View Operations on Objects with Roles for a Statically Typed Database Language: The Type Rules and the Operational Semantics

ANTONIO ALBANO, GIUSEPPE ANTOGNONI, and GIORGIO GHELLI
Dipartimento di Informatica, Università di Pisa, Italy

To deal with the evolution of data and applications, and with the existence of multiple views for the same data, the object data model needs to be extended with two different sets of operations: object extension operations, to allow an object to dynamically change its type, and object viewing operations, to allow an object to be seen as if it had a different structure. Object extension and object viewing operations are related, in that they are both identity-preserving operations; but different, in that object extension may modify the behavior of the original object while object viewing creates a new view for the original object without modifying its behavior. In this paper a set of object viewing operations is defined in the context of a statically and strongly typed database programming language, which supports objects with roles, and the relationships with object extension and role mechanisms is discussed. We then show how the object viewing operations can be used to give the semantics of a higher level mechanism to define views for object databases. Examples of the use of these operations are given with reference to the prototype implementation of the language Galileo97. Finally, the formalization is included of both the type rules and the operational semantics of the language mechanisms presented.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*; H.2.3 [**Database Management**]: Languages

General terms: Database Programming Languages

Additional Key Words and Phrases: object databases, object data models, objects with roles, object views, database programming languages

1. INTRODUCTION

The success of the object data model is mainly due to its expressivity and its ability to deal with both structural and procedural aspects of real-world entities. The technology of object databases (ODBs) is, however, still relatively new. It still has some limitations, in particular with respect to the possibility of adapting a database to the evolution of application requirements. It is widely recognized that long lasting, evolving databases need support for:

—objects that may evolve over time, and may exibit a different behavior in different contexts;

—views mechanisms to adapt the database schema to the changing needs of users;

—schema evolution, data evolution and versioning mechanisms to support changing application requirements.

In this paper we consider a set of operators on objects, in the context of a statically and strongly typed database programming language, to support the first two aspects of database evolution:

Author's address: Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56125 Pisa, Italy. E_mail: {albano, ghelli}@di.unipi.it This work has been supported in part by grants from the C.E.C. under ESPRIT BRA No.6309 (FIDE2: Fully Integrated Data Environment), by PASTEL Esprit Working Group 22552, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica" (MURST), Progetto INTERDATA.

—to allow objects to acquire and lose types during their life, to model the dynamic and context dependent behavior of real-world entities. For example, to model situations such as that of a human being who is initially classified as a person, then becomes a student, and finally an employee. When an object acquires a new type it preserves its identity, but it may change its behavior. The definition of a type-safe extension operator implies, as will be shown later, that objects are enriched with a *role* notion. In our approach, this means that every object is equipped with a set of roles which can change over time; a message is sent to an object only through one if its roles, and the answer to the message may depend on the role the message is addressed to, and on the roles possessed by the object.

—to allow the creation of *virtual objects* starting from base or virtual objects using a set of object algebra operators. A virtual object is used to model a different interface to the same conceptual entity, and so it has the same identity as the object from which it has been constructed, but it may have a different structure and behavior which, in contrast to role acquisition, does not affect the behavior of the object from which it has been constructed. We will also show how the object algebra operators can be used to give the semantics of a higher level mechanism to define views for ODBs, i.e. query-defined collections of virtual objects.

As will be shown later, objects with roles and virtual objects are two related mechanisms, hence it is essential to compare them in order to understand their similarities and differences. Objects with roles and virtual objects have been implemented in the original Galileo97 language, a statically and strongly typed language, thus proving that flexibility can be achieved without compromising the well known benefits of static typechecking. Galileo97 is the result of a redesign of the Galileo language [Albano et al. 1985], and is aimed at a better integration of an object mechanism into a database programming language. The role mechanism in Galileo97 is based on the one proposed for Fibonacci [Albano et al. 1993], [Albano et al. 1995]. The role mechanism is thus not a novel contribution of this paper, but is presented here for several reasons: (a) to compare it with the virtual objects mechanism proposed, (b) to show their different semantics and their interactions, and (c) to discuss why both mechanisms are needed to support database evolution at both instance and schema levels.

The main contributions of this paper are:

—the definition of an identity-preserving object algebra which allows virtual objects to be defined and typed; the novelty of our approach is that our operators are defined in the context of a statically strongly typed language, where data and code are mixed, and we define a type system to deal with both of them (Section 4 and Appendix A);

—the formalization of both the type rules and the operational semantics of the mechanisms that we present (Appendixes A and B);

—the definition of a translation from a view mechanism (i.e. a mechanism to define virtual *classes*) to our virtual *object* mechanisms (Section 5). Most work in this field focusses on one aspect only, and we have never found a formal definition of the relationship between the two mechanisms.

Objects with roles have recently been studied by several authors (e.g., [Bertino and Guerrini 1995], [Gottlob et al. 1996], [Odberg 1994], [Papazoglou and Krämer 1997], [Richardson and Schwartz 1991], [Shilling and Sweeney 1989], [Wieringa et al. 1995]); a comparison of these proposals is beyond the scope of this paper but can be found in [Papazoglou and Krämer 1997].

View mechanisms for ODBs have also been proposed by several authors (e.g., [Dayal 1989], [Bancilhon et al. 1990], [Bertino 1992], [Abiteboul and Bonner 1991], [Santos et al. 1994], [Guerrini et al. 1997], [Kim and Kelley 1995], [Ohori and Tajima 1994], [Parent and Spaccapietra 1985], [Rundensteiner 1992], [Scholl and Scheck 1990], [Scholl et al. 1994], [Heiler and Zdonik 1990], [Zdonik 1990],[Leung et al. 1993]), and the main approaches are compared in the survey paper [Motschnig-Pitrik 1996], together with a discussion of the functionalities needed by a general mechanism for defining views in ODBs. A deep comparison between our and those approaches will be presented once our approach has been defined, in Section 6. We only mention here that most of these approaches focus on the virtual class mechanism, while we study here virtual object operators, as done in [Ohori and Tajima 1994]; we also study the relationship between the two approaches (Section 5), while this aspect is not dealt with in other works.

The paper is organized as follows. Section 2 defines some basic terminology and introduces the concepts that will be used to explain the goals of the paper. Section 3 briefly presents the object with roles mechanism and its semantics. Section 4 describes the virtual objects mechanism and its semantics. Section 5 outlines the mechanism for defining views, whose semantics is given in terms of the virtual objects operators. Section 6 compares related works. In Section 7 some conclusions are drawn. Appendix A presents the type rules for the Galileo97 subset used in the paper. Appendix B presents a formal operational semantics for most of the features we presented; object with roles and virtual objects are fully described in this definition.

2. CONCEPTS, TERMINOLOGY AND ISSUES

This section establishes some basic terminology and describes informally the notion of objects with roles and virtual objects. We will also introduce the essentials of the Galileo97 language syntax that will be used to give examples in the rest of the paper to make the presentation more concrete.

2.1 Objects and Types

An *object* is a software entity which has an internal state (*instance variables*) equipped with a setof local operations (*methods*) to manipulate that state. The request for an object to execute an operation is called a *message* to which the object can reply. When the object state is not visible directly, but can only be accessed through the methods, the object encapsulates its state. The object state and methods are also called the object structural and behavioral aspects.

An object is an instance of a type defined with a *generative type constuctor*, i.e. each object type definition produces a new type, which is different from any other previously defined types. An object type describes the state fields and the implementation of methods of its possible instances. An object type definition introduces a constructor of its instances, and so an object can be constructed only after its object type definition has been given.

The signature $\Downarrow\mathcal{T}$ of an object type $\mathcal{T}$ is the set of label-type pairs of the messages which can be sent to its instances.

Each application of an object constructor returns a new object with a different *identity* that persists over time, independently of changes to its state. The equality operator on objects is based on identity. Object identity may be implemented as a hidden field with a system-generated and system-wide unique *object-identifier* (OID). Hereafter we will not talk about OID, but we will assume that (a) objects are first class values, and therefore can be embedded in data structures, passed as a parameter and returned as a value, and (b) that equality on objects is sameness

(OID equality).

In the object-oriented programming context this approach to objects is called *class-based* since the description of objects is called a *class*; we prefer the term "type" since we will use "class" with a different meaning according to the database tradition.

Objects and Types in Galileo97

In Galileo97 a new object type is defined by the operator `<->`[1]. An object type description is defined using the record constructor `[]` and a set of label-type and label-method pairs. Each label is called a "field" or "attribute". A label-type pair, such as `Ide: string`, represents an instance variable (a *state component*), while a label-method pair, such as `Ide:=meth ...`, represents a method. Methods can access the fields of an object using the predefined identifier `self`. Messages are sent to objects using the dot notation.

Components of the state can be declared to be updatable (e.g. `Ide:= var T`); they are updated in an object `O` with the operator `<-` (e.g. `O.Ide <- v`), and their value can be obtained with the operator `at` (e.g. `at O.Ide`). The distinction of constant and updatable attributes in the Galileo97 type system is an important feature to ensure static typechecking.

Components of the state or methods can be defined as `private` to enforce encapsulation, otherwise they are considered `public`, however for the sake of simplicity we will not consider this issue, which is orthogonal to our main subject.

The following is an example of the object type `Person` definition:

```
let rec type Person <->
    [Name: string;
     BirthYear: int;
     Address: var string;
     WhoAreYou:= meth(): string is "My name is  " & self.Name & "." ];
```

The definition of an object type $\mathcal{T}$ introduces the function `mk`$\mathcal{T}$ to construct objects of type $\mathcal{T}$: the function parameter is a record type contaning the label-type pairs of $\mathcal{T}$. For example, the following expression binds the identifier `John` to an object of type `Person`:

```
let John := mkPerson ([Name := "John Smith";
                       Address := var "A street";
                       BirthYear := 1967 ]);
```

2.2 Inheritance

Inheritance is a mechanism which allows something to be defined, typically an object type, by only describing how it differs from a previously defined one. Inheritance should not be confused with subtyping: subtyping is a relation between types such that when $\mathcal{T} \leq \mathcal{S}$, then any operation which can be applied to any value of type $\mathcal{S}$ can also be applied to any value of type $\mathcal{T}$. The two notions are sometimes confused because, in object oriented languages, inheritance is generally only used to define object subtypes, and object subtypes can only be defined by inheritance. However, we will keep the two terms distinct and will use each of them with its proper meaning.

[1] Galileo97 syntax is not very different from those of other object oriented database languages, and the query language is very similar to the ODMG-OQL syntax.

2.3 Inheritance in Galileo 97

In Galileo 97 an object type $\mathcal{T}$ can be defined by inheritance from another object type $\mathcal{T}'$ as follows:

`type` $\mathcal{T}$ `<->` `is` $\mathcal{T}'$ `and` $\mathcal{H}$

The type $\mathcal{T}$ *inherits* the $\mathcal{T}'$ attributes, i.e. both its instance variables and methods. Galileo 97 allows *strict* inheritance only: a $\mathcal{T}'$ attribute $\mathcal{A}_i$, with type $\mathcal{H}_i$, may be redefined in $\mathcal{T}$ only by specializing its type, that is the new type $\mathcal{H}'_i$ of $\mathcal{A}_i$ must be a subtype of $\mathcal{H}_i$. For this reason, the inheritance mechanism in Galileo 97 always produces an object subtype, i.e. in our example we will have $\mathcal{T} \leq \mathcal{T}'$.

The following is an example of an object type defined by inheritance:

```
let rec type Student <-> is Person and
    [Code: string;
     Faculty: string;
     WhoAreYou := meth(): string is
                  super.WhoAreYou  &
                  " I am a student of " & self.Faculty ];
```

In an object type defined by inheritance, the special identifer `super` in a new method, such as `WhoAreYou`, is used to invoke the old version of a method `m` from a direct supertype. As usual, any occurrence of `self` within the method `m` is interpreted with respect to the current subtype, and not with respect to the supertype.

Multiple inheritance is generally possible, i.e. inheritance from several supertypes, however for the sake of simplicity we will not consider this issue.

2.4 Static and Dynamic Dispatch

Given a method invocation of the form `O.m(...)`, a language dependent technique is responsible for identifying the appropriate method `m` of the object `O` that has to be executed. Let us consider the following function:

```
let print := fun(x: Person):string is x.WhoAreYou;
```

Let `John` be an object of type `Person` and `Bob` an object of type `Student`. The problem is the meaning of `x.WhoAreYou` in the body of `print` during the invocation of `print(Bob)`. According to a *static dispatch* technique (also called *early binding*), based on compile-time information about `x`, the code of `WhoAreYou` in `Person` is executed. On the other hand, according to a *dynamic dispatch* technique, based on the run-time information about `x`, the code of `WhoAreYou` in `Student` is executed. Dynamic dispatch (also called *late binding*), is found in all object oriented languages, and it is considered to be one of their defining properties.

2.5 Sequences, Classes and Subclasses

An object data model supports a mechanism to define a collection of homogeneous values to model multivalued attributes or collections of objects to model databases. Usually two different mechanisms are provided.

To model multivalued attributes, type constructors are available for bags, lists (or sequences), and sets. For the sake of simplicity we will only consider sequences.

To model databases we consider a mechanism called *class*. A class is a modifiable sequence of objects with the same type. A class definition has two different effects:

—it introduces the definition of the type $\mathcal{T}$ of its elements and a constructor for values of this type (*intensional aspect*),

—it supplies a name to denote the modifiable sequence of the elements of type $\mathcal{T}$ currently in the database (*extensional aspect*).

In Galileo97 classes and sequences can be queried using the same operators, similar to those offered by the language OQL. However, classes and sequences differ in that classes are automatically updated every time an object of the corresponding type is created or deleted, while the extent of a sequence is immutable. In addition, classes only can be defined by inheritance and organized into a *subclass* hierarchy, such that if $\mathcal{C}_1$ is a subclass of $\mathcal{C}_2$, then the following properties hold:

—the type of the elements in $\mathcal{C}_1$ is defined by inheritance from the type of the elements in $\mathcal{C}_2$;

—the elements in $\mathcal{C}_1$ are a subset of the elements in $\mathcal{C}_2$.

Subtype, inheritance, and subset are three different kinds of relations between types and values of an object language. *Subtype* is a relation between types which implies value substitutability; *inheritance* is a relation between definitions, which means that the inheriting definition is specified "by difference" with respect to the super-definition; *subset* is a subset relation between collections of objects, which also implies a subtype relation between the types of their elements. Languages exist that support only subtypes, or subtypes and inheritance, or subtypes, inheritance and subsets.

Sequences, Classes and Subclasses in Galileo 97

Sequences are constant collections of homogeneous values of any type. `seq` $\mathcal{T}$ is the type of sequence of elements of type $\mathcal{T}$. Sequences are enclosed in curly brackets and their elements are separated by semicolons.

A class definition introduces a name for the class and one for the object type of its elements. In the following example, the class `Persons` is defined, whose members belong to the object type `Person`:

```
let rec Persons class Person <->
    [Name: string;
     BirthYear: int;
     WhoAreYou:= meth(): string is "My name is  " & self.Name & "."];
```

Classes of objects model sets of entities of the observed world, while relationships between such entities are represented as objects that have other objects as components. When an object with the type $\mathcal{T}$ of the elements of a class is constructed by `mk`$\mathcal{T}$ or `in`$\mathcal{T}$, then the object automatically becomes an element in that class. When an object loses the type $\mathcal{T}$ (`drop`$\mathcal{T}$), then it is also removed from the corresponding class. Here is an example of a subclass definition:

```
let rec Students subset of Persons class
Student <-> is Person and
    [Code: string;
     Faculty: string;
     WhoAreYou := meth(): string is
                  super.WhoAreYou &
                  " I am a student of " & self.Faculty ];
```

2.6 Objects with roles

We call *object extension* the operation which allows an object to assume a new type without changing its identity. This operation is necessary to model the behavior of real world entities. It is also useful in the context of database evolution: when a new subtype is added to an object type hierarchy, it is often useful to make some existing objects acquire the new type.

With the object mechanism described so far, object extension is not allowed. Moreover, an object is always an instance of a single *minimal type*, that is a type $\mathcal{T}$

such that all the other types to which it belongs are supertypes of $\mathcal{T}$. The minimal type of an object is the one that it receives at construction time, and is the one which dictates which method the object uses to answer a message.

When object extension is allowed, it becomes possible for an object to acquire several minimal types, with possibly conflicting state component and method definitions (see Section 3 for an example). This problem should not be solved by having one minimal type which prevails on the other ones, since in general every context where an object has been extended with a minimal type expects that minimal type to be the prevailing one. For this reason, several authors have proposed to adopt a notion of object with *role* to support objects that can evolve over time by changing type and appearance. In this proposals, an *object role* (or simply a *role*) is one of the perspectives of an object as an instance of a type, and it defines a particular context for method execution. One object may possess one different role for each of its minimal types, or even one for each of its types (as in Galileo97). In any case, whenever an object is extended with a new minimal subtype it acquires a new role. Note that in this approach a new type is only acquired when a new role is acquired and viceversa, hence we do not have two distinct 'type acquisition' and 'role acquisition' operations.

The notion of objects with roles is essential to support object extension, but is also useful to model situations where one real world entity may exhibit a different behavior in different contexts. An analysis of these approaches can be found in [Papazoglou and Krämer 1997]. Three important issues in the design of a language that supports objects with roles are the technique to choose the role which will answer a specific message, the method lookup algorithm to adopt, and the semantics of the operator to drop types from an object. They will be discussed in Section 3 in the context of Galileo97.

2.7 Views for ODBs

Relational database systems provide a well-known view mechanism to define derived (or computed) tables as the result of a query on other real (or derived) tables. These derived tables can be queried as if they were real tables, but updating a derived table is generally not allowed, since it may not be possible to map the operation onto the real tables.

A similar functionality may be provided for object databases. For example, the following Galileo97 expression defines a sequence containing all the young persons currently present in the `Person` class:

```
let AllYoungPersons := derived  Persons where BirthYear > 1985;
```

`AllYoungPersons` is a computed value of type `seq Person`. A `derived` sequence (a) is computed every time it is used, (b) it can be queried as with any other sequence or class, (c) its elements can be updated in the same way as objects of a class can be updated and the effects are the same, and (d) its elements change if objects are added or removed from the class `Persons`, as one would expect. This is the only way to change the number of elements of a derived sequence.

