Extensible Objects for Database Evolution: Language Features and Implementation Issues

A. Albano, M. Diotallevi, and G. Ghelli

Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56125 Pisa – Italy, e-mail: albano@di.unipi.it, ghelli@di.unipi.it

Summary. 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. This is unacceptable, since it creates 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 codes, 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 [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.

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 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. shows how the model must be further modified to deal with the possibility of giving different implementations for values of the same object type, as happens in Fibonacci. Section 7. 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]. When the chosen language involves making choices which may have an implementative impact (e.g. single inheritance or single representation for a single object type), we also discuss the alternatives.

2.1 The language

2.1.1 Object Types. In Galileo 95 objects are modeled using the so-called "object as record" analogy, adopted initially in Simula, formalized by Cardelli [Car88], and used in most object database systems: objects are essentially records that may have functional components to model methods; message passing is implemented as field selection. In this context, 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 object fields in the object-as-record analogy, 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. The consequences of this different approach are discussed in Section 6..

An object type is specified by a set of pairs of two kinds:

1. A label-type pair (`A`$_i$ `:T`$_i$) represents one component of the object state (the identifiers `A`$_i$ are called *attributes*); the value associated with an attribute `A`$_i$ of an object `O` is extracted with the dot expression `O.A`$_i$.
2. A label-function pair (`A`$_i$ `:= fun(...) ...`) represents a method, i.e. a function shared by all instances of the object type; a method can access the attributes and methods of the object using the predefined identifier `self`. A message `A`$_i$ with parameters `p` is sent to an object `O` employing the `O.A`$_i$`(p)` notation.

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 })
         ];
```

Where `[House :string]` is an example of a tuple type, {`"a";"b"`} is a sequence of string, `implode` concatenates a sequence of strings, and the declaration `let type T = object [SoP]`, where `SoP` is a set of pairs as previously described, introduces into the current environment a new type `T` and the function `mkT` to construct values of type `T`. The input parameter of the `mkT` constructor is a record of type `[SoP']` with one field for each component in the object state, i.e. `SoP'` is the set of `SoP` label-type pairs. Each application of the `mkT` constructor returns an object of type `T` with a different identity.

An example of the construction of an object of type `Person`, and examples of state access and message passing are:

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

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

In general, object methods and attributes — called the *properties* of an object — may be either *public* or *private*. A private property is only accessible from within the type definition, while a public property may always be accessed. Hereafter, properties are assumed to be public, since this aspect is not relevant for the considerations we are interested in.

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 how its state and method set differ from T state and methods. For methodological and technical reasons, most object-oriented languages only allow strict inheritance, which means that T' can be defined from T only by:

- adding instance variables or methods to T;
- refining T's state and methods, where refining an instance variable means substituting its type with a subtype, and refining a method m means substituting it with a new method whose type is a subtype of the type of m.[1]

Whenever T' is defined by strict inheritance from T, T' is also a subtype of T, since T' supports the whole T interface.

Strict inheritance is also adopted in Galileo 95, where this is the only way to define a subtype of an object type. To define an object type `T` by inheritance from another object type `T'`, we write:

```
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;
```

[1] In any strongly and statically typed language the type $S' \to U'$ of m' is a subtype of the type $S \to U$ of m when $S \subseteq S'$ and $U' \subseteq U$; the inversion of the direction of the comparison between S and S' is explained in [Car88].

```
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` $\mathtt{T}_1$`, ...,` $\mathtt{T}_n$ `and` $\mathtt{T}_r$ (multiple inheritance). If the supertypes have a property with the same name and different types, the property is inherited from the last (w.r.t. the $\mathtt{T}_1$, ..., $\mathtt{T}_n$ order) supertype which defines it. In this case, strict inheritance means that every property 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 $\mathtt{T}_1$, ..., $\mathtt{T}_n$. Not every object oriented language allows multiple inheritance; inheritance from a single supertype is the most common solution.

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 type T' inheriting 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. We give a simple, not unrealistic, way of representing this information, without discussing alternative representations and optimizations.

In our model, every object is represented by a reference to a representation of its run-time type (the object type descriptor) plus a representation of the object's own state. The object type descriptor and the object state representation are described in the following subsections.

