!m` notation. Moreover, if two attributes with the same name are added to two different subtypes of a common supertype, as in the `Code` example in the introduction, both attributes are present in an object which has been extended to belong to both subtypes. For this reason, for both methods and attributes, a search structure is needed which associates a method, or a position, to each triple "name, role, search technique", where search technique may either be "." (double lookup) or "!" (upward lookup). The simplest implementation is obtained by dealing with attributes as if they were methods, associating a different Local Attribute Table to each role, as shown in Figure 5.

Methods and attributes are then searched using the graph search algorithm (downward and upward) previously defined, and attributes can be accessed in the same way. Of course, two different attribute tables may assign two different positions to the same attribute, as happens with the `Code` attribute. In this situation, the object contains the temporal sequence of the acquired roles plus the object state.

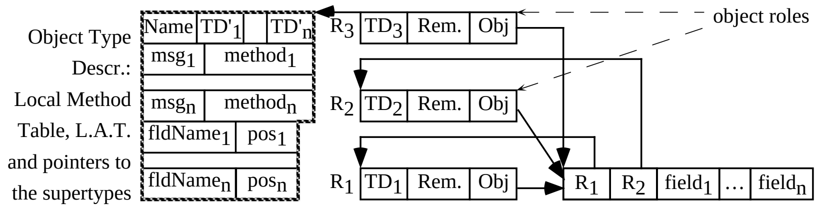

Figure 5: The structure of an object with a plurality of behaviors.

A new role is created when the object is created, and when the object is extended with a new type. Creating an object in a non-root type is implemented in the same way as creating the object in the root type and then extending it. Object extension is only valid if the object has all the supertypes of the new type but does not have the type is acquiring.

As in the other cases, a method is represented by a function which takes a “self” parameter which, in this case, is bound as defined in Section 5.2.

This representation may be easily optimized, as in the other cases, by flattening the method and attribute tables, and by sharing them, and the role history, among the objects with the same history. However, in this case every role needs two tables, one for double lookup, and the other one for upward lookup.

6 Conclusion

We have described a sequence of object models of increasing complexity, starting with the standard model and ending with a model with the following features:

- extensible objects;
- shrinkable objects;
- context dependent behavior (roles).

Drawing on our experience in the implementation of Galileo, Galileo 95, and Fibonacci, we have presented a sequence of implementation models of increasing complexity, to show which implementative features are linked to every linguistic feature, and to help to distinguish the basic run-time information, which is strictly needed to implement every operation, from the structures that are added to obtain a faster implementation of message passing.

We did not discuss how multiple implementations of a single type and how private attributes affect object representation; these issues will be studied in a more complete version of this paper.

7 Acknowledgements

This work has been supported in part by grants from the C.E.C. under ESPRIT BRA No.6309 (FIDE2: Fully Integrated Data Environment), the Progetto finalizzato “Sistemi informatici e calcolo parallelo” of C.N.R. under grant No. 92.001561.PF69, and by “Ministero dell’Università e della Ricerca Scientifica e Tecnologica”.

References

[AAG95] A. Albano, G. Antognoni, and G. Ghelli. View operations on objects with roles. Technical report, Università di Pisa, Dipartimento di informatica, 1995. (submitted).

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. of the Nineteenth Intl. Conf. on Very Large Data Bases (VLDB), Dublin, Ireland*, pages 39–51, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985. Also in S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[AGO91] A. Albano, G. Ghelli, and R. Orsini. Objects for a database programming language. In P. C. Kanellakis and J. W. Schmidt, editors, *Proc. of the Third Intl. Workshop on Data Base Programming Languages (DBPL), Nafplion, Greece*, pages 236–253, San Mateo, California, 1991. Morgan Kaufmann Publishers.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found also in *Semantics of Data Types*, LNCS 173, 51–67, Springer-Verlag, 1984.

[FBC+87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. D. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.

[RS91] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple, indipendent roles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 298–307, Denver, CO, 1991.

[SS89] J. J. Shilling and P. F. Sweeney. Three steps to view: Extending the object-oriented paradigm. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 10 of *ACM SIGPLAN Notices*, pages 353–361, 1989.

[SZ89] L. A. Stein and S. B. Zdonik. Clovers: The dynamic behavior of type and instances. Technical Report CS-89-42, Brown University Technical Report, 1989.