The `AllYoungPersons` example shows that a derived sequence is adequate for representing a view whose elements are a subset of a class, while problems arise when one needs to change the structure of the class elements in the view. The only way to achieve the effect with the Galileo97 operators seen so far is to build a new value starting from a class element: it may be either a record (using the record constructor []) or a new object which does not have the same identity as the original one. Using the terminology proposed in [Scholl and Schek 1991], the first kind of view is called a *relational semantics view*, and the second an *object-*

generating semantics view. For example:

```
let ItalianRecordPersons := derived
       select [ Nome:= Name; AnnoNascita:= BirthYear]
       from Persons;

let type Italian <-> [ Nome: string; AnnoNascita: int];

let ItalianObjectPersons := derived
       select mkItalian([Nome:= Name; AnnoNascita:= BirthYear])
       from Persons;
```

However, two things are lacking:

—the ability to define *object-preserving semantics views*, i.e. views whose elements are virtual objects with the same identity as the corresponding base object;

—the ability to place the defined view into a subset hierarchy.

To solve the first problem, an object language should have an identity preserving algebra on objects which allows one to define different interfaces for the same objects (called *virtual objects*), so that different users may see the same objects with a different structure and behavior [Scholl and Schek 1991].

To solve the second problem, the language should have a *virtual class* mechanism, to allow the programmer to define collections which are: (a) virtual, i.e. computed starting from a base collection, as in our examples above; (b) classes, i.e. which are placed into the subset hierarchy and which collect all existing values of their associated type.

Object algebras have been proposed by several authors (e.g., [Shaw and Zdonik 1989] [Rundensteiner 1992] [Scholl et al. 1990] [Guerrini et al. 1997] [Leung et al. 1993] [Parent and Spaccapietra 1985]), and the basic algebraic operators are *selection* to define subsets of a class, *projection* to see a subset of the object attributes or methods, and *extension* to add new attributes and methods. Some of these papers combine the virtual objects and virtual classes mechanisms, while in this paper they are studied separately, and we show how the virtual objects operators can be used to give the semantics of a virtual class mechanism.

There are particular problems which must be addressed in order to include a set of object algebraic operators into a statically and strongly typed object-oriented programming language:

—what is the type of a virtual object, and how is it related to the type of the object it is based on?

—When the mechanism allows new methods to be added to objects, can the implementation of these methods have access to the state of the base objects? If a method can be overriden in the virtual object, what impact is there on the method lookup algorithm?

—What are the differences between the role mechanism and the virtual object mechanism?

—If a join operator is supported by the object algebra to combine two objects, what is the identity of the resulting objects?

A solution to these problems will be given in the context of Galileo97 in Section 4, and a comparison with other proposals will be presented later in Section 6.

3. OBJECTS WITH ROLES IN GALILEO 97

In Galileo97, besides the operator $\texttt{mk}\mathcal{T}$ to construct objects of type $\mathcal{T}$, the operator $\texttt{in}S$ exists to extend dynamically an object with a new subtype S of $\mathcal{T}$, without

changing its identity, but with the possibility of changing its behavior. The operator `in`S adds a new *role* to the object, and returns a reference to this new role of that object. In Galileo 97 an object expressions does not return an object alone, but always one specific role for that object.

The following example shows the definition of the type `Person`, with a method `WhoAreYou`, and two subtypes `Student` and `Athlete` defined by inheritance that have a `code` field with a different type. Such a definition is allowed in any object-oriented language, since the validity of an object type definition only depends on its object supertypes, but cannot be limited by the definition of its *cousin* (i.e. neither descendents nor ancestors) types:

```
let rec type Person <->
    [Name: string;
     BirthYear: int;
     WhoAreYou:= meth(): string is "My name is  " & self.Name & "."];

let rec type Student <-> is Person and
    [Code: string;
     Faculty: string;
     WhoAreYou := meth(): string is
                  super.WhoAreYou  & " I am a " & self.Faculty & "student" ];

let rec type Athlete <-> is Person and
    [Code: int;
     Sport: string;
     WhoAreYou:= meth(): string is
                 super.WhoAreYou & " I play " & self.Sport ];
```

The following expression builds an object, with one role only, of type `Person`:

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

The answer to the message `John.WhoAreYou` is `"My name is John Smith"`. The object with role `John` can now be extended with the subtype `Athlete` as follows:

```
let JohnAsAthlete := inAthlete(John, [ Code := 245; Sport := "tennis" ]);
```

As a consequence of this extension, we now have two different ways to access the same object. `John` refers to the `Person` role while `JohnAsAthlete` refers to the `Athlete` role of the same object. As a consequence, method lookup generally gives different results, depending on the role used.

As a consequence of this extension, `John` now has two different methods to answer the `WhoAreYou` message, and Galileo 97 allows both of them to be accessed, using the different notations `John.WhoAreYou` and `John!WhoAreYou`.

In both cases method lookup starts from the `Person` role of `John`. With the "." notation, the method is first looked for in the subtypes of `Person` (*downward lookup* phase). If this phase fails, the method is looked for in `Person` and in its supertypes (*upward lookup* phase). This whole process is called *double lookup*, and finds the `Athlete` implementation of `WhoAreYou`. With the "!" notation, only upward lookup is performed, thus finding the `Person` implementation of `WhoAreYou`. If the method is found in a supertype, `self` stands for the role that receives the message, otherwise `self` stands for the role that answers the message.

Both techniques are statically guaranteed to find a method of the right type, and both are instances of the late binding mechanism, since they do not depend on the static type of the receiver, but on its dynamic types and on the role through which it is accessed (this is not evident in this case, since the `Person` role of `John` corresponds to its static type; this is not the case in our next example).

Another, more important, consequence of the extension is that now we have two different roles to access the same object, which behave differently. For example if the `WhoAreYou` message is sent with the upward lookup notation, the two roles `John` and `JohnAsAthlete` answer in two different ways.

The object with role `John` can also be extended with the type `Student`:

```
let JohnAsStudent := inStudent(John, [ Code := "0123"; Faculty := "Science" ]);
```

We say that `John`, `JohnAsStudent` and `JohnAsAthlete` are three different roles of the same object, of type `Person`, `Student` and `Athlete`, respectively.

This extension has the following effects that show how in Galileo97 messages can be addressed to every role of an object, and the answer may depend on the role addressed:

—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 answer to the message `WhoAreYou` sent to `JohnAsStudent` is `"My name is John Smith.  I am a Science student"`, while the answer to the same message sent to `JohnAsAthlete` is `"My name is John Smith.  I play tennis"`;

—the answer to the message `WhoAreYou` sent to `John` with the dot notation changes and becomes `"My name is John Smith.  I am a Science student"`.

While upward and double lookup are two different forms of dynamic binding, static binding to the method of type $\mathcal{T}$ can be obtained through the `(O As T)!msg` idiom, where `O As T` is the operator which allows one to access the role with type $\mathcal{T}$ of an object `O`. Let us consider the following function:

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

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

The language also provides other operators on objects and roles which allow one to discover which roles can be played by an object (*RoleExpr* `isalso` $\mathcal{T}$), and to remove the role with type $\mathcal{T}$ and its subroles from an object (`drop`$\mathcal{T}$`(`*ObjExpr*`)`). If a role of type $\mathcal{T}$ is an element of a class, when an object loses the type $\mathcal{T}$ (`drop`$\mathcal{T}$), then it is also removed from the corresponding class. The semantics of the `drop` operator must be defined carefully, and it is discussed in the next section.

3.1 The Semantics of the drop Operator

To understand the following discussion, a clear distinction must be made between the static and run-time type of a role.

A run-time type of a role value is the type assigned to it when the role has been acquired. A run-time type of a role expression is the run-time type of the role value denoted by that expression. The same expression may have a different run time type whenever it is evaluated.

A static time type of a role expression is the type assigned to the expression at compile time.

To clarify this point, let us consider the following example:

```
let print := fun (x: Person):string is x.WhoAreYou;

print(John);
print(JohnAsStudent);
```

The compile type of the parameter `x` is `Person`, while its run-time type is `Person` when print is applied to `JohnAsPerson`, and `Student` when print is applied to `JohnAsStudent`. Note that the method lookup only depends on the run-time type of the receiver, which is always a subtype of its compile time type.

A language with the possibility of extending objects dynamically with new types should also provide an operator to drop types from an object (e.g. a person becomes a student and once graduated he is no longer a student). This possibility is supported by the Galileo 97 language with the operator `dropT`. Two different semantics can be given to the drop operator with different consequences both on how an application is modeled and on the complexity of the operator's implementation:

(1) A first solution is to assume that when an object `O` loses a type `T`, an access to `O` through the role with run-time type `T` will cause a run-time failure. This solution is simple to implement, but creates problems in the design of an application, as it is shown in the following example:

```
type Person <-> [Name:string;
                 WhoAreYou := meth():string is
                              "My name is " & self.Name & "." &
                              " I am a person."];
type Student <-> is Person and
                     [SNumber :int;
                      WhoAreYou := meth():string is
                                   "My name is " & self.Name & "." &
                                   " I am a student."];
type Car <-> [Plate:string;
              Owner :Person;
              OwnerName := meth():string is
                           self.Owner.WhoAreYou];
```

Let us construct an object with a role of type `Person`, and then let us extend it with the type `Student`:

```
let Bob := mkPerson([Name:= "Bob"]);
let BobAsStudent := inStudent(Bob, [SNumber := 1]);
```

Now, since the type of `BobAsStudent` is a subtype of `Person`, `BobAsStudent` can be used as `Owner` of a car:

```
let ACar := mkCar([ Plate := "xyz"; Owner := BobAsStudent]);
```

and the result of the expression `ACar.Owner.WhoAreYou` is `"My name is Bob. I am a student."`.
Now let us assume that `Bob` loses the type `Student`, by evaluating the expression `dropStudent(Bob)`, but still has the type `Person` Then if the expression `ACar.Owner.WhoAreYou` is evaluated again, a run-time failure will be generated since the message `WhoAreYou` is sent to a role which has been removed.
This effect is not satisfactory since the `Owner` of a car is expected to be a `Person` and its present value is an object which has lost the role with type `Student`,

but it has still the role with type `Person`. So there is an undesirable interaction between subtyping and losing roles.
As matter of fact, if the `ACar` would have been constructed as follows:

```
let ACar := mkCar([ Plate := "xyz"; Owner := BobAsStudent As Person]);
```

after the evaluation of `dropStudent(Bob)`, the result of `ACar.Owner.WhoAreYou` would have been correctly `"My name is Bob.  I am a person."`. This means that when writing an application attention must be paid to use a value of a certain type where a value of supertype is expected, if that value can lose the subtype. Consequently the advantage of subtyping is lost.

(2) Another solution, which avoids the above problems, is the following one adopted in Galileo97.
A different semantics has been adopted in Galileo97 for the drop operator, which avoids the above problems.
When an object receives a message, the method lookup algorithm ignores the methods redefined in dropped roles. However before executing the selected method, the system checks that the method has been found in a role whose type is a subtype of the static type of the role which has received the message. If it is not, a failure is generated. In other words, message passing fails if and only if the receiving object has lost the role which corresponds to the static type of the role which has received the message.
According to this semantics for the `drop` operator, after the evaluation of `dropStudent(Bob)`, the result of `ACar.Owner.WhoAreYou` is `"My name is Bob. I am a person."` although the `ACar` were constructed as

```
let ACar := mkCar([ Plate := "xyz"; Owner := BobAsStudent]);
```

This solution avoids the problems shown in the previous example, but it is more complex to implement, since the semantics of a message passing depends both on the static type of the receiver and on its the run-time status.

3.2 A Storage Model for Objects with Roles

This section presents a simple storage model for objects in order to give an informal semantics of objects with roles. For a discussion of extensions to this model toward a more realistic case see [Albano et al. 1995].

We prefer to present here just an informal "arrows and boxes" model, instead of a full formal description, because we believe it is more readable. The translation to a formal storage model, where every arrow becomes a location and every box becomes a tuple in the store, is shown in Appendix B.

The behavior of objects of standard class-based languages can be explained in terms of the simple storage model of Figure 1 [Abadi and Cardelli 1996]. In this model, an object is represented as a record which contains the fields of the object state and its methods. The special identifier `self` in a method `m` is always bound to the object that, according to this simple storage model, contains the method `m`. Usually the storage model is more complex than this one, and methods are not embedded into objects but are factored into the object type descriptor and shared by the objects of the same type. Even though a more complex storage model would be justified for reason of efficiency, the behavior of objects can be explained by this simple model.

Figure 2 shows how this simple storage model for objects might be modified to deal with roles. In this model there are two structures, the *object history*, and the *role structure*.

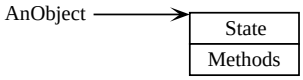

Fig. 1. A simple storage model for objects.

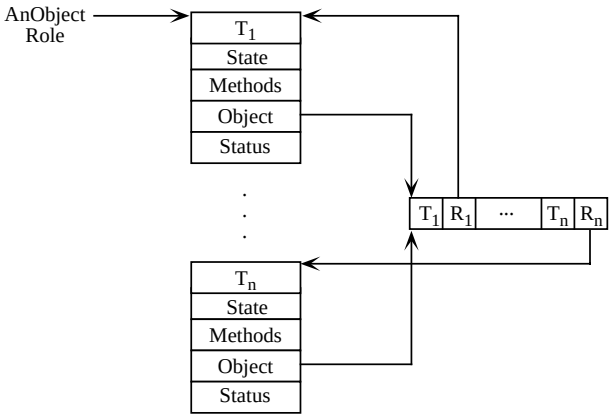

Fig. 2. A simple storage model for objects with roles.

The object history is used to store the history of the roles that an object has acquired. This structure is a sequence of pairs (type identifier, reference to the corresponding role structure), one pair for each type acquired by the object. The set of pairs is in temporal order. The object history is used (a) for method lookup, (b) to implement the operators `As` and `isalso`, (c) to find all the roles associated with a subtype of `T` which must become invalid when the operator `dropT` is executed, and (d) to implement object identities (e.g. two objects are the same when their object history is the same).

Each role structure stores information about the methods and the fields which are defined for the first time, or redefined, in the corresponding role type. In particular, a role structure contains the following information:

—the role type $\texttt{T}_i$;

—a status which is `valid` if the object has the role with type $\texttt{T}_i$, and is `removed` if the object has lost that role;

—a reference to the object history;

—the methods and the fields defined for the first time, or redefined, in the corresponding role type.

When a new object is created, a new role structure is created, and a reference to it is returned. When an object is extended with a new role type, a new role structure is created and is connected to the object, and a reference to this new structure is returned; an object is not allowed to be extended with a role type `T`, if a role `T` is already found in the object history.

When an object is created with a subtype $\texttt{T}_j$ of the root type $\texttt{T}_i$, the effect is the same as the creation of the object with the root type $\texttt{T}_i$, and then its extension with the subtype $\texttt{T}_j$. For example, the object in Figure 3 can be obtained by creating a `Person` and then by extending it with the type `Student`, or by creating directly a `Student`. The difference between the two cases is that, in the former the system returns a reference to a role structure of type `Person` and then a reference to a role structure of type `Student`, while in the latter the system returns only a reference to a role structure of type `Student`, and the access to the other role is possible only

by using the operator `As`.

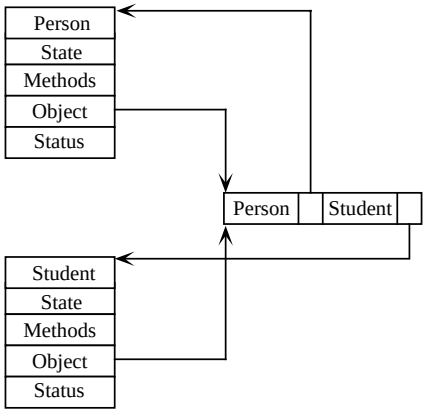

Fig. 3. The structure of an object with role types `Person` and `Student`.

When an object loses the type $\mathtt{T}_i$, the following actions are executed:

(1) the status of the role structure `R` with type $\mathtt{T}_i$ becomes `removed`;
(2) the pair $\mathtt{T}_i$,`R` is removed from the object history;
(3) steps 1 and 2 are repeated for every role in the object history whose type is a subtype of $\mathtt{T}_i$.

When a message is sent to a role `r`, its status is first checked in the corresponding structure, and then its run-time type `RT` is read and the object history is used to retrieve the role structures where the method should be looked for.

If the status of `r` is `valid`, downward lookup is performed by scanning the history from the last acquired role back to the `RT` role, looking for those roles whose type is a subtype of `RT` (the subroles of `r`). Upward lookup is then performed, by scanning the history from the `RT` role back to the first role, looking for the superroles of `r`.

If the status of `r` is `removed`, two possibilities arise. With the first semantics (Section ??), a failure is raisen. With the second semantics, every message has one extra parameter, the static type `ST` assigned by the compiler to `r`, and a failure is raisen if no `ST` role is found in the object history. If the `ST` role exists, then upward lookup is performed by scanning the history from the last role back to the first role, only considering the superroles of `r`. The presence of the `ST` role, and of its superroles, makes us sure that a method will be found.

4. VIRTUAL OBJECTS

The operators on objects presented so far allow objects to be built, and roles to be added and dropped without affecting object identity. These operators allow one to model the dynamic behavior of real-world entities, and are also useful for dealing with the most common kind of schema evolution, i.e. attribute addition or specialization. In this situation, in fact, it is possible to introduce a new subtype of the old type and to extend the old values with the new information. The type correctness of the preexisting applications and data structures will not be affected, and it is even possible to decide to partially modify the behavior of an application by specializing the behavior of some methods.[2]

[2]More precisely, a preexisting application is affected by method specialization only if the application exploits the double lookup ($obj.msg$) form of message passing.

However, the role mechanism cannot cope with the related problem of giving different views of the same object without affecting its behavior. This is because object extension actually modifies the object, and object extension can only modify an object in a very limited way (only field addition and specialization). If we call "real objects" those that have been explicitly constructed using the `mk` or `in` operators, what is needed is the possibility to define "virtual objects" by starting with objects and changing their interface while preserving their identity.

The virtual object mechanism we are going to describe has the following features:

—virtual objects are first class, statically and strongly typed values;

—a virtual object has the same identity as the object it is based on; when it is based on the combination of several objects, then its identity is a combination of the identities of the objects it is based on;

—a virtual object can add, remove, and rename fields of its base object; moreover a virtual object can have its own instance variables, which are accessed by its own methods;

—a virtual object can be based on more than one real object;

—the behavior of a real object is not affected by the existence of a related virtual object;

—a virtual object can be used exactly like a real object and vice versa, at least as long as the type rules described later on are satisfied;

—virtual objects allow one to update those components of the state of the real object which the virtual object allows one to view;

—a message to a virtual object to execute a method imported from an object returns an answer which is the same as if the message were directly sent to the real object.

We will now show how virtual objects are defined, and discuss their types and how these types are related to object and record types. Our approach is based on a set of basic operators that can be applied to objects, real or virtual, in order to produce other objects (virtual) with an identity preserving semantics. These operators can also be applied to records, which will be seen as a special case of virtual objects, although we will not stress this point. These object operators are then extended to collections of objects to define collections of virtual objects which are similar to views built over relations.