2.2.1 The Object Type Descriptor. We need to make a distinction between the *compile-time object type descriptor*, which is used by the compiler to determine the correctness of message passing operations and of new object type definitions, and the *run-time object type descriptor*, which is used for method lookup. Both structures are persistent in an object oriented database system, but the first belongs to the schema while the second is managed by the run-time system.

A run-time object type descriptor thus only contains the method and state lookup table for that type. This is a table which associates every message accepted by an object of that type with the code of the corresponding method (even if inherited), and every component of the state of the object with its position inside the object state representation. The second piece of information is needed since the language supports multiple inheritance; when single inheritance alone is supported, the offset of each state component can be directly computed by the compiler. It would also be possible, but more costly, to keep field names inside the object itself. We will also call this structure *extended* run-time object type descriptor, to emphasize that it contains both owned and inherited methods, while in most of the other implementative models we will present the run-time object type descriptor will only contain information about owned (not inherited) properties.

2.2.2 The Representation of the Object State. The state of each object is simply represented as a tuple containing its non-method fields. The run-time representation of an object is depicted in Figure 2.1.

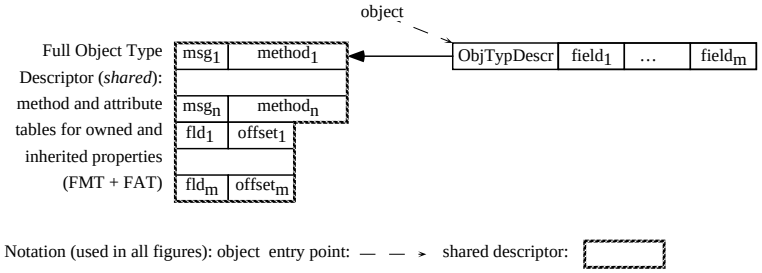

Fig. 2.1. The structure of an object which does not change type.

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 extend an instance of type `T` with the new type `T'`, without affecting the object identity.

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();
                                         " 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`. However, a `Student` can 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 `johnAs-Student 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 $\{T_1, \ldots, T_n\}$ such that 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 $\{T_1, \ldots, T_n\}$ receives a message `m`, the method to execute is searched for in two steps:

1. first, the method is looked for in methods that belong to (i.e. are not inherited) the last type T_n acquired; if it is not found there, the search

goes on in the type history, in the inverse temporal order T_{n-1}, T_{n-2}, ..., T_1;
2. then, if the method is not even found in the construction type T_1, the search goes up the supertype chain of T_1 as in the basic language. Static typechecking ensures that the search will eventually find the appropriate method.

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`.

3.2 Self-reference Semantics

When a method contains a `self.msg` invocation, the interpretation of the self-reference depends on how the method has been determined:

- if the method has been found in the type T_i by the search into the type history, the type of `self` is T_i, the type containing the selected method;
- if the method has been found by a search in the supertype chain, the type of `self` is T_1, the creation time type of the object that received the message `m`.

Hence, we can say that `self` is statically bound for methods found during the history search phase, while it is dynamically bound for methods found during the upward search phase. Performing a dynamic binding of `self` to T_1 for a method found in type T_i during the history search phase would not be sound since, when a message is compiled, `self` type it is assumed to be a subtype of the T_i type being defined, which is not true for T_1. 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.

As an example of the `self` interpretation rule, let us consider the following definitions:

```
let rec type  W1 := object [s := fun():int is 3;
                            r := fun():int is self.s() ];
let type      W11 := object is W1 and [s := fun():int is 4 ];
let rec type  W12 := object is W1 and [s := fun():int is 5;
                                       r := fun():int is 2*self.s() ];
```

Let us construct a value `v1` of type `W11`, and send it the message `v1.r()`:

```
let v1 := mkW11([ ]);
vi.r();                 returns 4
```

`vi.r()` returns 4 because the method for `r` is inherited from `W1`; here `self` is assigned type `W11` (dynamic binding), hence `self.s` returns 4.