As we have pointed previously, similar operators for collections of objects have already been proposed by other authors, and the main contribution of our approach is to introduce these operators at an instance level in the framework of statically and strongly typed programming languages, to clarify the interactions between virtual objects and objects with roles, and to show the different semantics of method overriding and evaluation in virtual objects and objects with roles. The formal type rules of the virtual object mechanism are given in Appendix A.

4.1 Virtual object types

A virtual object role, which for the sake of simplicity we will sometimes call virtual object or virtual role, can be seen as a pair formed by the base object role and a mapping which may hide, rename, or even add some fields (even state components) with respect to the original object role (as described in the storage model in Section 4.3). More precisely, a virtual object role can generally be based on a set of base objects, with a mapping which manipulates their components and gives the external impression of a unique virtual entity.

Virtual object roles are typed with the `view` type constructor:

<$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]

where $\mathcal{T}_1, \ldots, \mathcal{T}_m$ are role types; $A_1, \ldots, A_n$ are labels; $\mathcal{S}_i$ are types. As a syntactic abbreviation, the type $\mathcal{S}_i$ of a label A_i can be omitted and it is assumed that it is the type of label A_i in one of the types $\mathcal{T}_1, \ldots, \mathcal{T}_m$, considered in the order.

Intuitively, the statement:

O: <$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]

means that O is a virtual object based on m object roles with types $\mathcal{T}_1, \ldots, \mathcal{T}_m$, and with signature [$A_1$:$\mathcal{S}_1$;...;$A_n$:$\mathcal{S}_n$]. As for object roles, for any object role type $\mathcal{T}$ with signature $\Downarrow\mathcal{T}$, the following type equivalence holds:

<$\mathcal{T}$> view [$\Downarrow\mathcal{T}$] $\sim \mathcal{T}$

where $\mathcal{T} \sim \mathcal{S}$ means that $\mathcal{T} \leq \mathcal{S}$ and $\mathcal{S} \leq \mathcal{T}$, i.e. that values of each type can be considered as if they were values of the other one. Similarly, for every record type [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$] the following type equivalence holds:

<> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$] $\sim$ [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]

The subtype relationship among real and virtual object types is defined in the next section.

Type hierarchies with `view` types

A subtype relationship $\mathcal{T}' \leq \mathcal{T}$ means that any operation which can be applied to any value of type $\mathcal{T}$ can also be applied to any value of type $\mathcal{T}'$. A virtual object O with type:

$\mathcal{T}$= <$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]

accepts two kinds of operations: message passing (O.A_i, O!A_i), and object extraction (O As $\mathcal{T}_j$). Hence, a type $\mathcal{T}'$ is a subtype of $\mathcal{T}$ if it contains enough object identities and components to be able to deal with every object extraction and message passing operation which is supported by $\mathcal{T}$. More precisely:

<$\mathcal{T}'_1$,...,$\mathcal{T}'_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]
$\leq$ <$\mathcal{T}_1$,...,$\mathcal{T}_r$> view [B_1:$\mathcal{R}_1$;...;B_s:$\mathcal{R}_s$]

if (a) for each $\mathcal{T}_i$ there exists a $\mathcal{T}'_j$ such that $\mathcal{T}'_j \leq \mathcal{T}_i$ and (b) for each pair B_i:$\mathcal{R}_i$ there is a pair A_j:$\mathcal{S}_j$ such that $A_j = B_i$ and $\mathcal{S}_j \leq \mathcal{R}_i$.

The principle that an object view type specifies the object extraction and message passing operations which can be applied to a type, implies that the following two equivalences hold:

$\mathcal{T}$ $\sim$ <$\mathcal{T}$> view [$\Downarrow\mathcal{T}$]
[A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$] $\sim$ <> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]

4.2 Virtual object constructors

The operators to build virtual objects are: **project**, **rename**, **extend** and **times**. These operators can be applied to sequences of objects using the notation **project***, **rename***, **extend*** and **times***.

Project

project is used to hide properties from an object role.

O project [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$].

returns the object O with components A_1,...,A_n only. More precisely, if O is an object role with type

<$\mathcal{T}_1,\ldots,\mathcal{T}_m$> view [A_1:$\mathcal{S}'_1$;...;A_n:$\mathcal{S}'_n$; B_1:$\mathcal{R}_1$;...;B_l:$\mathcal{R}_l$],

and if each $\mathcal{S}'_i$ is a subtype of $\mathcal{S}_i$, then O project [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$] returns the same object role O seen through type

<$\mathcal{T}_1,\ldots,\mathcal{T}_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$].

The type $\mathcal{S}_i$ of a label A_i can be omitted, and it is then assumed to be the type $\mathcal{T}_i$ of A_i in O.

Note that although project is formally defined on virtual objects alone, it can also be applied to real objects too, thanks to the type equivalence

<$\mathcal{T}$> view [$\Downarrow\mathcal{T}$] $\sim \mathcal{T}$.

The same observation holds for all the other virtual object operators that will be presented, which can all be applied in the same way to real and virtual objects to produce virtual objects.

Example 1. *Let us consider the following definitions:*

```
let type AnAddress :=
    [Street: var string;
     City: var string;
     Country: var string ];

let rec Persons class Person <->
    [Name: string;
     Income: var int;
     Address: AnAddress;
     BirthYear: int;
     Parents: [ Father: Person;
                Mother: Person ];
     WhoAreYou:= meth(): string is "My name is  " & self.Name & "." ];

let rec Employees subset of Persons class
Employee <-> is Person and
    [Salary: var int;
     Company: Company;
     WhoAreYou:= meth(): string is
                 super.WhoAreYou & " I work with company " & self.Company.Name ]
and
Companies class Company <->
    [Name: string;
     Location: string;
     Revenue: int ];

let type PersonView :=
         <Person> view [ Name;
                         Address: [ Street: var string;
                                    City: var string ];
                         WhoAreYou ];
```

Let John *be an object role of type* Person, JohnView *a virtual role of type* PersonView, Foo *and* Goo *two functions defined as follows:*

```
let JohnView :=
      John project [ Name;
                     Address: [ Street: var string;
                                City: var string ] ;
                     WhoAreYou] ;
```

```
let Foo := fun(x: Person) :Person is x;
let Goo := fun(x: PersonView):int is
              if x isalso Employee
                 then at (x As Employee).Salary
                 else at (x As Person).Income;
```

The following considerations apply:

—*By projecting* `Address` *to the indicated supertype of the original type,* `Country` *component is hidden in* `JohnView`*;*

—`John = JohnView` *returns true since* `project` *is an identity preserving operator and* `JohnView` *is defined starting from* `John`*;*

—`John.WhoAreYou = JohnView.WhoAreYou` *returns true since the method executed to answer the message* `WhoAreYou` *sent to* `JohnView` *is the method defined for* `John`*;*

—*the component* `Street` *of the* `Address` *of* `JohnView` *can be updated and this will also effect* `John`*. Likewise, an update of* `John` *will have the same effect on* `JohnView`*;*

—*as will be explained in Section 4.1,* `Person` *is a subtype of* `PersonView`*, but not vice versa, hence* `Goo(John)` *would be typed, while* `Foo(JohnView)` *would not;*

—*let us assume that* `John` *has been extended with the role type* `Employee`*. The following virtual object:*

```
let JohnEmplView :=
      (John As Employee)
      project [ Name;
                Company: [ Name: string; Location: string ] ;
                WhoAreYou] ;
```

has a type which is a supertype of `Employee`*, but it is not comparable with either* `Person` *or* `PersonView`*.*

—*The following example shows the definition of a derived sequence:*

```
let EmployeesView := derived
      Employees project* [ Name;
                           Company: [ Name: string; Location: string ] ;
                           WhoAreYou] ;
```

The elements of `EmployeesView` *are those of* `Employees` *with a supertype of* `Employee`*.*

□

Extend

`extend` is used to add or redefine methods and fields to an object.

`O extend [`$\mathtt{A}_1$`:`$\mathcal{S}_1$ `:= Expr`$_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$ `:= Expr`$_n$`. ]`

returns the object `O` extended with new fields whose value is specified by the expressions $Expr_1,\ldots,Expr_n$. If a label $\mathtt{A}_i$ was already present in `O`, `extend` overrides $\mathtt{A}_i$ with a value of a possibly unrelated type. More precisely, if `O` has type

`<`$\mathcal{T}_1,\ldots,\mathcal{T}_m$`> view [ `$\mathtt{B}_1$`:`$\mathcal{R}_1$`;...;`$\mathtt{B}_l$`:`$\mathcal{R}_l$`;`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_k$`:`$\mathcal{S}_k$`]`

with $k \leq n$, then the extension expression above has the type:

`<`$\mathcal{T}_1,\ldots,\mathcal{T}_m$`> view [ `$\mathtt{B}_1$`:`$\mathcal{R}_1$`;...;`$\mathtt{B}_l$`:`$\mathcal{R}_l$`;`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$`]` .

Note that this rule allows one to extend both real and virtual objects.

`extend` can be used to add both new fields and new methods, which belong to the virtual part of the object. To this end, the expression associated with a label may contain the pseudo-variable `me`, which denotes, recursively, the whole virtual object *after* it has been extended. This `me` variable can be used much in the same way as `self` in the real object, but there is a difference.

When a method defined in a role $\mathcal{R}$ but activated by inheritance by a message sent to a subrole $\mathcal{S}$, sends a message `msg` to `self`, then method lookup for `msg` starts from role $\mathcal{S}$. When any expression, message passing included, is applied to `me`, then `me` denotes the virtual object that has been created by the `extend` operation it is bound to, hence method lookup for `me.msg` starts from the virtual object where the method invoking `me.msg` is found. Technically, we say that `self` is *dynamically bound* to the role that receives the message, while `me` is *statically bound* to the virtual object that is created by the `extend` expression which `me` is bound to. The reason for this difference, which is discussed in Section 4.5, is that there is no guarantee that the virtual object which receives the message belongs to a subtype of the virtual object where the method is defined, and is connected with the requirement that the behavior of a real object must not be affected by the existence of a related virtual object.

EXAMPLE 2. *The following example shows how to redefine the structure and behavior of a person object; note that, in the definition of fields* `Mother` *and* `Father`*, using* `me` *or* `John` *makes no difference, while* `me` *is essential to access field* `Age` *inside* `WhoAreYou`*.*

```
let rec AnotherJohnView :=
  (John extend [ Age := meth():int is
                        CurrentDate().Year - me.BirthYear;
                 Mother := meth():Person is me.Parents.Mother;
                 Father := meth():Person is me.Parents.Father;
                 WhoAreYou := meth(): string is
                              (me As Person)!WhoAreYou &
                               " I am " & stringofint(me.Age) & " years  old"]
   ) project [ Name; Age; Mother; Father; WhoAreYou] ;
```

□

EXAMPLE 3. *The following example shows how to redefine the behavior of the person object* `John` *without changing its type so that it can be used as a parameter of the function* `Foo`*:*

```
let rec JohnForItalians := John extend
                              [WhoAreYou := meth(): string is
                                            "Mi chiamo  " & me.Name & "." ];
```

□

Rename

`rename` is used to change the name of the properties of an object. If `O` has the labels $\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{B}_1,\ldots,\mathtt{B}_l$,

`O rename (`$\mathtt{A}_1$ `=> A'`$_1$`; ...; `$\mathtt{A}_n$ `=> A'`$_n$`).`

returns `O` with the labels `A'`$_1$, ..., `A'`$_n$, $\mathtt{B}_1$, ..., $\mathtt{B}_l$, which must all be different. A label $\mathtt{A}_i$ may also be a path expression $\mathtt{A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}$.

More precisely, if `O` has the type:

`<`$\mathcal{T}_1,\ldots,\mathcal{T}_m$`> view [ `$\mathtt{B}_1$`:`$\mathcal{R}_1$`; ...; `$\mathtt{B}_l$`:`$\mathcal{R}_l$`; `$\mathtt{A}_1$`:`$\mathcal{S}_1$`; ...; `$\mathtt{A}_n$`:`$\mathcal{S}_n$`]`

and A'$_1$, ..., A'$_n$, B$_1$, ..., B$_l$, contains no duplicate, then the renaming expression above has type:

<$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [B$_1$:$\mathcal{R}_1$; ...; B$_l$:$\mathcal{R}_l$; A'$_1$:$\mathcal{S}_1$; ...; A'$_n$:$\mathcal{S}_n$] .

EXAMPLE 4. *The following definition gives a view of the person object* John *for Italian users:*

```
let JohnViewForItalians :=
       (John rename (Name => Nome;
             extend [ Presentati := meth(): string is
                                      "Mi chiamo " & me.Nome "."]
             project [ Nome; Presentati] ;
```

□

It may appear that **rename** is an operator that can be defined in terms of **extend** and **project**; for example; let O be an object with an attribute A$_i$:

```
let r := O rename ( Ai => Ai' );

let r' := (O extend [  Ai' := meth() :Ti is me.Ai ] )
              project [  <all attributes except Ai> ]
```

The two expressions above actually have a different meaning, since both r'.A'$_i$ and r'!A'$_i$ are equivalent to O.A$_i$, while r.A'$_i$ and r!A'$_i$ are generally different since r keeps upward and double lookup distinct.

For instance, let us assume that John has the type Person, and it has been extended with the type Student, where the method WhoAreYou has been redefined. Then the message John.WhoAreYou is answered using the method defined in Student, while John!WhoAreYou is answered using the method defined in Person. Now let us define two views:

```
let JohnViewOne:= John rename (WhoAreYou => IntroduceYourself);

let JohnViewTwo:= (John extend [  IntroduceYourself :=
                                      meth():string is me.WhoAreYou ]
                  ) project [  <all attributes except WhoAreYou> ] ;
```

While the effect of JohnViewOne.IntroduceYourself and JohnViewOne!IntroduceYourself is different as it was for John, JohnViewTwo.IntroduceYourself and JohnViewTwo!IntroduceYourself return the same value as John.WhoAreYou .

Times

times is used to create a virtual object by starting from two objects whose component names are all different.

```
O times O'.
```

returns an object which contains the identities of both O and O', and has the fields of O and O'.

More precisely, if:

O: <$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [A$_1$:$\mathcal{S}_1$;...;A$_n$:$\mathcal{S}_n$]
O': <$\mathcal{T}'_1$,...,$\mathcal{T}'_l$> view [B$_1$:$\mathcal{R}_1$;...;B$_k$:$\mathcal{R}_k$]

then O times O' has type:

<$\mathcal{T}_1$,...,$\mathcal{T}_m$,$\mathcal{T}'_1$,...,$\mathcal{T}'_l$> view [A$_1$:$\mathcal{S}_1$;...;A$_n$:$\mathcal{S}_n$;B$_1$:$\mathcal{R}_1$;...;B$_k$:$\mathcal{R}_k$] .

For example, if $\mathcal{T}$ and $\mathcal{T}$' are two real object role types with no common component names, then

`O times O': <`$\mathcal{T}$`,`$\mathcal{T}$`'> view [ `$\Downarrow\mathcal{T}$`;`$\Downarrow\mathcal{T}$`'] .`

This type indicates that the virtual object (a) can answer all the $\Downarrow\mathcal{T}$, $\Downarrow\mathcal{T}$' messages, and (b) contains both a $\mathcal{T}$ and a $\mathcal{T}$' object role, which can be recovered with the `obj As` $\mathcal{T}$ operator previously defined.

The `times` operator has been introduced to complete the object algebra, and to give a semantics to the `from` clause of a query that includes several collections of objects. `times` may appear similar to the `extend` operator, and as a matter of fact in many cases `extend` is sufficient, but there are two differences: `extend` is used to add attributes to a base object, or to redefine some of them, and the result is a virtual object that preserves the identity of the base object; `times` combines two base objects with different attributes, and the result is a virtual object that preserves the identity of both the base objects.

EXAMPLE 5. *The following view, built on the* `Employees` *and* `Companies` *classes in Example 1, defines a class of employees who are combined with their company.*

```
let EmployeesAndCompanies := derived
  select Emp times (Emp.Company rename (Name => CompName))
  from   Emp In Employees;
```

The result has the type: `seq <Employee,Company> view [Name; Income; Address; BirthYear; Sex; Parents; WhoAreYou; Salary; Company; CompName; Location; Revenue]`. *If* `John` *is an element of* `EmployeesAndCompanies`, *then* `John As Company` *returns the company where* `John As Employee` *works.*

□

4.3 A storage model for virtual objects

The behavior of virtual objects can be explained in terms of the simple storage model in Figure 4. In this model, a virtual object is an interface adaptor for a virtual or real object. When a virtual object is created from an object `O`, a reference is returned to a structure whose shape depends on the operator which has been used as follows :

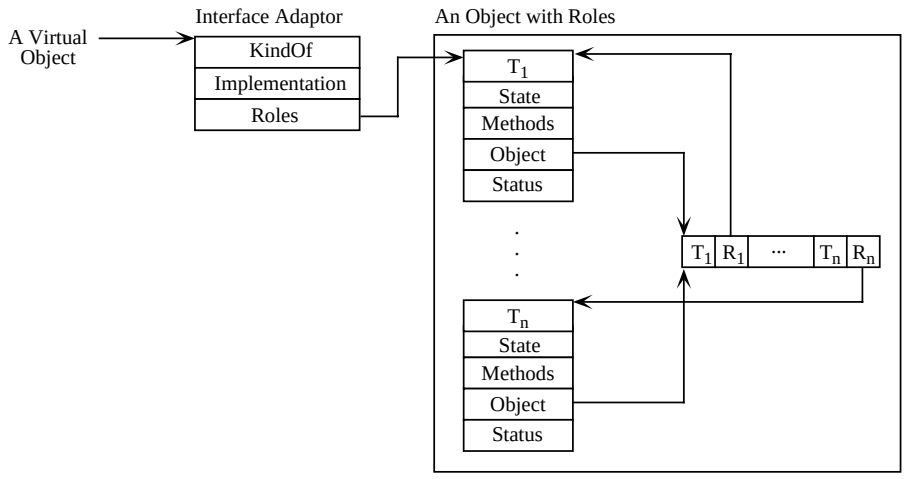

Fig. 4. The structure of a virtual object.

—`O project [A1:S1;...; An:Sn]` has no run-time effects and returns a reference to `O` with a new type which hides the components different from `[A1;...; An]` ;

—`O extend [A1:S1 := Expr1;...; An:Sn := Exprn]` returns a reference to an interface adaptor of kind `extend` which contains the new methods and fields `A1;...; An` defined in the virtual object, and a reference to the object `O` (Figure 5); the special identifier `me` in a method is recursively bound to the interface adaptor itself;

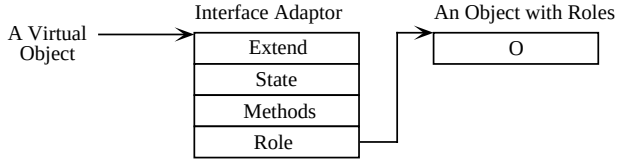

Fig. 5. The structure of a virtual object of kind `extend`.

—`O rename (A1 => A'1; ...; An => A'n)` returns a reference to an interface adaptor of kind `rename` which contains a table that maps the attribute and method names `A'i` in the virtual object to the original names `Ai` in `O`, and a reference to the object `O` (Figure 6);

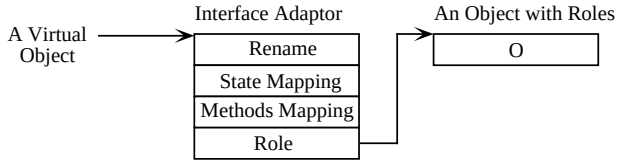

Fig. 6. The structure of a virtual object of kind `rename`.

—`O times O'` returns a reference to an interface adaptor of kind `times` which contains a table that associates every attribute with the object where the attribute should be looked for, and two references to the objects `O` and `O'` used to define the virtual object (Figure 7).

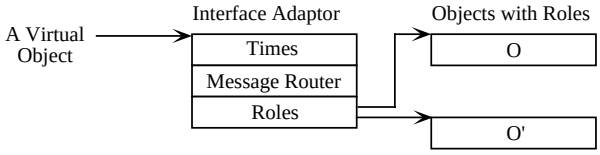

Fig. 7. The structure of a virtual object of kind `times`.

When a virtual object receives a message `m`, it tries to resolve it using its interface adaptor, otherwise it sends the message to the object used to define the virtual object.

Figure 8 shows an example of a virtual object `VirtualJohn` defined from the object role `John` as follows:

```
let VirtualJohn := John project [Name]
                        rename (Name => Surname)
                        extend[NewAttribute := ...;
                               NewMethod := meth() ... ]
```

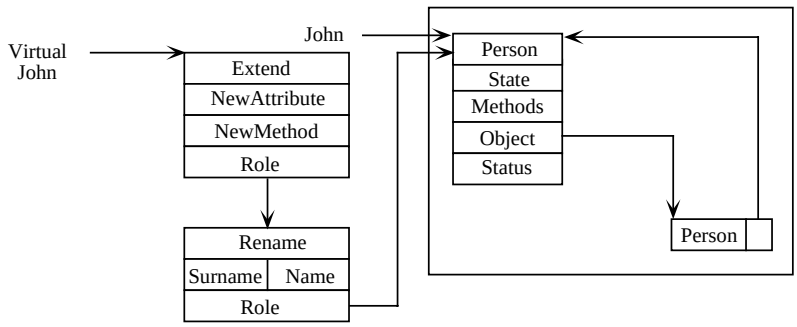

Fig. 8. An example of a virtual object.

With a more realistic storage model, to avoid chains of interface adaptors, every interface adaptor of a virtual object is a combination of those used to explain the effect of each operator and it ends with a reference to the real object (or the real objects), as shown in Figure 9.

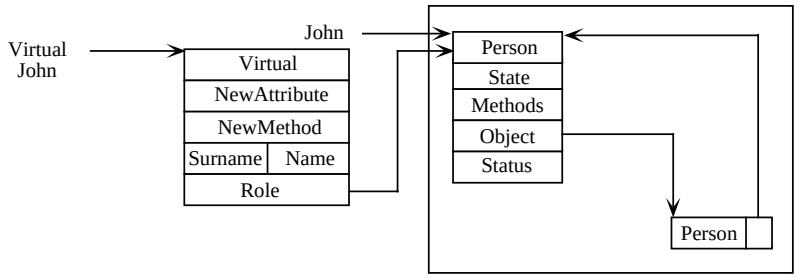

Fig. 9. An example of a virtual object.

4.4 The semantics of `As`

The `As` operator allows one to move among the roles of an object and to recover the real object from a virtual one. An expression `O As` $\mathcal{T}$, where `O: <`$\mathcal{T}_1, \ldots, \mathcal{T}_n$`> view [...]`, is well typed iff (a) $\mathcal{T}$ is a role type and (b) if at least one of the $\mathcal{T}_i$ has a common object supertype with $\mathcal{T}$. In this case, the `As` operator searches to see whether one of the real objects on which the virtual object is based has the $\mathcal{T}$ role. If this is case, the $\mathcal{T}$ role of that object is returned; otherwise, a failure arises. The `As` operator raises two issues:

—protection: in some situations, recovering the real object inside the virtual object should be disallowed;

—semantics: when a virtual object contains more than one real object with a $\mathcal{T}$ role, the `As` $\mathcal{T}$ operator must choose just one of them.

The protection issue can be dealt with by hiding a role type $\mathcal{T}$ in some sections of the code. Consequently, in those sections the system will not accept an `As` $\mathcal{T}$ operation. For example, the `As` operation at the end of the following piece of code is statically refused by the Galileo97 system.

```
let type Person := ...;
let John := mkPerson...;
let type MyPerson := <Person> view [ ...] ;
let JohnView := john extend [...]  project [...] ;
hide type Person;
```

```
JohnView As Person;        <------------
```

Note that, even if `Person` is hidden, the type `MyPerson` is still a valid type, since static scoping is used in Galileo97.

As for the semantics, consider the following code.

```
let John := mkPerson...;
let Peter := mkPerson... rename ...;
let JP := John times Peter;
```

It is not clear whether `JP As Person` should be equal to `John` or to `Peter`. In Galileo97 semantics, it is equal to `John`. In fact, in Galileo97, the real object roles hidden in a virtual object are ordered. Specifically, in the virtual object `(O1 times O2)`, all real object roles in `O1` come before those in `O2`; the `As` operator respects this order while looking for the result. A reasonable alternative would be a non-deterministic semantics, which does not specify in which order the real objects are considered. This semantics would make the `times` operator commutative, and would leave more freedom to Galileo97 implementors. In any case, this kind of semantics ambiguity is not very likely to arise in a real application.

4.5 The Semantics of `self` and `me`

Suppose that a message `m` is sent to a role of type `T`, and that the lookup mechanism activates the method `M` defined for `m` in a supertype `T'` of `T`. Now, if `M` sends a message `p` to `self`, should the search for the corresponding method start from the originally receiving role `T`, or from the role `T'` where the method `M` has been found? In every object oriented language, in this situation `self` stands for the most specialized role `T` which first received the message (dynamic binding of `self`). Consider the following example.

```
let rec type Picture <->
    [BoundingRect:= meth(): Rectangle is fail;
     DrawBoundingRect:= meth():null is
                 use Rectangle:= self.BoundingRect
                 in  <draw Rectangle>;
     ... ] ;
```

This piece of code exemplifies a typical usage of object-oriented languages: the bounding rectangle is computed in a different way in different subclasses. However, `DrawBoundingRect` can be implemented once and for all in the superclass since, thanks to the dynamic binding of `self`, the `self.BoundingRect` invocation will invoke the specific function of the receiving object, rather than the useless function defined for the generic `Picture` type.

The dynamic binding of `self` is an essential component of object-oriented languages, but it has a price: it constrains inheritance to be only used to produce subtypes. This means that it is impossible, for example, to remove or rename fields in the object type which is defined by inheritance. In fact, when a method such as `DrawBoundingRect` above is compiled and type-checked, the compiler only knows that `self` belongs to a type which inherits from `Picture`. Hence, if inheritance were not constrained, the compiler would not know anything about the type of the result of any message invocation for `self`. Typed object-oriented languages therefore always constrain inheritance to be used to produce subtypes alone.[3]

The situation is different when `extend` with `me` is considered. `extend` is essentially an inheritance operator, since it allows an object (the virtual one) to be built by

[3] See also [Bruce et al. 1995] for a different but strictly related approach.

starting from another one. The fact that here inheritance is exploited at instance level, rather than at the type level, is irrelevant. However, dynamic binding is not used for `me` for the following reasons:

—we cannot constrain `extend` to only produce values belonging to a subtype; one of the main reasons for introducing the viewing operator is to overcome the same limitation in the role mechanism;

—dynamic binding of `me` is not very useful. Dynamic binding of `self` is essential in role inheritance, because we expect that the role type where `self` is used will be extended with new subroles, and that these subroles may be used to complete the meaning of the current definition (consider the example above). With virtual objects, this style of programming by adding inheritance over inheritance (extension over extension) is not common. Moreover, the idea of completing or modifying the meaning of a virtual object by building a new object over it, is not compatible with the basic assumption that adding a view does not affect the behavior of the object viewed.

For the above reasons, the `me` identifier used in a method definition inside an `extend` operation is not linked dynamically to the virtual object that receives the message, but is statically linked to the object which is defined by the `extend` operation.[4]

4.6 Equality

In the context of object databases, three kinds of equality are usually considered:

—*Identity equality* (*identical*, "`==`"). This corresponds to the equality of references or pointers in conventional languages: two objects are identical if their identities are the same.

—*Shallow equality* (*shallow equal*, "`=`"). Two objects are shallow equal if they have the same run-time type and their states are identical. That is, it goes one level deep, and compares corresponding identities of the state components.

—*Deep equality* (*deep equal*). This is a purely value-based equality: two objects are deep equal if they have the same run-time type and their states are value-based deep equal.

In Galileo97, every type is associated with an equality function, and the equality of two values is determined using the equality function associated with their static type. More precisely, `a = a'` is well typed if the type of `a` is either a supertype or a subtype of the type of `a'`, and `a` and `a'` are compared using the equality function associated with the supertype. As a consequence, the same pair of values can be equal or different according to the type it is accessed through. For example, the following two expressions would evaluate to `false` and `true`, respectively, since the `b` field would be ignored in the second comparison:

```
[ a:=1; b:=2] = [ a:=1; b:=3] ==> false
[ a:=1; b:=2] :[ a:int] = [ a:=1; b:=2] ==> true
```

In Galileo97, equality is value-based for most concrete types, such as record and sequence types, while it is determined by identity for modifiable values (values of type `var T`), functions and roles. Equality by identity seems the most appropriate for roles and locations for two reasons:

—in database applications it is usually more important to know whether two objects represent the same real-world entity, rather than knowing whether they give the

[4] Static binding of `me` should not be confused with static binding of messages. For example, when a message is sent to `me`, the relative method is looked up dynamically.

same answer to every message. Likewise, for two locations, containing the same value is generally less interesting than being the same location;

—when two objects need to be compared by structure, they can simply be looked at through their record supertype. Likewise, to compare two locations `l1` and `l2` by content, one need only write `(at l1 = at l2)`.

The situation is slightly more complex with `view` types, which are to some extent intermediate between object and record types. In this case, the rule is that two values `O1` and `O2` belonging to type

$$\texttt{<}\mathcal{T}_1\texttt{,...,}\mathcal{T}_m\texttt{> view [A}_1\texttt{:}\mathcal{S}_1\texttt{;...;A}_n\texttt{:}\mathcal{S}_n\texttt{]}$$

are equal if:

$$\begin{aligned}&\texttt{(O1 As }\mathcal{T}_1\texttt{) = (O2 As }\mathcal{T}_1\texttt{) And ... And (O1 As }\mathcal{T}_m\texttt{) = (O2 As }\mathcal{T}_m\texttt{) And}\\&\texttt{ O1.A}_1\texttt{ = O2.A}_1\texttt{ And ... And O1.A}_n\texttt{ = O2.A}_n\texttt{ And}\\&\texttt{ O1!A}_1\texttt{ = O2!A}_1\texttt{ And ... And O1!A}_n\texttt{ = O2!A}_n\texttt{.}\end{aligned}$$

Note that:

(1) for each $\texttt{A}_i$ field the associated equality is used. Hence, methods are compared by identity (since they are functions), updatable fields are also compared by identity, and constant concrete fields are compared by value;

(2) two records are equal in type `[ ...]` iff they are equal in type `<> view [ ...]` .

The above points highlight that equality on virtual object values generalizes both role and record equalities. One may also expect that, whenever $T \sim S$, then $a : T = b$ is the same as $a : S = b$. However, this is not always true. As a counterexample, consider a pair of role types $\mathcal{S} \leq \mathcal{T}$. In accordance with the view types subtyping rules, the following equivalence holds:

$$\texttt{<}\mathcal{S}\texttt{> view [ ]} \sim \texttt{<}\mathcal{T}\texttt{,}\mathcal{S}\texttt{> view [ ]}$$

Now, let `s` be a role of type $\mathcal{S}$ and `t1,t2` be two different roles of type $\mathcal{T}$. Comparing `t1 times s` with `t2 times s` gives two different answers in the two types above, since only in the second case are the two virtual objects also compared with respect to the result of the operation `x As` $\mathcal{T}$, which gives two different results.

$$\begin{aligned}&\texttt{(t1 times s):<}\mathcal{S}\texttt{> view [ ] = (t2 times s) => true}\\&\texttt{(t1 times s):<}\mathcal{T}\texttt{,}\mathcal{S}\texttt{> view [ ] = (t2 times s) => false}\end{aligned}$$

The fact that two types that are equivalent with respect to subtyping are not equivalent with respect to equality is not very satisfactory, but could be avoided by adopting a more complex notion of equality, where two objects in type $\texttt{<}\mathcal{T}_1\texttt{,...,}\mathcal{T}_m\texttt{>}$ `view [...]` are also compared with respect to the result of `O As` $\mathcal{S}$ for every supertype $\mathcal{S}$ of every $\mathcal{T}_i$. Choosing the best approach is a matter for further research.

As an example, consider the roles `John`, `JohnAsStudent`, and `JohnAsAthlete` defined at the beginning of Section 3, and the role `NewAthlete` defined as follows.

```
let NewAthlete := mkAthlete([ Name := "John Smith";
                              BirthYear:=  1967;
                              Code := 2;
                              Sport := "Basket" ] );
```

The following expression

```
JohnAsStudent  = JohnAsAthlete;
```

is not well typed since `JohnAsStudent` and `JohnAsAthlete` are not subtypes of each other.

`John` and `JohnAsAthlete` can be compared if they are considered of type `Person` as follows, and the equality prdicates returns true:

```
(JohnAsStudent:Person)  = JohnAsAthlete;
```

Two virtual objects can be compared by identity using an appropriate view type. For example:

```
let type IdPerson := <Person> view [] ;

(JohnAsStudent:IdPerson) = (JohnAsAthlete:IdPerson); is true
```

Two role values can be compared by ignoring their identity using an appropriate record type, which allows both the shallow and deep equality to be simulated. For example:

```
let type PersonRecord := [Name:string; BirthYear: int] ;

(JohnAsStudent:PersonRecord) = (JohnAsAthlete:PersonRecord); is true

(JohnAsStudent:Person) = NewAthlete; is false

(JohnAsStudent:PersonRecord) = (NewAthlete:PersonRecord); is true
```

5. VIRTUAL CLASSES

The virtual objects operators described so far, apart from being interesting in themselves, constitute the building blocks that can be used to define a higher level virtual class mechanism along the lines, for example, of the one proposed in [Guerrini et al. 1997]. The translation of such a mechanism in terms of virtual object operations defines its semantics in a precise way.

We will consider the following virtual class mechanism (`where`, `store`, `compute`, and `import` clauses are optional).

```
let NameView classview as
        x In BaseClass where Cond
    EleType := TBaseClass
                store    [ S1:= D1; ...; Sn:= Dn ]
                compute  [ C1:= E1; ...; Cn:= En ]
                import   [ I1; ...; In ];
```

This mechanism specifies:

(1) the name *NameView* and the extent of the virtual class, by a query over a base class *BaseClass* which may be either real or virtual;

(2) the type *EleType* of the objects in the virtual class, whose signature includes the base object attributes I_i specified with the clause `import`, the computed attributes C_i specified with the clause `compute`, and the stored attributes S_i (D_i are the default values) specified with the clause `store`. The I_i, C_i, and S_i must be distinct names.

A virtual class definition introduces a new kind of relation between classes, called *based-on*: *NameView* is *based-on BaseClass*. This relation implies the subset relation between *NameView* and *BaseClass* (as far as the identity based equality is concerned), but not a subtype relation.

A virtual class definition is translated into the definition of a view type for the virtual class element type, and a `derived` expression which defines the virtual

class extent. The **compute** and **import** clauses are directly translated in terms of **extend*** and **project***; for the sake of simplicity, we postpone the treatment of stored attributes.

```
let type EleType := <TRealClass> view
                    [ I1; ...; In; C1 :T1; ...; Cn :Tn ];

let NameView := derived
                    select   x
                    from     x In BaseClass
                    where    Cond
                    extend*  [ C1:= E1; ...; Cn:= En ]
                    project* [ I1; ...; In; C1 :T1; ...; Cn :Tn ];
```

A virtual class mechanism must also allow one to insert a virtual class into the class hierarchy, considering its three aspects:

—the subtype relationship between the subclass element type and the superclass element type;

—the subset relationship between the subclass extent and the superclass extent;

—the inheritance relationship between the subclass definition and the superclass definition.

This is accomplished by the following mechanism, which allows a virtual class to be defined by inheritance from a previous virtual class definition.[5]

```
let NameSubView subset of SuperView classview as
        x In BaseClass where Cond
    SubEleType := is SuperEleType and
                  TBaseClass
                     store    [ S1 := D1; ...; Sn:= Dn ]
                     compute  [ C1:= E1; ...; Cn:= En ]
                     import   [ I1; ...; In ];
```

In this case, *SuperView* must be a (virtual) class based on a superclass of *BaseClass*. The class *NameSubView* inherits the **where**, **store**, **compute** and **import** clauses from the *SuperView* definiton, and it can extend them or override the attribute definitions with the *strict inheritance* constraint: whenever an attribute is redefined, its type must be a subtype of its previous type (or the same type). Thanks to these constraints and to the inheritance of the **where** clause, the extent of the defined virtual class is included in that of the superclass, and its element type is a subtype of that of the superclass.

The subset relationship between virtual classes, and the *based-on* relationship between a virtual class and a base class, are two different relationships: the first implies the subset, subtype, and inheritance relationships, while the second implies the subset relationship alone (Figure 10).

The semantics of the virtual subclass definition can be defined by extending the previous translation with a treatment of the inherited clauses. Inherited clauses are copied inside the translation; the order in which they are inserted in the translation determines the relative priorities. Since **project*** requires all the projected attributes to be different, we combine the different sets of attribute names with a **+** operator, which, whenever an attribute is both on its left and on its right hand side, puts in the result only the one on the right hand side:

[5]It is also possible to inherit from a real class, which is seen as a virtual class based on itself, without **where**, **store**, and **compute** clauses, and where every attribute is imported.