Let us extend `v1` with the type `W12`, and send it again the message `v1.r()`:

```
let v2 := inW12(v1,[ ]);
v1.r();                  returns 10
```

This time the method for `r` is found in `W12` by history search, hence `self` is statically bound to type `W12`, hence `self.s` returns 2*5.

Note that there is no modification in the semantics of `super`.

3.3 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, and a modifiable sequence of label-value pairs to represent the state. In this case a type descriptor only contains the code for its own methods. Method search is executed through the two-phase algorithm described above, while field search is executed exploiting field labels. When an object is extended with a new subtype, the new type is added to its history and the new fields are added to its state; if an attribute of the supertype is redefined, its value is directly replaced by the new value.

More specifically, every method is represented as a function which receives, apart from the message parameter, a self parameter which is bound to the receiving object, as happens with objects that do not change type, plus a boolean parameter which says whether the method has been found during the history search phase or during the upward search phase. This boolean parameter is used to determine whether the history search must be used when a message is sent to `self`. This simple representation is shown in Figure 3.1. Note that the object is accessed indirectly to allow the object to be extended without modifying its identity, i.e. preserving any external reference to the object; any other technique to allow identity preserving extensibility (e.g., concatenating new field to the object tail) would work.

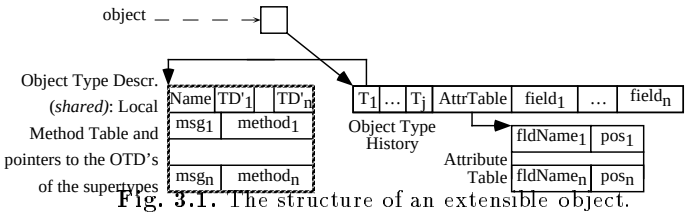

Fig. 3.1. The structure of an extensible object.

3.4 An Improved Implementation Model

To obtain a more efficient execution of message passing, an object representation can be used which closely resembles the one in Figure 2.1. In this case, each object only contains a reference to an *Object Type History Descriptor* (*OTHD*). An OTHD contains an object type history (i.e. a sequence of references to object type descriptors), a method lookup table and a field lookup table, which allow one to find the code of every method and the position of every field for any object with the story described by the OTHD. The system maintains a pool of OTHD's and creates a new one only when an object is created whose story is different from any story which is currently described in the OTHD. This optimized representation is shown in Figure 3.2.

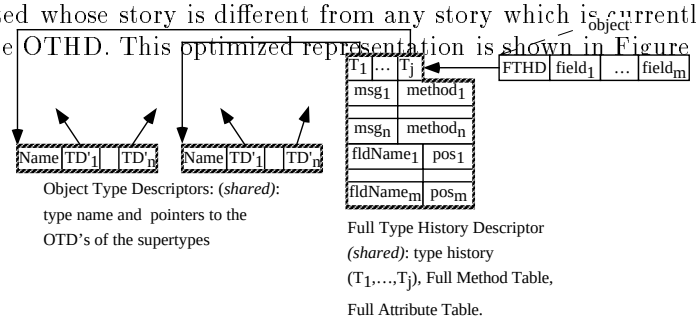

Fig. 3.2. A better structure for extensible objects.

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 happens with `mkT` and `inT`.

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

- if the object is accessed through the identifier $\texttt{Ide}_2$, a run-time failure will arise when a message is sent 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 $\mathtt{T}_2$ by going through $\mathtt{Ide}_1$.
- the `isalso` and `As` operators can still be applied to $\mathtt{Ide}_2$, 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 implementative 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 type (e.g., $\mathtt{Ide}_1$ is associated with the `Person` type while $\mathtt{Ide}_2$ is associated with the `Student` type), the validity (e.g., $\mathtt{Ide}_1$ is valid while, after the $\mathtt{dropT}_2$ operation, $\mathtt{Ide}_2$ is not valid any more), and a reference to the object. The object itself must contain 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 4.1, where an object with two valid roles and one removed role is represented. Note that the indirection level given by roles can also be exploited to allow identity preserving modifications of the object.

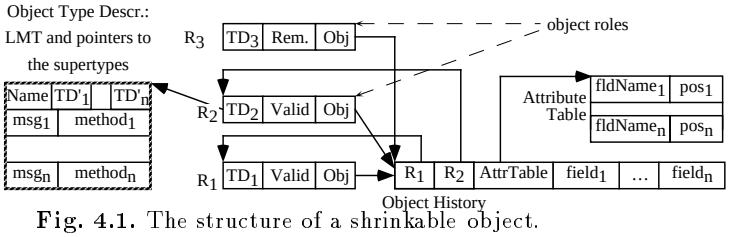

Fig. 4.1. The structure of a shrinkable object.

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

- the status of the $\mathtt{T}_i$ role becomes `removed`;
- the type $\mathtt{T}_i$ is removed from the object type history;
- the first steps are repeated for every role of the object whose type is a subtype of $\mathtt{T}_i$.

We do not discuss, in this case, any optimized implementation.

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 Fi-

bonacci [ABGO93], and then adapted to Galileo 95 [AAG95]. The proposal has the following main features:

Objects with roles An object has an immutable identity and is organized as acyclic graphs 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 will first describe 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}) ];
```

The semantics of the expression `(self As Person)!Introduce()` used in the definition of the method `Introduce` will be explained 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.

Besides the functions `mkT` and `inT'`, the following operators, similar to those seen in the previous section, are also provided on objects and roles:

- `dropT(Expr)`, to drop the role with type `T` and all its sub-roles from the object reachable through the role `Expr` (hereafter "the object `Expr`"). A run-time failure will arise if a message is sent to a removed role.
- `Expr isalso T`, to test whether an object `Expr` also has the role type `T`; for example `johnAsStudent isalso Athlete` is true.
- `Expr As T`, to retrieve the role `T` of the object `Expr`. For example, `johnAsStudent As Athlete` returns the role with type `Athlete` of the object which is reached through the role `johnAsStudent`.
- `Expr isexactly T`, to test the run-time role type of the role denoted by the expression `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_i 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_i, in the inverse acquisition time order;[2]
2. if the method is not found, the search proceeds in the role type T_i, and finally goes up the supertype chain of T_i until the root type is reached. Static typechecking ensures that the search will eventually find the appropriate method.

[2] Note the difference with the corresponding rule given in the previous section: there the method was looked for in the whole object type history, here it is looked for in the subroles of the receiving role only.

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(johnAs-Student)` 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 the method selected by a role `r` to answer a message `m` contains a message whose receiver is `self`, the interpretation of the self-reference depends on how the method has been found:

- if the method has been found in the downward lookup phase, hence in a type `T` which is a subtype of the type 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.

The observations made in Section 3.2 apply in this case too.

5.3 The Implementation Model

In this language, methods and attributes are not associated with the object as a whole, but each property is associated with one specific role. For this reason, the simplest implementation model is obtained starting from the last model presented and moving methods and state fields from the object, whose only aim is now to collect the history of the roles it acquired, to the roles, as shown in Figure 5.1. A role now stores the following information:

- a reference to the role type descriptor;

– a status `valid` or `removed` to describe whether the object lost that type;
– a reference to the object;
– the values of the owned attributes of its role type.

A new role descriptor 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.

The object descriptor is a sequence of references to the roles of the object, kept in temporal order.

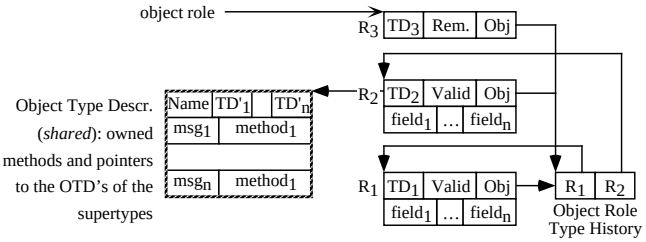

Fig. 5.1. The structure of an object with a plurality of behaviors.

This representation may easily be optimized, as will be shown in the next section.

6. Extensibility, Roles, and Multiple Implementations

The final feature of the language is separation between role type interfaces and implementations, as in Fibonacci, where:

– a role type only specifies the role interface but not the method implementation;
– a role implementation is only given when a role is constructed, or when an object is extended with new roles. Different role implementations can be given for the same role type;
– the state of a role is encapsulated and is accessible only through methods of the same role; this last point is not, however, essential for our discussion.

In this context, types `Person` and `Student` would be defined as follows:

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