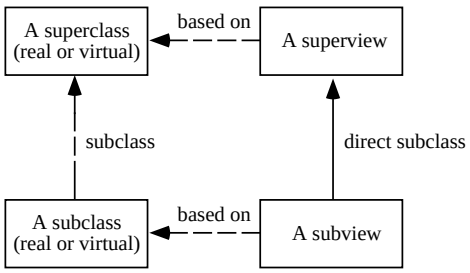

Fig. 10. The subset and *based-on* relationships.

$(A:T;B:U)+(A:V;C:W)=(B:U;A:V;C:W)$. In the translation, we use I',C',T',E' for the inherited attributes and their types and definitions, and $Cond'$ for the inherited condition (for the sake of simplicity, we will again postpone the treatment of stored attribute).

```
let type SubEleType := <TRealClass> view
                        [ (I'_1; ...; I'_n);
                        + (I_1; ...; I_n);
                        + (C'_1 :T'_1; ...;C'_n :T'_n);
                        + (C_1 :T_1; ...;C_n :T_n) ];

let NameSubView := derived
                   select   x
                   from     x In BaseClass
                   where    Cond And Cond'
                   extend*  [ C'_1:= E'_1; ...; C'_n:= E'_n ]
                   extend*  [ C_1:= E_1; ...;C_n:= E_n ]
                   project* [ (I'_1; ...;I'_n);
                            + (I_1; ...; I_n);
                            + (C'_1 :T'_1; ...;C'_n :T'_n);
                            + (C_1 :T_1; ...;C_n :T_n) ];
```

The value of such a translation is twofold:

—once the translation has been formally defined, the semantics of the virtual class construct is defined too; for example, the translation specifies what happens when an attribute with the same name is both "computed" in a superview and imported in a subview;

—the translation suggests which typing rules should be defined for the virtual class construct; for example, the translation suggests the strict inheritance constraint, and the fact that the expressions E_i should be type-checked in a context where the type of `me` is *TBaseClass* extended with the inherited computed attributes. Observe that, in a strongly typed context, defining the correct type rules is generally as difficult and as crucial as defining a non ambiguous semantics.

EXAMPLE 6. *The following example shows two virtual classes* `Adults` *and* `EngineeringStudents` *defined from the classes* `Persons` *and* `Students`*:*

```
let Persons class Person <-> [Name: string; BirthYear: int];

let Students subset of Persons class
    Student <-> is Person and [SNumber: string; Faculty: string];

% virtual classes %
```

```
let rec
Adults classview as
        p In Persons where  CurrentYear() - p.BirthYear > 17
Adult := Person
          compute [WhoAreYou := meth(): string is
                                "My name is  " & me.Name & "." ]
          import  [Name];

let rec
EngineeringStudents subset of Adults classview as
    s. In Students where s.Faculty = "Engineering"
EngineeringStudent := is Adult and
                      Student
                      compute [Age := meth ():int is
                                      CurrentYear() - me.BirthYear]
                      import  [SNumber; Faculty];
```

The translation of these definitions using the virtual object operators is as follows:

```
let type Adult := <Person> view [Name;WhoAreYou:string];

let Adults := derived
   (select p
   from    p In Persons
   where   CurrentYear() - p.BirthYear > 17)
   extend*  [WhoAreYou := meth(): string is
                          "My name is  " & me.Name & "."]
   project* [Name];

let type EngineeringStudent :=
    <Student> view [Name;
                    SNumber; Faculty;
                    WhoAreYou:string; Age:int ];

let EngineeringStudents := derived
   (select s
    from   s In Students
    where  s.Faculty = "Engineering" And  CurrentYear() - s.BirthYear > 17 )
    extend*  [WhoAreYou    := meth(): string is
                              "My name is  " & me.Name & "."]
    extend*  [Age          := meth():int is CurrentYear() - me.BirthYear]
    project* [Name; SNumber; Faculty; WhoAreYou; Age];
```

□

To translate a virtual class with stored attributes, a new real class (*extension class*) is defined with elements that contain the stored attributes of a virtual object and a reference to its base object. The extension class is queried whenever a stored attributed of the virtual class is requested. The elements of the extension class are created on demand: the first time a stored attribute of a virtual object is accessed, the corresponding object in the extension class is created and initialized as specified in the `store` clause. The next time a stored attribute of the same virtual object is accessed, the same object of the extension class is retrieved.

Let us show first the translation for the case without inheritance, as defined at the beginning of the section; we assume that U_i is the type of the expression D_i. In Galileo97, E_1 `iffails` E_2 executes E_2, and returns its value, if E_1 fails; `get` Q fails when it finds no value which satisfies the query Q.

```
let
```

```
NameView_Ext class
TNameView_Ext <-> [the_TBaseClass :TBaseClass;
                   S1:= D1 :U1; ...; Sn:= Dn :Un];

let get_NameView_Ext := fun(One_TBaseClass :TBaseClass) :TNameView_Ext is
                        get NameView_Ext
                        where the_TBaseClass = One_TBaseClass
                        iffails
                        mkTNameView_Ext([the_TBaseClass:= One_TBaseClass]);

let type EleType := <TRealClass> view
                    [I1; ...; In;
                     S1 :U1; ...; Sn :Un;
                     C1 :T1; ...;Cn :Tn ];

let NameView := derived
                select   x
                from     x In BaseClass
                where    Cond
                extend*  [S1:= meth():U1 is  get_NameView_Ext(me).S1;
                          ...;
                          Sn:= meth():U1 is  get_NameView_Ext(me).Sn ]
                extend*  [ C1:= E1; ...; Cn:= En ]
                project* [I1; ...; In; S1; ...; Sn;
                          C1 :T1; ...;Cn :Tn ];
```

EXAMPLE 7. *The following example shows a virtual class* `PersonsWithAddress` *with a stored attribute, and its translation using the virtual object operators:*

```
let rec
PersonsWithAddress classview as
    p In Persons where true
PersonWithAddress := Person
                     store   [Address := var ""]
                     compute [ChangeAddress := meth(newAddr:string): null is
                                               me.Address <- newAddr]
                     import  [Name];
```

The translation is as follows:

```
let PersonsWithAddress_Ext class
    TPersonsWithAddress_Ext <-> [the_Person: Person;
                                 Address:= var "" ]);

let get_PersonsWithAddress_Ext :=
    fun (One_Person :Person) :TPersonsWithAddress_Ext
    is get PersonsWithAddress_Ext where the_Person = One_Person
       iffails
       mkTPersonsWithAddress_Ext([the_Person:= One_Person]);

let type PersonWithAddress := <Person> view [Name;
                                             Address:var string;
                                             ChangeAddress:string->null];

let PersonsWithAddress := derived
   (select p
    from   p In Persons
    where  true)
```

```
extend*  [Address := meth(): var string is
                       get_PersonsWithAddress_Ext(me).Address ]
extend*  [ChangeAddress := meth(newAddr:string): null is
                           me.Address <- newAddr]
project* [Name; Address; ChangeAddress];
```

□

Finally, let us show the translation for a virtual class with stored attributes defined by inheritance from a superview, which has stored attributes too. In this case, the extension class *NameSubView_Ext* for the stored attributes is defined as a subclass of the extension class *NameSupView_Ext* used to translate the superview, and the function *get_NameSubView_Ext* is defined in terms of the *get_NameSupView_Ext* function. If the superview is defined without stored attributes, the *NameSubView_Ext* extension class and the function *get_NameSubView_Ext* are defined as shown before for a virtual class with stored attributes, and the inherited attributes S'_i will not appear in the translation.

```
let
NameSubView_Ext subset of NameSupView_Ext class
TNameSubView_Ext <-> is TNameSupView_Ext and
                        [ S1:= D1 : U1; ...; Sn:= Dn : Un];

let get_NameSubView_Ext :=
            fun(One_TBaseClass : TBaseClass) : TNameSubView_Ext is
            use the_Ele:= get_NameSupView_Ext(One_TBaseClass)
            in the_Ele As TNameSubView_Ext
               iffails
               in TNameSubView_Ext(the_Ele, []);

let type SubEleType := <TRealClass> view
                        [ (I'1; ...; I'n);
                        + (I1; ...; In);
                        + S'1 : U'1; ...;S'n : U'n;
                        + S1 : U1; ...; Sn : Un;
                        + (C'1 :T'1; ...;C'n :T'n);
                        + (C1 :T1; ...;Cn :Tn) ];

let NameSubView := derived
         select   x
         from     x In BaseClass
         where    Cond And Cond'
         extend*  [ S'1:= meth():U'1 is get_NameSupView_Ext(me).S'1;
                    ...;
                    S'n:= meth():U'n is get_NameSupView_Ext(me).S'n ]
         extend*  [ C'1:= E'1; ...; C'n:= E'n ]
         extend*  [ S1:= meth():U1 is get_NameSubView_Ext(me).S1;
                    ...;
                    Sn:= meth():Un is get_NameSubView_Ext(me).Sn ]
         extend*  [ C1:= E1; ...; Cn:= En ]
         project* [ (I'1; ...; I'n);
                  + (I1; ...; In);
                  + (S'1; ...; S'n);
                  + (S1; ...; Sn);
                  + (C'1 :T'1; ...;C'n :T'n);
                  + (C1 :T1; ...;Cn :Tn) ];
```

EXAMPLE 8. *The following example shows a virtual class* `StudentsWithPhone`*, with a stored attribute* `Phone`*, defined by inheritance from the virtual class* `Persons-WithAddress`*, and its translation, using the virtual object operators:*

```
let rec
EngStudentsWithPhone subset of PersonsWithAddress classview as
    s In Students where s.Faculty = "Engineering"
EngStudentsWithPhone:= is PersonWithAddress and
                        Student
                        store   [Phone := var ""]
                        compute [ChangePhone := meth(newPhone:string): null is
                                                me.Phone <- newPhone]
                        import [SNumber; Faculty];
```

The translation is as follows:

```
let EngStudentsWithPhone_Ext subset of
        PersonsWithAddress_Ext class
    TEngStudentWithPhone_Ext <-> is TPersonsWithAddress_Ext and
                                    [Phone:= var ""];

let get_EngStudentsWithPhone_Ext :=
    fun (One_Student :Student) :TEngStudentWithPhone_Ext

    is use the_Ele := get_PersonsWithAddress_Ext(One_Student)
       in the_Ele As TEngStudentWithPhone_Ext
          iffails
          inTEngStudentWithPhone_Ext(the_Ele,[]);

let type EngStudentWithPhone :=
    <Student> view [Name; Address:var string; ChangeAddress:string -> null;
                    Phone:var string; ChangePhone:string -> null;
                    SNumber; Faculty ];

let EngStudentsWithPhone := derived
   (select s
    from   s In Students
    where  s.Faculty = "Engineering" )
    extend*  [Address := meth(): var string is
                         get_PersonsWithAddress_Ext(me).Address ]
    extend*  [ChangeAddress := meth(newAddr:string): null is
                               me.Address <- newAddr]

    extend*  [Phone := meth(): var string is
                       get_EngStudentsWithPhone_Ext(me).Phone ]
    extend*  [ChangePhone := meth(newPhone:string): null is
                             me.Phone <- newPhone]
    project* [Name; Address; ChangeAddress;
              Phone; ChangePhone; SNumber;Faculty];
```

□

6. RELATED WORKS

The idea of extending object databases with a view mechanism has been discussed by several authors (e.g., [Dayal 1989], [Bancilhon et al. 1990], [Bertino 1992], [Abiteboul and Bonner 1991], [Santos et al. 1994], [Guerrini et al. 1997], [Kim and Kelley 1995], [Ohori and Tajima 1994], [Parent and Spaccapietra 1985], [Rundensteiner

1992], [Scholl and Scheck 1990], [Scholl et al. 1994], [Heiler and Zdonik 1990], [Zdonik 1990],[Leung et al. 1993]). Most of them follow a "collection based" approach, which is quite different from our "object based" approach. The collection based approach is based on a "virtual class" mechanism. The object based approach, which is adopted in Galileo97, first defines a set of virtual object operations, which allow a virtual object to be built starting from a set of other objects, and then combines these operators with a binding-by-name operation, such as our `let ViewName:= derived QueryExpr`, to define a view mechanism.

The main issues which are studied in this approach are:

—the semantics of virtual object operations;

—method resolution and the semantics of `self`;

—a type system for virtual objects.

The essential differences between the two approaches are:

—Semantics: The object based approach is characterized by a simpler semantics. The notion of "a class whose objects are defined by a query" breaks some basic assumptions of the object-oriented data model, especially when classes are identified with object type extents and when virtual classes are inserted into the class hierarchy. This raises some problems, for example with respect to: (a) method resolution, (b) semantics of insertion/removal for virtual classes, (c) placement of virtual classes in the class hierarchy, and (d) assignment of an "object identity" to the elements of virtual classes. The object based approach, on the other hand, divides virtual class definition into two atomic notions whose semantics is easier to define.

—Expressive power: In the object based approach, object operations and binding-by-name are first class operators which can be freely used inside programs, while in the class based approach the virtual class operation can only be used to define schemas; this makes the object based approach very flexible. The ability to build virtual objects from more than one base object is also a feature which is not usually found in other approaches.

—Virtual class and virtual object updating: In the object based approach it is not possible to explicitly insert or remove elements from a virtual class, while this is possible, under certain conditions, in some class based approaches. This is an important limitation of the object based approach, which can however be overcome. At the object level, in both approaches every modifiable field of the base object which is directly accessible through the virtual object can be modified from the virtual object as well.

—Class hierarchies: Most class based approaches merge the subtype, inclusion and method lookup hierarchies. This fact creates some problems with virtual classes. For example, if a virtual class `V` is defined by both restricting and projecting a base class `B`, then `V` extension is included in `B`, but `V` type is a supertype of `B` type. A solution to these problems is in [Guerrini et al. 1997].
In our object based approach we distinguish the subtyping, subset inclusion, and method lookup hierarchies (inheritance).[6] Virtual objects are assigned a first-class type, which is automatically placed in the subtype hierarchy, similarly to any other first class type of the language. Method resolution is not class based but is object based, hence virtual objects do not participate to the method lookup hierarchy.

[6]By method lookup hierarchy we mean the order relation between object types which is used to perform method lookup.

More specifically, the different approaches can be analyzed with respect to the following questions:

(1) Is the view mechanism based on collections of objects or on individual objects?

(2) How do virtual objects and message resolution interact?

(3) Does the language provide two orthogonal mechanisms to extend dynamically objects with new types and to define virtual objects?

(4) Do the operators for defining virtual objects preserve object identity?

(5) Is there a mechanism to group different objects into a single virtual object?

(6) Is a virtual object a first class value, and, consequently, does it have a type and can it be used like any other value of the language? Similarly, does a virtual class behave exactly like a base class?
Moreover, if the language is typed, is a virtual object type included in the language subtype hierarchy? Are rules provided to establish when a virtual object type is a subtype of an object type or of another virtual object type?

(7) Is it possible to insert objects into a virtual class?

(8) Can a virtual object have its own state and methods?

(9) Is there an operator to go from a virtual object to the object from which it has been defined?

Let us compare the solution offered by Galileo97 with three other proposals in the framework of a typed database programming language, AQUA [Leung et al. 1993], COCOON [Scholl et al. 1994], and O_2 [Abiteboul and Bonner 1991], and with the proposals in [Kim and Kelley 1995] (UniSQL/X), in [Ohori and Tajima 1994], and in Chimera [Guerrini et al. 1997].

(1) The Galileo97 and the Ohori-Tajima view mechanisms are based on the operation of building a virtual object, while the other approaches work by defining "virtual classes". While in Galileo97 and in the Ohori-Tajima approach virtual objects are built by value-level operations which can be exploited by any program, in the other approaches the creation of a virtual class is a schema-level operation, which is available during the schema definition phase only. This is the fundamental design choice, which has many consequences: the Galileo97 and Ohori-Tajima approach, based on a first class virtual object creation operation, gives rise to a more flexible mechanism, while the alternative approach, which confines virtual class creation to schema definition time, makes it easier to exploit traditional (i.e. relational) implementation techniques.

(2) In Galileo97, a virtual object is essentially composed by a virtual interface applied to a base object. When the virtual object receives a message, the corresponding method is first looked up in the interface and then in the base object. This lookup process resembles standard message resolution, with two important differences. Firstly, the semantics of `self` has to be modified, as explained in this paper. Secondly, a method defined for a virtual object, which overrides a method defined for the real object, does not affect how the real object itself answers the message, even when double lookup is used; the behavior of a real object can be changed only by extending it with a new type.
In the Ohori-Tajima proposal, method definition and message resolution are not studied.
The O_2 approach is very different. While in our approach each virtual object is represented by a specific data structure which contains a reference to the corresponding real object, in the O_2 approach only real objects have a physical representation (at least in the abstract model, which does not necessarily

coincide with the actual implementation). Virtual objects, or rather virtual *classes*, come into play since all applications access data through a specific schema, which may be either the real schema or a virtual one, and the behavior of objects depends on the chosen schema. A virtual schema specifies a set of virtual classes, along with which objects belong to these virtual classes, and how methods are implemented for each virtual class. When a virtual schema exists, the interfaces and methods of every object are those specified by that schema. Hence, when a message is sent to an object, it is necessary to determine to which classes the object belongs according to the current schema in order to execute the corresponding method; determining these classes is generall not easy. Moreover, if the object belongs to different unrelated virtual classes which implement the message, the ambiguity needs to be broken in some way. It is interesting to note that the current version of O_2 deals with these problems by adopting an approach which is quite similar to our approach, i.e. by "materializing" virtual objects [dos Santos 1994; dos Santos and Waller 1995]. It is also interesting to note, though this point is not explicitly discussed in the O_2 papers, that the semantics they adopt for `self` seems to be the same as the one we adopt, despite the major differences between the two approaches.
The UniSQL/X approach is intermediate. Each element in a virtual class has an OID which is different from the OID of the corresponding base object. The OID identifies the virtual class to which the object belongs, and is used to perform method dispatching. Hence, method resolution is class based as in O_2, but every virtual object contains (in its OID) enough information to perform an efficient method dispatch, as in our approach.
In Chimera each virtual object has the same OID as the corresponding real object. Method resolution is based on both the object OID and the static type of the object. The type hierarchies of real and virtual objects are kept separate, hence there is no interference between real and virtual methods.
In COCOON overriding is not allowed, hence no message resolution is needed.

(3) Only the Galileo97 type system supports two orthogonal mechanisms to extend dynamically objects with new types and to define virtual objects. The same features have been studied in Chimera.

(4) Galileo97, O_2, the Ohori-Tajima approach, and COCOON provide operators for defining object views with object identity preserving semantics. In AQUA this property only holds for views that do not change object structures: the project and join operator return a tuple. UniSQL/X objects do not have the same OID as the corresponding base object, but the OID of the base object can be extracted from the virtual object. In Chimera, when a virtual class is defined, the programmer can choose whether the original OID will be preserved, or a new OID generated, or no OID provided.