```
let rec type Student = object is Persona and
                         [Code :string;
                          Faculty :string;
                          Phones :[House :string; GuestHouse :string];
                          Introduce :string ];
```

The implementation of methods is given when a role is constructed:

```
let john = role Person with
               private
                 MyName := "Antonio Rossi";
                 MyBirthYear := 1950;
                 MyPhone := [House :="565443"]
               method
                 Name:= MyName;
                 BirthYear := MyBirthYear;
                 Phones := {MyPhone};
                 Introduce:= implode({"My name is ";self.Name})
               end;
```

A student may be built either by extending a person, or from scratch, by defining the owned and inherited methods together; this second possibility affects the implementation model, as we will see.

Method determination and self-reference semantic rules are the same as in the previous section.

6.1 Implementation Model

The implementation model is shown in Figure 6.1. The main difference is that methods move from the role type descriptor to the role representation. Another difference arises because an object can acquire more than one role type in a shot, both when it is created and when it is extended, hence it is not true that an object has as many role as types. For example, suppose that four types $\mathtt{T}_4 \subseteq \mathtt{T}_3 \subseteq \mathtt{T}_2 \subseteq \mathtt{T}_1$ are defined. An object may be created in $\mathtt{T}_2$ and then extended in type $\mathtt{T}_4$. In this case the object would only have two roles, the first one returned by both `obj As` $\mathtt{T}_1$ and `obj As` $\mathtt{T}_2$, and the second one returned by both `obj As` $\mathtt{T}_3$ and `obj As` $\mathtt{T}_4$.[3]

The *role type descriptor* (*RTD*) now contains the following information:

- the type name (actually, its run-time representation);
- the list of all the supertypes of the type, in inheritance order, i.e. it $\mathtt{T}_1, \ldots, \mathtt{T}_n$ is the list of immediate supertypes of `T`. The list of all the supertypes of `T` contains first the list of all supertypes of $\mathtt{T}_n$, then the list of all the supertypes of $\mathtt{T}_{n-1}$ which are not supertypes of $\mathtt{T}_n$, and so on.
- the list of all the subtypes of the type.

[3] Note that this feature is orthogonal to multiple implementation; we might have introduced it in the previous section too.

We may only keep immediate supertypes in the RTD, as in the previous sections. However, the increased complexity of lookup method justifies the decision of making this small optimization from the beginning.

The *role representation* contains the following information:

- a reference to the most specialized type associated with the role;
- a `valid` or `removed` status;
- the role local state, shared by all role's own methods;
- a table containing names and implementations for every owned method of the role;
- a reference to the object descriptor.

Finally, the object descriptor is a sequence, kept in temporal order, of pairs (types, role). Whenever an object acquires a new set *types* of types, thanks to an extension operation, a new pair *types,role* is added to the sequence, where *role* is the newly acquired role. Associating the set of types with each role in the object descriptor is a small optimization, which helps to implement message passing and the `As` operation.

Double lookup message passing is implemented as follows:

1. Downward lookup: if the object is accessed through a role $\mathtt{R}_i$ whose most specific type is $\mathtt{T}_i$, the subtype list is obtained from the role type descriptor of type $\mathtt{T}_i$, and the method is searched in every role `R` in the object role type history such that the most specific type of `R` is a subtype of $\mathtt{T}_i$. The role type history is looked for in reverse temporal order.
2. Upward lookup: if downward lookup fails, then the supertype list is obtained from the role type descriptor of type $\mathtt{T}_i$, and the method is searched in every role `R` in the object role type history such that at least one type of `R` is a supertype of $\mathtt{T}_i$. Roles are examined in the inheritance order, i.e. according to the order of types in the supertype list.