(5) Galileo97 provides an operator to construct a virtual object by starting from two (or more) objects; the resulting virtual object contains the identities of both the starting objects. AQUA provides an operator to construct a pair which has two objects as its components; this is also possible in the Ohori-Tajima approach. O_2 provides an operator to construct a record with the attributes of two objects, and a mechanism to construct a new object from this record. In UniSQL/X a virtual class whose objects are built starting from more than one object can be created without any particular problem since, in this approach, the identity of the virtual object is different from the identity of its base object(s). The same happens in Chimera by generating a new OID.

(6) In all of these systems a virtual object is a first class value. Consequently, it has

a type, and can be used like any other value in the language. A virtual object type is included in the language subtype hierarchy, and rules are provided to establish when a virtual object type is a subtype of an object type, a virtual object type, or a record type. In UniSQL/X two different hierarchies are defined for base object types and for virtual object types. In Galileo97 and in the Ohori-Tajima approach the subtype relationship is inferred automatically by the type checker, exploiting rules such as those in Section 4.1. In O_2 and in UniSQL/X, on the other hand, the subtype relationship must be defined explicitly.

(7) Only the UniSQL/X approach allows one to insert objects into virtual classes, under some conditions. This is achieved by translating this operation into the insertion of an object into the base class. The same is true for class removal.

(8) Only Galileo97 and Chimera enable one to add new state components and new methods to a virtual object. In the Ohori-Tajima approach new state components can be added, but methods are not dealt with. In O_2 and in UniSQL/X a virtual object can be extended with new computed attributes and methods, but can have no new modifiable state components on its own. In COCOON a virtual object can be extended only with new computed attributes. In AQUA a virtual object cannot be extended at all.

(9) Galileo97, UniSQL/X and the implemented version of the O_2 approach support an operator to go from a virtual object to the object from which the virtual object has been defined. This operator may be added without any problems to the Ohori-Tajima approach.

The Galileo97 approach also has similarities with the one adopted in [Cardelli and Mitchell 1991], where a set of operators on records is defined. In fact both approaches work in the context of a statically and strongly typed functional programming language. However, our work also deals with the object-oriented aspects, namely identity preservation and semantics of *self*, while Cardelli and Mitchell's study extends to polymorphic functions, which are not addressed here.

7. CONCLUSIONS

Two mechanisms have been presented to add flexibility to a programming language for object databases: roles and virtual objects. Roles allow objects to be dynamically extended to model entities which change their behavior, and the class they belong to, over time. Virtual objects are a mechanism to give a different interface of objects, redefining their structure and behavior, without affecting the behavior of the original object.

Roles and virtual objects are similar in many respects, since they both allow objects to be defined whose behavior depends on the observer, and they both allow an object to be extended. There are, however, some essential differences:

—the set of the roles of an object is part of the object itself, and the object can be tested with the predicate `isalso` to find out which roles it has; while a view is conceptually external to the object (for this reason, an operation is defined to remove a role from an object, while no similar operation is needed for virtual objects);

—adding a new role to an object transforms its type into a subtype, while the corresponding view operation `extend` produces an object whose type may not be related to the original one;

—the behavior of an object changes when it gains a new role, while it is not affected by the creation of a new virtual object;

—the set of roles which an object can belong to, is constrained by the role type hierarchy which has been statically defined, while the set of view types which can be used to access an object is open ended;

—both roles and virtual objects can send messages to themselves, but in the first case the message is received by the role that received the "original message" (see Section 4.5), while in the second case the message is received by the virtual object where the method is defined.

The main contributions of this work are (a) the clarification of the relationship between the two mechanisms, (b) the study of a set of statically and strongly typed operators which allow the virtual object mechanism to be included in a typed language, and (c) the use of these operators to give the semantics of a higher level mechanism to define virtual classes. The semantics of these operators has been defined through a simplified storage model, which reflects some aspects of the current implementation. A formal account of the essential mechanisms of the language is presented in Appendix B.

Both the role and the virtual object mechanisms have been implemented in a main memory implementation of the Galileo97 language for personal computers.

Acknowledgments

This article benefitted from discussions with S. Abiteboul, E. Bertino, G. Guerrini, G. Mainetto, and M. Scholl. Thanks also to the anonymous referees for their constructive comments, which helped us reorganize the paper and add important new material.

References

Abadi, M. and Cardelli, L. 1996. *A Theory of Objects*. Berlin: Springer-Verlag.

Abiteboul, S. and Bonner, A. 1991. Objects and Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 238–247.

Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. 1993. An Object Data Model with Roles. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Dublin, Ireland, pp. 39–51.

Albano, A., Cardelli, L., and Orsini, R. 1985. Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems 10,* 2, 230–260. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

Albano, A., Diotallevi, M., and Ghelli, G. 1995. Extensible Objects for Database Evolution: Language Features and Implementation Issues. In *Proc. of the Fifth Intl. Workshop on Data Base Programming Languages (DBPL), Gubbio, Italy.*

Albano, A., Ghelli, G., and Orsini, R. 1995. Fibonacci: A Programming Language for Object Databases. *The VLDB Journal 4,* 3, 403–439.

Bancilhon, F., Cluet, S., and Delobel, C. 1990. A Query Language for the O_2 Object-Oriented Database System. In R. Hull, R. Morrison, and D. Stemple (Eds.), *Proceedings of the Second International Workshop on Database Programming Languages*, San Mateo, California, pp. 122–138. Morgan Kaufmann Publishers.

Bertino, E. 1992. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob (Eds.), *Advances in Database Technology – EDBT '92*, Volume 580 of *Lectures Notes on Computer Science*, Berlin, pp. 136–151. Springer-Verlag.

Bertino, E. and Guerrini, G. 1995. Objects with Multiple Most Specific Classes. In *Proc. Ninth European Conference on Object-Oriented Programming*, LNCS N. 952, Berlin, pp. 102–126. Springer-Verlag.

Bruce, K. B., Schuett, A., and van Gent, R. 1995. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In W. Olthoff (Ed.), *Proceedings of* ECOOP '95, LNCS 952, Aarhus, Denmark, pp. 27–51. Springer-Verlag.

Cardelli, L. and Mitchell, J. 1991. Operations on Records. *Mathematical Structures in Computer Sciences 1,* 1, 3–48.

Dayal, U. 1989. Queries and Views in an Object-Oriented Data Model. In R. Hull, R. Morrison, and D. Stemple (Eds.), *Proc. of the Second Intl. Workshop on Data Base Programming Languages (DBPL), Salishan Lodge, Oregon*, San Mateo, California, pp. 80–102. Morgan Kaufmann Publishers.

dos Santos, C. S. 1994. Design and Implementation of an Object-Oriented View Mechanism. In *10èmes Journées Bases de Données Avancées*, Clermont-Ferrand, France.

dos Santos, C. S. and Waller, E. 1995. The O_2 Views User Manual - Version 2. O_2 Technology, Versailles, France.

Gottlob, G., Schrefl, M., and Röck, B. 1996. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems 14,* 3, 268–296.

Guerrini, G., Bertino, E., Catania, B., and Garcia-Molina, J. 1997. A Formal Model of Views for Object-Oriented Database Systems. *Theory and Practice of Object Systems (TAPOS) 3,* 2, 103–125.

Heiler, S. and Zdonik, S. 1990. Object Views: Extending the Vision. In *IEEE Data Engineering*, Los Angeles, pp. 86–93.

Kim, W. and Kelley, W. 1995. On View Support in Object-Oriented Database Systems. In W. Kim (Ed.), *Modern Database Systems*, pp. 108–129. Reading, Massachusetts: Addison-Wesley.

Leung, T., Mitchell, G., Subramanian, B., Vance, B., Vandenberg, S., and Zdonik, S. 1993. The AQUA Data Model and Algebra. In *Fourth Intl. Workshop on Database Programming Languages*, New York, pp. 157–175.

Motschnig-Pitrik, R. 1996. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems 21,* 3, 229–252.

Odberg, E. 1994. Category Class: Flexible Classification and Evolution in Object-Oriented Databases. In G. Wijers, S. Brinkkemper, and T. Wasserman (Eds.), *Proc. of the 6th Conference on Advanced Information Systems Engineering (CAISE '94), Utrecht, The Netherlands*, Number 811 in LNCS, Berlin, pp. 406–420. Springer-Verlag.

Ohori, A. and Tajima, K. 1994. A Polymorphic Calculus for Views and Object Sharing. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, Minneapolis, Minnesota, pp. 255–266.

Papazoglou, M. and Krämer, B. 1997. A Database Model for Object Dynamics. *The VLDB Journal* 6, 73–96.

Parent, C. and Spaccapietra, S. 1985. An Algebra for a General Entity-Relationship Model. *IEEE Transactions on Software Engineering 11,* 7, 634–643.

Richardson, J. and Schwartz, P. 1991. Aspects: Extending Objects to Support Multiple, Indipendent Roles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, pp. 298–307.

Rundensteiner, E. 1992. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. of the Eighteenth Intl. Conf. on Very Large Data Bases (VLDB), Vancouver, British Columbia, Canada*, San Mateo, California, pp. 187–198. Morgan Kaufmann Publishers.

Santos, C. D., Abiteboul, S., and Delobel, C. 1994. Virtual classes and bases. In *Advances in Database Technology – EDBT '94*.

Scholl, M., Laasch, C., Rich, C., Scheck, H., and Tresch, M. 1994. The COOCON Object Model. Technical Report 211, Departement Informatik, ETH Zurich.

Scholl, M., Laasch, C., and Tresch, M. 1990. Views in Object-Oriented Databases. In *Proc. Second Intl. Workshop on Foundations of Models and Languages for Data and Objects*, pp. 37–58.

Scholl, M. and Scheck, H. 1990. A Relational Object Model. In S. Abiteboul and P. Kanellakis (Eds.), *Proc. of the Third Intl. Conf. on Database Theory (ICDT), Paris, France*, Number 470 in LNCS, Berlin, pp. 89–105. Springer-Verlag.

Scholl, M. and Schek, H.-J. 1991. Supporting Views in Object-Oriented Databases. *IEEE Data Engineering Bulletin 14*, 2.

Shaw, G. and Zdonik, S. 1989. An Object-Oriented Query Algebra. In R. Hull, R. Morrison, and D. Stemple (Eds.), *Proc. of the Second Intl. Workshop on Data Base Programming Languages (DBPL), Salishan Lodge, Oregon*, San Mateo, California, pp. 103–112. Morgan Kaufmann Publishers.

Shilling, J. and Sweeney, P. 1989. 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*, pp. 353–361.

Wieringa, R., Jonge, W., and Spruit, P. 1995. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems (TAPOS) 1*, 1, 61–83.

Zdonik, S. 1990. Object-Oriented Type Evolution. In F. Bancilhon and P. Buneman (Eds.), *Advances in Database Programming Languages*, pp. 277–288. Reading, Massachusetts: Addison-Wesley.

Appendix A: Type rules

This section contains the type rules for the Galileo 97 subset used in the paper. Some trivial type rules are omitted, such as those for *if then else*, *iffails*, and operations on integers, booleans, strings and null values.

A formal specification of the type rules is useful for different purposes. First of all, it helps the language designer to discover whether the language is underspecified, and to spot rules which are too complex or not natural. Then, it guides the implementation of the type checker, ensuring compatibility among different implementations, and making implementation easier. There is not enough experience to assess whether programmers are ready to take advantage of a formal presentation of type rules. Our opinion is that the type rules of most of the current languages are too complex to make it possible to present them in a way which is formal, complete, and easy to understand. However, this would become possible if the importance of a clean formal presentation of the type system were kept into account when a language is designed. In this case, programmers would be able to take advantage of the type rules, as they currently take advantage of the BNF presentation of the language grammar.

Due to the specialistic nature of this section, few comments are made to explain the rules. The typing and subtyping rules have in any case been informally described in the paper.

We first define the syntactic elements which compose the good formation, typing and subtyping judgements. For the sake of simplicity, we adopt for records the set-oriented syntax $[\{A_i : T_i\}^{i\in I}]$, where we assume that every enumerated set $\{A_i : T_i\}^{i\in I}$ that we use always satisfies the property that $i \neq j \;\Rightarrow\; A_i \neq A_j$.

Environments:

Σ ::=	environment
$\emptyset$	empty environment
$\Sigma, X{:}T$	expression variable
$\Sigma, X{:=}T$	concrete type
$\Sigma, X \leftrightarrow$ *is* Y *and* $[\{A_i : T_i\}^{i\in I}]$	object type; Y may be *Top*
$\Sigma, \neg X$	variable hidden by *hide*

Declarations:

D ::=	declaration
let B	expression declaration
let rec B	recursive expression declaration
let type TB	type declaration
let rec type TB	recursive type declaration
hide X	name hiding
B ::=	binding
X *class* $Y \leftrightarrow O$	class binding
X *subset of* Z *class* $Y \leftrightarrow O$	subclass binding
$X := E$	value binding
$X :=$ *derived* E	expression binding
TB ::=	type binding
$X := T$	concrete binding
$X \leftrightarrow O$	object binding
O ::=	object type binding
$[\{A_i : T_i\}^{i\in I}; \{B_j = meth(\Sigma_j) : U_j \text{ is } E_j\}^{j\in J}]$	top object binding
is Y *and* $[\{A_i : T_i\}^{i\in I}; \{B_j = meth(\Sigma_j) : U_j \text{ is } E_j\}^{j\in J}]$	sub object binding

The judgement $\Sigma \vdash T = Obj(X)(\Sigma')$ means that T is an object type with supertype X and

signature Σ'; Σ' also contains all the fields which T inherits. When T has not been defined by inheritance, X is equal to the reserved identifier *Top*. The judgement $\Sigma \vdash D \triangleright \Sigma'$ means that the declaration D enriches the environment with the information in Σ'; the judgement $\Sigma \vdash B \triangleright \Sigma'$ is only an auxiliary judgement used to define $\Sigma \vdash D \triangleright \Sigma'$.

Judgements:

$\Sigma \vdash OK$	good environment
$\Sigma \vdash T\ OK$	good type
$\Sigma \vdash T = Obj(X)(\Sigma')$	T is an object type with supertype X and full signature Σ'
$\Sigma \vdash B \triangleright \Sigma'$	binding B produces Σ'
$\Sigma \vdash D \triangleright \Sigma'$	declaration D produces Σ'
$\Sigma \vdash E : T$	expression E has type T
$\Sigma \vdash P : T$	program P has type T
$\Sigma \vdash T \leq T'$	T is a subtype of T'
$\Sigma \vdash T \uparrow X$	T and X are object types with a common supertype

Types:

$T, U ::=$	type
X	type identifier
Top	maximal object type identifier
$[\{A_i : T_i\}^{i \in I}]$	record type
$<\{X_i\}^{i \in I} > \ view\ [\{A_l : T_l\}^{l \in L}]$	view type
$class\ X$	class type (internal use only)
$seq\ T$	sequence type
$var\ T$	updatable locations
$T_1 \# \ldots \# T_n \to U$	function type
$bool$	boolean values
$null$	the type of commands

Expressions:

$E, S, B ::=$	Expression
X	variable
$X\ In\ S$	naming sequence elements
$S\ where\ B$	selection
$select\ E\ from\ S$	mapping E to S
$[\{A_i := E_i\}^{i \in I}]$	record construction
$var\ E$	variable allocation
$at\ E$	dereferencing
$E \leftarrow E$	variable updating
$fun(X_1 : T_1, \ldots, X_n : T_n)\ is\ E$	function construction
$E(E_1, \ldots, E_n)$	function application
$E.A$	double lookup message passing
$E!A$	upward lookup message passing
$E\ As\ X$	changing role
$E\ isexactly\ X$	exact role testing
$E\ isalso\ X$	role testing
$E\ project\ [\{B'_j : U'_j\}^{j \in J'}]$	object projection
$E\ project*\ [\{B'_j : U'_j\}^{j \in J'}]$	sequence projection
$E\ extend\ [\{B_j = meth(\Sigma_j) : U_j\ is\ E_j\}^{j \in J}]$	virtual object extension
$E\ extend*\ [\{B_j = meth(\Sigma_j) : U_j\ is\ E_j\}^{j \in J}]$	sequence extension
$E\ times\ E'$	object product
$E\ times*\ E'$	sequence cartesian product
$E\ rename\ (\{A_i \Rightarrow B_i\}^{i \in I})$	object renaming

$E\ rename*\ (\{A_i \Rightarrow B_i\}^{i\in I})$	sequence renaming

Programs:

$P ::=$	program
E	expression
$D; P$	declaration plus program

Our type rules obey the discipline of static binding: names can be redefined, but the redefinition of an X does not affect whatever was in the scope of the previous definition of the same X. This is formalized by the rules which deal with variables, which prevent variable capture with conditions such as $X \notin DV(\Sigma')$ and $FV(T) \cap DV(\Sigma') = \emptyset$ in rule (Var) below (FV are the free variables, while $DV(\Sigma)$ are the variables defined in Σ, i.e. $DV(X_1 := T_1, X_2 : T_2, \neg X_3, \ldots)$ is $\{X_1, X_2, X_3, \ldots\}$). This rule alone would be too restrictive since, once a type variable X has been redefined, it would not be possible to access an expression variable $x : X$ bound to the old X. Accessing such x is made possible by rule (α renaming), which allows any consistent renaming of variables in any judgement (two judgements J and J' are α equivalent ($J =_\alpha J'$) if and only if one can be obtained from the other by a consistent renaming of its bound variables).

α renaming:

(Var)
$$\frac{\Sigma, X : T, \Sigma' \vdash OK \quad X \notin DV(\Sigma') \quad FV(T) \cap DV(\Sigma') = \emptyset}{\Sigma, X : T, \Sigma' \vdash X : T}$$

(α renaming)
$$\frac{J \qquad J =_\alpha J' \quad J' \text{ is well formed}}{J'}$$

Environments:

(empty Σ)
$$\frac{}{() \vdash OK}$$

(expression Σ)
$$\frac{\Sigma \vdash T\ OK}{\Sigma, X : T \vdash OK}$$

(concrete Σ)
$$\frac{\Sigma \vdash T\ OK}{\Sigma, X := T \vdash OK}$$

(object Σ)
$$\frac{\Sigma \vdash Y = Obj(Z)(T) \quad \Sigma \vdash T \oplus [\{A_i : T_i\}^{i\in I}] \leq T}{\Sigma, X \leftrightarrow \text{ is } Y \text{ and } [\{A_i : T_i\}^{i\in I}] \vdash OK}$$

(hide Σ)
$$\frac{\Sigma \vdash OK}{\Sigma, \neg X \vdash OK}$$

The following "$\Sigma \Rightarrow U$" notation is used in rules (Object Top) and (Object Sub) to specify how the type of a method depends on its formal parameters $A_1 : T_1, \ldots, A_n : T_n$.

$$\begin{array}{lll} (A_1 : T_1, \ldots, A_n : T_n) \Rightarrow U & = & (T_1 \# \ldots \# T_n) \to U \text{ if } n > 0 \\ () \Rightarrow U & = & U \end{array}$$

The $\oplus$ operator, used in rules (Object $\leq$) and (Extend), combines the fields of two record types, with fields in the second type overriding fields in the first with the same name.

$$\begin{array}{l} [\{A_i : T_i\}^{i\in I}] \oplus [\{B_j : U_j\}^{j\in J}] \\ \quad = \ [\{A_i : T_i \mid i \in I \wedge \forall j \in J.\ B_j \neq A_i\} \cup \{B_j : U_j\}^{j\in J}] \end{array}$$

For the sake of simplicity, in rules (Object Top) and (Object Sub) we only consider stored fields ($A : T$) and methods ($A := meth \ldots$). Constant fields ($A := E$) can be considered, for the purpose of typing, as parameterless methods ($A := meth()\ is\ T$).

Declarations:

(let)
$$\frac{\Sigma \vdash B \ \triangleright \ \Sigma'}{\Sigma \vdash \mathit{let}\ B \ \triangleright \ \Sigma'}$$

(let rec)
$$\frac{\Sigma, \Sigma' \vdash B \ \triangleright \ \Sigma'}{\Sigma \vdash \mathit{let\ rec}\ B \ \triangleright \ \Sigma'}$$

(hide)
$$\frac{\Sigma \vdash \mathit{OK}}{\Sigma \vdash \mathit{hide}\ X \ \triangleright \ \neg X}$$

(let type)
$$\frac{\Sigma \vdash \mathit{TB} \ \triangleright \ \Sigma'}{\Sigma \vdash \mathit{let\ type}\ \mathit{TB} \ \triangleright \ \Sigma'}$$

(let rec type)
$$\frac{\Sigma, \Sigma' \vdash \mathit{TB} \ \triangleright \ \Sigma'}{\Sigma \vdash \mathit{let\ rec\ type}\ \mathit{TB} \ \triangleright \ \Sigma'}$$

(value)
$$\frac{\Sigma \vdash E : T}{\Sigma \vdash X := E \ \triangleright \ X{:}T}$$

(derived)
$$\frac{\Sigma \vdash E : T}{\Sigma \vdash X := \mathit{derived}\ E \ \triangleright \ X{:}T}$$

(concrete)
$$\frac{\Sigma \vdash T \ \mathit{OK}}{\Sigma \vdash X := T \ \triangleright \ X{:=}T}$$

(object top)
$$\frac{\forall i \in I.\ \forall j \in J.\ A_i \neq B_j \qquad \forall j \in J.\ \Sigma, \Sigma_j, \mathit{self}{:}X \vdash E_j : U_j}{\begin{array}{l}\Sigma \vdash X \leftrightarrow [\{A_i : T_i\}^{i\in I}; \{B_j = \mathit{meth}(\Sigma_j) : U_j \ \mathit{is}\ E_j\}^{j\in J}] \\ \quad \triangleright \ X \leftrightarrow \ \mathit{is}\ \ \mathit{Top}\ \mathit{and}\ [\{A_i : T_i\}^{i\in I}; \{B_j : \Sigma_j \Rightarrow U_j\}^{j\in J}], \\ \quad\ \mathit{mkX} : \ ([\{A_i : T_i\}^{i\in I}]) \to X, \\ \quad\ \mathit{dropX} : \ X \to \mathit{null}\end{array}}$$

(object sub)
$$\frac{\begin{array}{c}\forall i \in I.\ \forall j \in J.\ A_i \neq B_j \quad \Sigma \vdash Y = \mathit{Obj}(Z)(\{A'_k : T'_k\}^{k\in K}) \\ \Sigma \vdash \mathit{mkY} : \ (\{A''_w : T''_w\}^{w\in W}) \to Y \\ \forall i \in I.\ \forall k \in K.\ A'_k = A_i \ \Rightarrow \ T_i \leq T'_k \\ \forall j \in J.\ \forall k \in K.\ A'_k = B_j \ \Rightarrow \ \Sigma_J \Rightarrow U_j \leq T'_k \\ \Sigma, \Sigma_j, \mathit{self}{:}X, \mathit{super}{:}Y \vdash E_j : U_j\end{array}}{\begin{array}{l}\Sigma \vdash X \leftrightarrow \ \mathit{is}\ Y\ \mathit{and}\ [\{A_i : T_i\}^{i\in I}; \{B_j = \mathit{meth}(\Sigma_j) : U_j \ \mathit{is}\ E_j\}^{j\in J}] \\ \quad \triangleright \ X \leftrightarrow \ \mathit{is}\ Y\ \mathit{and}\ [\{A'_k : T'_k\}^{k\in K}] \oplus [\{A_i : T_i\}^{i\in I}; \{B_j : \Sigma_j \Rightarrow U_j\}^{j\in J}], \\ \quad\ \mathit{mkX} : \ ([\{A''_w : T''_w\}^{w\in W}] \oplus [\{A_i : T_i\}^{i\in I}]) \to X, \\ \quad\ \mathit{InX} : \ (Y \# [\{A_i : T_i\}^{i\in I}]) \to X, \\ \quad\ \mathit{dropX} : \ X \to \mathit{null}\end{array}}$$

(class)
$$\frac{\Sigma \vdash X \leftrightarrow O \ \triangleright \ \Sigma'}{\Sigma \vdash Y\ \mathit{class}\ X \leftrightarrow O \ \triangleright \ Y : \mathit{class}\ X,\ \Sigma'}$$

(subclass)
$$\frac{\Sigma \vdash X \leftrightarrow O \ \triangleright \ \Sigma' \qquad \Sigma, \Sigma' \vdash X = \mathit{Obj}(X')(T)\ (X' \neq \ \mathit{Top}) \qquad \Sigma \vdash Z : \ \mathit{class}\ Z' \qquad \Sigma \vdash X' \leq Z'}{\Sigma \vdash Y\ \mathit{subset\ of}\ Z\ \mathit{class}\ X \leftrightarrow O \ \triangleright \ Y : \mathit{class}\ X,\ \Sigma'}$$

Types:

(good formation)
$$\frac{\Sigma \vdash T \leq T}{\Sigma \vdash T \ \mathit{OK}}$$

(object)
$$\frac{\Sigma, X \leftrightarrow \ \mathit{is}\ Y\ \mathit{and}\ T, \Sigma' \vdash \mathit{OK} \quad X, Y \notin \mathit{DV}(\Sigma') \quad \mathit{FV}(T) \cap \mathit{DV}(\Sigma') = \emptyset}{\Sigma, X \leftrightarrow \ \mathit{is}\ Y\ \mathit{and}\ T, \Sigma' \vdash X = \mathit{Obj}(Y)(T)}$$

Notation:

$$\Sigma \vdash T \sim U \quad stands\ for \quad \Sigma \vdash T \leq U \ \wedge\ \Sigma \vdash U \leq T$$

Subtypes:

(trans)
$$\frac{\Sigma \vdash T \leq U \quad \Sigma \vdash U \leq V}{\Sigma \vdash T \leq V}$$

(concrete $\leq$)
$$\frac{\Sigma, X{:=}T, \Sigma' \vdash OK \quad X \notin DV(\Sigma') \quad FV(T) \cap DV(\Sigma') = \emptyset}{\Sigma, X{:=}T, \Sigma' \vdash X \sim T}$$

(concrete-refl$_\leq$)
$$\frac{\Sigma \vdash OK \quad X{:=}T \in \Sigma}{\Sigma \vdash X \leq X}$$

(object$_\leq$)
$$\frac{\Sigma \vdash X = Obj(Y)(T)}{\Sigma \vdash X \leq T}$$

(object-refl$_\leq$)
$$\frac{\Sigma \vdash X = Obj(Y)(T)}{\Sigma \vdash X \leq X}$$

(sub object$_\leq$)
$$\frac{\Sigma \vdash X = Obj(Y)(T) \quad Y \neq Top}{\Sigma \vdash X \leq Y}$$

(record$_\leq$)
$$\frac{\forall j \in J'.\ \Sigma \vdash U'_j\ OK \quad \forall j \in J.\ \exists j' \in J'.\ B'_{j'} = B_j \ \wedge\ \Sigma \vdash U'_{j'} \leq U_j}{\Sigma \vdash [\{B'_j : U'_j\}^{j \in J'}] \ \leq\ [\{B_j : U_j\}^{j \in J}]}$$

(view$_\leq$)
$$\frac{\begin{array}{c}\forall i \in I.\ \Sigma \vdash X_i = Obj(Y_i)(\Sigma'_i) \quad \forall i \in I'.\ \Sigma \vdash X'_i = Obj(Y'_i)(\Sigma''_i) \\ \forall j \in J'.\ \Sigma \vdash U'_j\ OK \\ \forall i \in I.\ \exists i' \in I'.\ X'_{i'} \leq X_i \quad \forall j \in J.\ \exists j' \in J'.\ B'_{j'} = B_j \ \wedge\ \Sigma \vdash U'_{j'} \leq U_j\end{array}}{\Sigma \vdash <\{X'_i\}^{i \in I'} > \ view\,[\{B'_j : U'_j\}^{j \in J'}] \ \leq\ <\{X_i\}^{i \in I} > \ view\,[\{B_j : U_j\}^{j \in J}]}$$

(view-rec$_\leq$)
$$\frac{\forall j \in J.\ \Sigma \vdash U_j\ OK}{\Sigma \vdash [\{B_j : U_j\}^{j \in J}] \ \sim\ <> \ view\,[\{B_j : U_j\}^{j \in J}]}$$

(view-obj$_\leq$)
$$\frac{\Sigma \vdash X = Obj(Y)([\{B_j : U_j\}^{j \in J}])}{\Sigma \vdash X \ \sim\ <X> \ view\,[\{B_j : U_j\}^{j \in J}]}$$

(seq $\leq$)
$$\frac{\Sigma \vdash T \leq U}{\Sigma \vdash seq\,T \leq seq\,U}$$

(class-seq $\leq$)
$$\frac{\Sigma \vdash T \leq U}{\Sigma \vdash class\,T \leq seq\,U}$$

(var $\leq$)
$$\frac{\Sigma \vdash T\ OK}{\Sigma \vdash var\,T \leq var\,T}$$

(function $\leq$)
$$\frac{\Sigma \vdash T_1 \leq T'_1\ \ldots\ \Sigma \vdash T_n \leq T'_n \quad \Sigma \vdash U' \leq U}{\Sigma \vdash T'_1 \# \ldots \# T'_n \to U' \leq T_1 \# \ldots \# T_n \to U}$$

Functions, variables, subsumption

(var)
$$\frac{\Sigma \vdash E :\ T}{\Sigma \vdash var\,E :\ var\,T}$$

(at)
$$\frac{\Sigma \vdash E :\ var\,T}{\Sigma \vdash at\,E :\ T}$$

(update)
$$\frac{\Sigma \vdash L :\ var\,T \quad \Sigma \vdash E :\ T}{\Sigma \vdash L \leftarrow E :\ null}$$

(function)
$$\frac{\Sigma, X_1{:}T_1, \ldots, X_n{:}T_n \vdash E : U}{\Sigma \vdash fun(X_1{:}T_1, \ldots, X_n{:}T_n) \ is \ E : (T_1\#\ldots,\#T_n) \to U}$$

(application)
$$\frac{\Sigma \vdash E : (T_1\#\ldots,\#T_n) \to U \quad \Sigma \vdash E_1 : T_1 \quad \ldots \quad \Sigma \vdash E_n : T_n}{\Sigma \vdash E(E_1, \ldots, E_n) : U}$$

(subsumption)
$$\frac{\Sigma \vdash E : T \quad \Sigma \vdash T \leq U}{\Sigma \vdash E : U}$$

We now define a renaming operator on record types:

$$\rho(\{A_i \Rightarrow B_i\}^{i \in I})[\{A''_l : T_l\}^{l \in L}]$$

Renaming:

$$\rho(\{A_i \Rightarrow B_i\}^{i \in I})[\{A''_l : T_l\}^{l \in L}]$$
$$= [\{B_i : T_l \mid l \in L, A_i = A''_l\} \cup \{A''_l : T_l \mid l \in L, \forall i \in I.\ B_i \neq A''_l\}]$$

In the next section we do not define the type rules for the "lifted" operators *extend**, *rename**, and *times**; we only give, as an example, the rule for *project**; the other rules are defined exactly in the same way.

Record, object, and view operations:

(record)
$$\frac{\forall i \in I.\ \Sigma \vdash E_i : T_i}{\Sigma \vdash [\{A_i := E_i\}^{i \in I}] : [\{A_i : T_i\}^{i \in I}]}$$

(dot) (!)
$$\frac{\Sigma \vdash E : [A : T]}{\Sigma \vdash E.A : T} \qquad \frac{\Sigma \vdash E : <X> \ view \ [A : T]}{\Sigma \vdash E!A : T}$$

(↑)
$$\frac{\exists i \in I.\ \Sigma \vdash X_i \leq Y \quad \Sigma \vdash X \leq Y \quad \Sigma \vdash Y = Obj(Z)(U)}{\Sigma \vdash (<\{X_i\}^{i \in I}> \ view \ T) \uparrow X}$$

(As)
$$\frac{\Sigma \vdash E : <\{X_i\}^{i \in I}> \ view \ T \quad \Sigma \vdash (<\{X_i\}^{i \in I}> \ view \ T) \uparrow X'}{\Sigma \vdash E \ As \ X' : X'}$$

(isexactly)
$$\frac{\Sigma \vdash E : <\{X_i\}^{i \in I}> \ view \ T \quad \Sigma \vdash (<\{X_i\}^{i \in I}> \ view \ T) \uparrow X'}{\Sigma \vdash E \ isexactly \ X' : bool}$$

(isalso)
$$\frac{\Sigma \vdash E : <\{X_i\}^{i \in I}> \ view \ T \quad \Sigma \vdash (<\{X_i\}^{i \in I}> \ view \ T) \uparrow X'}{\Sigma \vdash E \ isalso \ X' : bool}$$

(project)
$$\frac{\Sigma \vdash E : <\{X_i\}^{i \in I}> \ view \ [\{B_j : U_j\}^{j \in J}] \quad \forall j \in J'.\ j \in J \wedge \Sigma \vdash U_j \leq U'_j}{\Sigma \vdash E \ project \ [\{B'_j : U'_j\}^{j \in J'}] : <\{X_i\}^{i \in I}> \ view \ [\{B'_j : U'_j\}^{j \in J'}]}$$

(project*)
$$\frac{\Sigma \vdash E : \ seq(<\{X_i\}^{i \in I}> \ view \ [\{B_j : U_j\}^{j \in J}]) \quad \forall j \in J'.\ j \in J \wedge \Sigma \vdash U_j \leq U'_j}{\Sigma \vdash E \ project* \ [\{B'_j : U'_j\}^{j \in J'}] : \ seq(<\{X_i\}^{i \in I}> \ view \ [\{B'_j : U'_j\}^{j \in J'}])}$$

(extend)

$$\frac{\begin{array}{c}\forall i \in I.\ \Sigma : D_i : T_i \\ let\ T = <\{X_k\}^{k\in K} >\ view\ [\{A_l : T_l\}^{l\in L}] \oplus [\{A_i : T_i\}^{i\in I}; \{B_j : \Sigma_j \Rightarrow U_j\}^{j\in J}] \\ \Sigma \vdash E : <\{X_k\}^{k\in K} >\ view\ [\{A_l : T_l\}^{l\in L}] \\ \forall i \in I.\ \forall j \in J.\ A_i \neq B_j \quad \forall j \in J.\ \Sigma, \Sigma_j, me{:}T \vdash E_j :\ U_j\end{array}}{\Sigma \vdash E\ extend\ [\{A_i := D_i\}^{i\in I}; \{B_j = meth(\Sigma_j) : U_j\ is\ E_j\}^{j\in J}] :\ T}$$

(times)

$$\frac{\begin{array}{c}\Sigma \vdash E : <\{X_i\}^{i\in I} >\ view\ [\{A_l : T_l\}^{l\in L}] \\ \Sigma \vdash E' : <\{X'_i\}^{i\in I'} >\ view\ [\{A'_l : T'_l\}^{l\in L'}] \\ \forall l \in L.\ \forall l' \in L'.\ A_l \neq A'_{l'}\end{array}}{\Sigma \vdash E\ times\ E' : <\{X_i\}^{i\in I} \cup \{X'_i\}^{i\in I'} >\ view\ [\{A_l : T_l\}^{l\in L} \cup \{A'_l : T'_l\}^{l\in L'}]}$$

(rename)

$$\frac{\begin{array}{c}\Sigma \vdash E : <\{X_i\}^{i\in I} >\ view\ [\{A_l : T_l\}^{l\in L}] \\ \Sigma \vdash \rho(\{A_i \Rightarrow B_i\}^{i\in I})[\{A_l : T_l\}^{l\in L}]\ OK\end{array}}{\begin{array}{l}\Sigma \vdash E\ rename\ (\{A_i \Rightarrow B_i\}^{i\in I}) \\ \quad : <\{X_i\}^{i\in I} >\ view\ \rho(\{A_i \Rightarrow B_i\}^{i\in I})[\{A_l : T_l\}^{l\in L}]\end{array}}$$

In the following rules, observe that, by subsumption and by rules (view-rec$_{\leq}$), (view-obj$_{\leq}$), every record and every real object also has a view type. This implies that the following rules can all be applied to sequences of records, real and virtual objects alike. Moreover, by rule (class-seq $_{\leq}$), every class has a sequence type, which makes the following rules applicable to classes too.

Query operations:

(In)

$$\frac{\Sigma \vdash S :\ seq\, T}{\Sigma \vdash X\ In\ S :\ seq\,[X : T]}$$

(where)

$$\frac{\Sigma \vdash S :\ seq\,(<\{X_i\}^{i\in I} >\ view\,[\Sigma'])\quad \Sigma, \Sigma' \vdash B : bool}{\Sigma \vdash S\ where\ B :\ seq\,(<\{X_i\}^{i\in I} >\ view\,[\Sigma'])}$$

(select)

$$\frac{\Sigma \vdash S :\ seq\,(<>\ view\,[\Sigma'])\quad \Sigma, \Sigma' \vdash E : T}{\Sigma \vdash select\ E\ from\ S :\ seq\, T}$$

Programs:

(program)

$$\frac{\Sigma \vdash D\ \triangleright\ \Sigma' \quad \Sigma, \Sigma' \vdash E :\ T}{\Sigma \vdash D; E :\ T}$$

Appendix B: Operational Semantics

This section contains the operational semantics for a subset of the Galileo 97.