Figure 6.1 below represents a situation where types $\mathtt{T}_2$ and $\mathtt{T}_3$ are two subtypes of type $\mathtt{T}_1$, and the represented object has first been built from scratch in type $\mathtt{T}_2$, and then has been extended to type $\mathtt{T}_3$.

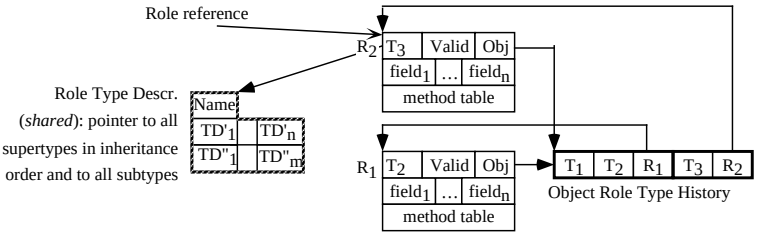

Fig. 6.1. The structure of an object with a plurality of behaviors and a separate definition of interface and implementation.

Note that no name is needed in the state representation, since it is only accessed by local methods, hence the position of each state component can be statically determined. On the other hand, the method lookup algorithm takes advantage of the fact that the method table associates names with method implementation.

The semantics of self is the same as in the previous section, and can be fully supported once the `As` operator is supported.

6.2 An Improved Implementation Model

The solution described so far is not realistic, since method lookup, which is the basic operation, requires an algorithm which is too slow. Hence, in the development of the Fibonacci system an optimization similar to the one described in Section 3.4 was adopted.

In Fibonacci, every role is associated with a full method lookup table, i.e. a table which associates with each owned or inherited message, that the role can receive, the following information:

- a reference to the method code;
- a reference (`owner`) to the role *type* where the method implementation belongs;
- a field (`overriden`) to remember whether the method is defined in a subrole;

More precisely, each role contains a reference to two method lookup tables, one corresponding to double lookup and the second corresponding to upward lookup only; the "overridden" field is only present in the double lookup table.

The two lookup tables are built when a role is built, and in that moment they coincide. Later, whenever a new subrole `subr` defining a method `meth` for `msg` is added to the object, the `msg` field in the double lookup table of all superroles of `subr` is updated with the triple (meth, type of subr, true). The upward lookup table never changes. A method is a function which receives all the message parameters plus two roles, the "state role" which must be used to retrieve state variables and the "self role" which is bound to `self`.

To answer a message `msg` a role `r` just looks in the appropriate table to find the correct method `meth` and its owner type `owner`. Then, if overridden is `false` or is not applicable (upward lookup), it calls `meth` binding "state role" to `r As owner` and "self role" to `r`. If overridden is `true`, then `r As owner` must be bound to "self role" too.

Note that the lookup table contains the type of the role that owns the implementation, rather than the role itself, so that two different roles with the same message-method correspondence may share the table. Actually, the Fibonacci implementation contains a *lookup table pool* where every new lookup table is stored, associated with the object extension history. In this context an object extension history is not just a sequence of types since two objects may acquire the same type but with a different implementation. The history

is a sequence of extension functions. This is because usually there are very few extension functions for every type, just one for most of them, and since almost every object construction or extension is made through a function, indexing the table pool with function histories is almost as effective as using type history in languages with one implementation per type.

The resulting situation is shown in Figure 6.2.

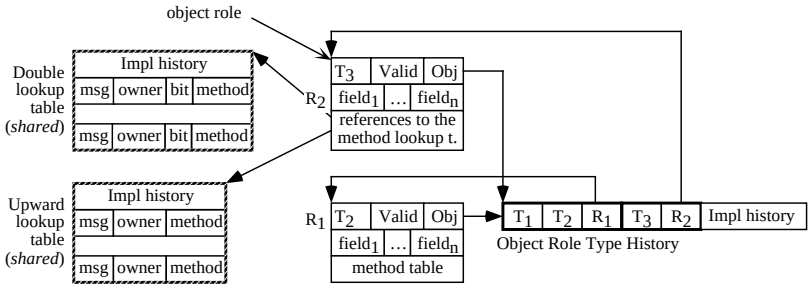

Fig. 6.2. The Fibonacci optimized implementation of roles.

7. 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;
- context dependent behavior (roles);
- shrinkable objects;
- multiple implementations for the same type.

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.

Acknowledgement. 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. 93.001502.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.

[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.