The operational semantics may be used for different aims. First of all, it is a specification for an implementor: once the semantics has been formally defined, the implementation task becomes much easier, and it becomes possible to produce different implementations which behave essentially in the same way.

The semantics may also be used to prove that the Galileo 97 type system is safe, i.e. that well-typed terms never go wrong; this theorem is called "static strong typing".[7] The techniques to prove this kind of result are known, but the dimension of the Galileo 97 language make this task quite difficult.

In this section, as in the previous one, we will only give the minimal amount of comments which make the formalization possible to understand.

The semantics of some operators, such as *times*, *selectfrom*, and *where*, depends on some information which is collected at compile-time. This is modelled by defining the semantics over a language of run-time expressions, which coincide with Galileo 97 expressions with the following exceptions.

Run-time expressions:

$E, S, B ::=$	Run-time expression
...	...
$S\ where^{X_1..X_n}\ B$	selection, where $X_1, \ldots, X_n$ are fields of S elements and may be accessed in B
$select^{X_1..X_n}\ E\ from\ S$	mapping E to S
$E\ times^{X_1..X_n, X'_1..X'_m}\ E'$	object product
$E\ times*^{X_1..X_n, X'_1..X'_m}\ E'$	sequence cartesian product
...	...

To connect Galileo 97 with run-time expressions we define a judgement $\Sigma \vdash E \rightarrow E' : T$ which specifies how an expression E is adorned with the information needed to produce the run-time expression E'. We only specify one of the rules for this judgement; the other are similar, and are obtained by modifying the corresponding type rule in the same way as in this case.

Run-time expressions:

(run-time select)

$$\frac{\Sigma \vdash S \rightarrow S' : seq(<> view[X_1 : T_1, \ldots, X_n : T_n]) \qquad \Sigma, X_1 : T_1, \ldots, X_n : T_n \vdash E \rightarrow E' : T}{\Sigma \vdash select\ E\ from\ S \rightarrow select^{X_1..X_n}\ E'\ from\ S' : seq\ T}$$

Now the semantics is defined by judgements with the shape $C, \sigma, E \rightarrow l, \sigma'$. Here, C is a context, i.e. a function from variable names to locations, σ is a store, i.e. a function from locations to storable values, and E is a run-time expression. The evaluation of an expression, with respect to a context and a store, yields a modified store σ' and a location l, which may be thought of as a pointer to the value of E. The context is used to evaluate the identifiers: $C, \sigma, X \rightarrow C(X), \sigma$. The store is used to represent any complex data structure. For example, an object role is represented by a location l which points to a quintuple. This quintuple contains:

—a tag "*OR*" which says that it represents an object,

—the object type l_X,

[7] In the following definition of the Galileo 97 operational semantics, static strong typing does not hold, since message passing to removed objects may fail, and the inX operation may fail too, when an object already possesses the type X. Hence, we should add two reductions to *known-error* for those two cases, and prove that no other errors may be generated.

—its method suite F,

—a pointer to the object history l_h,

—the object status.

This is formalized as

$$\sigma(l) = OR{<}l_X, F, l_h, status{>}.$$

The object history is a sequence of pairs type-role pointer, tagged by "$H()$". For example, if two roles l_1 and l_2 represent, in σ, the two roles of an object, then the following conditions hold:

$$\begin{array}{lcl} \sigma(l_1) & = & OR{<}l_X, F, l_h, valid{>} \\ \sigma(l_2) & = & OR{<}l_Y, G, l_h, valid{>} \\ l_h & = & H(<l_X, l_1> \cdot <l_Y, l_2>) \end{array}$$

Essentially, a location l represents an arrow in the store model, while $\sigma(l)$ represents the box pointed by that arrow, which may contain other pointers. Hence the three equations above can be represented by the picture in Figure 11 (see also Figure 2).

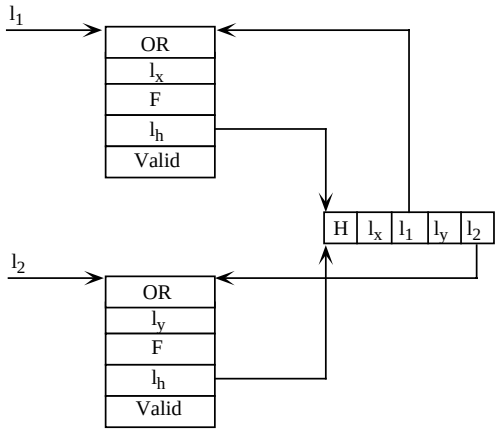

Fig. 11. The graphical representation of a store configuration.

The method suite F is represented by a record of closures, i.e. by a function which associates to a finite set of labels a set of methods; we write this fact as:

$F \in X \xrightarrow{fin} L$	*F maps labels to locations.*
$\forall A \in X.\ F(A)\downarrow$	*If $F(A)$ is defined,*
$\Rightarrow\ \sigma(F(A)) = M{<}C', \{X_i\}^{i\in I}, E{>}$	*then $F(A)$ points to a method.*

Here, $F(A)\downarrow$ means that F is defined on A, i.e. that A is a label in the record F. $M{<}C', \{X_i\}^{i\in I}, E{>}$ is the representation of a method in the store: M is the tag for those closures which represent methods, C' is the context as it was when the method with body E has been defined; it will be used to evaluate the non local identifiers in E. $\{X_i\}^{i\in I}$ are the formal parameters of the method, which must be bound to the actual parameters before evaluating E.

A store σ is a finite partial function from the set of locations L to the set of all storable values $\mathcal{SV}$, as defined by the following grammar. In this grammar, the metavariables L, C, E, and F range, respectively, over locations, contexts, expressions, and "semantic records", i.e. finite partial functions from identifiers to something else. σ is also required to be defined over four locations, l_{nil}, l_{true}, l_{false}, l_{Top}, which are required to be pairwise different. We do noy deal here with the representation of integers.

The storable values

$\mathcal{SV}$ ::=	Storable values

$S(L_1 \cdots L_n)$	a sequence
$R(F)$	a record, i.e. a finite function from identifiers to values (locations) ($F : X \stackrel{fin}{\rightharpoonup} L$)
$V(L)$	a variable (updatable location)
$F,M<C, \{X_i\}^{i \in I} E>$	a function (F) or a method (M) defined in a context (C), with parameters $\{X_i\}^{i \in I}$ and body E
$D<C, E>$	a derived values defined in a context C
$OR<L, F, L', \{valid \mid removed\}>$	a role with its type L, method suite $F : X \stackrel{fin}{\rightharpoonup} L$, history L', and status
$H(<L_1, L_1'> \cdots <L_n, L_n'>)$	a history, i.e. an ordered sequence of type-role pairs (the first element is the last acquired role)
$VE<F, L>$	a virtual extended object, with new methods and a reference to the base object
$VR<F, L>$	a virtual renamed object, with name mapping $F : X \stackrel{fin}{\rightharpoonup} X$ and a reference to the base object
$VT<F, L, L>$	a virtual times object, with name mapping $F : X \stackrel{fin}{\rightharpoonup} \{0, 1\}$ and references to two base objects
$T<X, L>$	$\sigma(l_X) = T<X, l_Y>$ means that the type represented by l_X is a subtype of the one represented by l_Y; X is its name

The semantics is defined by four judgements.

(1) The expression evaluation judgement $C, \sigma, E \ \rightarrow \ l, \sigma'$, described above.

(2) The declaration judgement $C, \sigma, D \ \rightarrow \ C', \sigma'$, which means that the evaluation of the declaration D transforms σ into σ' and produces a context C'; this judgements is related to the previous one by the rule (program).

(3) The auxiliary judgement $C, \sigma, l(l_1, \ldots, l_n) \ \stackrel{?A}{\rightarrow} \ l', \sigma'$, where ? may be either . or !, used to evaluate the effect of a message passing operation $E?A(E_1, \ldots, E_n)$ when E evaluates to l and E_i evaluates to l_i.

(4) The auxiliary judgement $l_X \leq_\sigma l_Y$, which returns true if the object type represented by l_X in σ is a subtype of the one represented by l_Y.

Contexts:

C ::=	Context
$\emptyset$	empty context
$\Sigma, X{:=}l$	X is bound to l

Judgements:

$C, \sigma, D \ \rightarrow \ C', \sigma'$	D, in context C and store σ, produces C', σ'
$C, \sigma, E \ \rightarrow \ l, \sigma'$	E, in context C and store σ, evaluates to l and produces σ'
$C, \sigma, l(l_1, \ldots, l_n) \ \stackrel{?A}{\rightarrow} \ l, \sigma'$	the message $?A$ (where "?" is either "." or "!") sent to l with parameters $l_1, \ldots, l_n$ yields l, σ'
$l_X \leq_\sigma l_Y$	l_X represents a subtype of l_Y

Programs:

(program)

$$\frac{C, \sigma, D \ \rightarrow \ C', \sigma' \quad C', \sigma', E \ \rightarrow \ l, \sigma''}{C, \sigma, (D; E) \ \rightarrow \ l, \sigma''}$$

We first define the subtyping judgement. l represents an immediate subtype of the object type represented by l' when l refers to a pair $T<X, l'>$. These pairs are stored when

object type definitions are evaluated (see rules (object top) and (object sub)). Subtyping is reflexive and transitive.

Inclusion:

$$\frac{\sigma(l) = T{<}X, l_Y{>}}{l \leq_\sigma l} \text{(Refl } \leq\text{)} \qquad \frac{l \leq_\sigma l' \quad l' \leq_\sigma l''}{l \leq_\sigma l''} \text{(Trans } \leq\text{)} \qquad \frac{\sigma(l) = T{<}X, l_Y{>}}{l \leq_\sigma l_Y} \text{(Base } \leq\text{)}$$

Before presenting the rules which govern message passing, let us define two auxiliary functions, *down* and *up*. $down^{l_X}_\sigma(A)(H)$ searches in the history H for the last acquired role whose type is a subtype of l_X and whose method suite has a method for A, and returns the role and the method for A. $up^{l_X}_\sigma(A)(H)$ searches in the history H for the first role whose type is a supertype of l_X and whose method suite has a method for A, and returns the method only.

$$\begin{aligned}
&down_\sigma(A)(\emptyset) && = \uparrow \text{ (undefined)}\\
&down^{l_X}_\sigma(A)(H({<}l_Y, l_{obj}{>}\cdot Rest)) && = {<}l_{obj}, l_{meth}{>}\\
&&& \text{if } l_Y \leq l_X,\ \sigma(l_{obj}) = OR{<}_, F, _, _{>},\ F(A) = l_{meth}\\
&down^{l_X}_\sigma(A)(H({<}l_Y, l_{obj}{>}\cdot Rest)) && = down^{l_X}_\sigma(A)(H(Rest))\\
&&& \text{if } l_Y \leq l_X,\ \sigma(l_{obj}) = OR{<}_, F, _, _{>},\ F(A) = \uparrow
\end{aligned}$$

$$\begin{aligned}
&up_\sigma(A)(\emptyset) && = \uparrow \text{ (undefined)}\\
&up^{l_X}_\sigma(A)(H({<}l_Y, l_{obj}{>}\cdot Rest)) && = l_{meth}\\
&&& \text{if } l_X \leq l_Y,\ \sigma(l_{obj}) = OR{<}_, F, _, _{>},\ F(A) = l_{meth}\\
&up^{l_X}_\sigma(A)(H({<}l_Y, l_{obj}{>}\cdot Rest)) && = up^{l_X}_\sigma(A)(H(Rest))\\
&&& \text{if } l_X \leq l_Y,\ \sigma(l_{obj}) = OR{<}_, F, _, _{>},\ F(A) = \uparrow
\end{aligned}$$

Message sending:

When (a) l is a real object whose run-time role type is l_X and (b) a search in its history reveals a method for A in a subrole l_{obj}, then that method is evaluated, in a context where *self* is not bound to l but to l_{obj} (for the sake of simplicity the binding of *super* is not considered here).

(real object, down method)

$$\frac{\begin{array}{l}\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad down^{l_X}_\sigma(A)(\sigma(l_h)) = {<}l_{obj}, l_{meth}{>}\\ \sigma(l_{meth}) = M{<}C_1, \{X_i\}^{i\in I}, E{>}\\ (C_1 \oplus \{X_i \mapsto l_i\}^{i\in I} \oplus self \mapsto l_{obj}), \sigma, E \ \rightarrow\ l', \sigma'\end{array}}{C, \sigma, l(l_1, \ldots, l_n) \ \stackrel{A}{\rightarrow}\ l', \sigma'}$$

If (a) l is real, (b) the "method" is found in the downward search, and (c) it is just a stored function, then it is evaluated without any self parameter.

(real object, down function)

$$\frac{\begin{array}{l}\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad down^{l_X}_\sigma(A)(\sigma(l_h)) = {<}l_{obj}, l_{meth}{>}\\ \sigma(l_{meth}) = F{<}C_1, \{X_i\}^{i\in I}, E{>}\\ (C_1 \oplus \{X_i \mapsto l_i\}^{i\in I}), \sigma, E \ \rightarrow\ l', \sigma'\end{array}}{C, \sigma, l(l_1, \ldots, l_n) \ \stackrel{A}{\rightarrow}\ l', \sigma'}$$

If (a) l is real, (b) the "method" is found by downward search, (c) it is just a stored field, and (d) there are no parameters, then the stored field is returned with no further evaluation.

(real object, down field)

$$\frac{\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad down_\sigma^{l_X}(A)(\sigma(l_h)) = {<}l_{obj}, l_{meth}{>} \quad \sigma(l_{meth}) \neq M{<}\ldots{>}}{C, \sigma, l() \xrightarrow{\cdot A} l_{meth}, \sigma}$$

When downward search fails, then upward search must be used.

(real object, down to up)

$$\frac{\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad down_\sigma^{l_X}(A)(\sigma(l_h)) = \uparrow \quad C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{!A} l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{\cdot A} l', \sigma'}$$

When the method is found during upward search, then *self* is bound to l.

(real object, up method)

$$\frac{\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad up_\sigma^{l_X}(A)(\sigma(l_h)) = l_{meth} \quad \sigma(l_{meth}) = M{<}C_1, \{X_i\}^{i \in I}, E{>} \quad (C_1 \oplus \{X_i \mapsto l_i\}^{i \in I} \oplus self \mapsto l), \sigma, E \rightarrow l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{!A} l', \sigma'}$$

(real object, up function)

$$\frac{\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad up_\sigma^{l_X}(A)(\sigma(l_h)) = l_{meth} \quad \sigma(l_{meth}) = F{<}C_1, \{X_i\}^{i \in I}, E{>} \quad (C_1 \oplus \{X_i \mapsto l_i\}^{i \in I}), \sigma, E \rightarrow l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{!A} l', \sigma'}$$

(real object, up field)

$$\frac{\sigma(l) = OR{<}l_X, \_, l_h, valid{>} \quad up_\sigma^{l_X}(A)(\sigma(l_h)) = l_{meth} \quad \sigma(l_{meth}) \neq M{<}\ldots{>}}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{!A} l_{meth}, \sigma}$$

When the object is a virtual extended object with a method suite F, there are two possibilities. If the method is defined in F, then it is executed, binding me with l. Otherwise, the method is delegated to the base object l_b.

(Extended object 1)

$$\frac{\sigma(l) = VE{<}F, \_{>} \quad F(A) = M{<}C_1, \{X\}^{i \in I}, E{>} \quad (C_1 \oplus \{X_i \mapsto l_i\}^{i \in I} \oplus me \mapsto l), \sigma, E \rightarrow l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}$$

(Extended object 2)

$$\frac{\sigma(l) = VE{<}F, l_b{>} \quad F(A) = \uparrow \quad C, \sigma, l_b(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}$$

When the object is a virtual renamed object with a renaming function F, and A is one of the renamed attributes, then the call to A becames a call to $F(A)$. Otherwise, the call is delegated to the base object l_b.

(Renamed object 1)

$$\frac{\sigma(l) = VR{<}F, l_b{>} \quad F(A) = A' \quad C, \sigma, l_b(l_1, \ldots, l_n) \xrightarrow{?A'} l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}$$

(Renamed object 2)

$$\frac{\sigma(l) = VR{<}F, l_b{>} \quad F(A) = \uparrow \quad C, \sigma, l_b(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \xrightarrow{?A} l', \sigma'}$$

When the object is a times-built virtual object, its F component is used to redirect a message to one of its two base objects.

(Times)

$$\frac{\sigma(l) = T{<}F, l_{b0}, l_{b1}{>} \quad F(A) = i \quad C, \sigma, l_{bi}(l_1, \ldots, l_n) \stackrel{?A}{\rightarrow} l', \sigma'}{C, \sigma, l(l_1, \ldots, l_n) \stackrel{?A}{\rightarrow} l', \sigma'}$$

We give some examples of the effect of declarations. We start with two simple cases, and then define the crucial cases (object top) and (object sub). Class declarations are not presented.

Declarations:

(value)

$$\frac{C, \sigma, E \rightarrow l, \sigma'}{C, \sigma, let\ X := E \rightarrow (C \oplus X \mapsto l), \sigma'}$$

A *derived* declaration creates a new structure in the store. The function $newloc^n(\sigma) =< \sigma', l_1, \ldots, l_n >$ takes a store σ and returns n unused locations and a store σ' which is σ extended with arbitrary values for these new locations. The *derived* declaration binds X with such a new location l into the context, and binds l with a closure into σ' (but σ may be used as well). This closure will be evaluated every time X is accessed (see rule (ide derived) in the section on functions, variables, identifiers). As usual in language with static scoping, the closure stores the current context C which will be used when E will be evaluated.

(derived)

$$\frac{newloc^1(\sigma) =< \sigma', l >}{C, \sigma, let\ X := derived\ E \rightarrow (C \oplus X \mapsto l), (\sigma' \oplus l \mapsto D{<}C, E{>})}$$

An object type declaration creates four new structures. The first records that X is a subtype of l_{Top}. The second contains a closure which is associated with mkX. This closure receives a formal parameter called $\$x$, and executes an "internal" operation $\$mk$ which takes, as parameters, the type X and a record which associates to each message either a value ($\$x.A_i$) or a method ($\$m.B_j$).[8] The definitions of the methods are collected in a record l_{meth}. $dropX$ is associated to a function which executes the internal operation $\$drop$ (we will later define the semantics of $\$mk$, but not that of $\$drop$). $vars(\Sigma)$ returns the sequence of the parameter names in the signature Σ.

(object top)

$$\frac{newloc^4(\sigma) =< \sigma', l_X, l_{mkX}, l_{meth}, l_{dropX} >}{\begin{array}{l} C, \sigma, let\ X \leftrightarrow [\{A_i : T_i\}^{i \in I}; \{B_j = meth(\Sigma_j) : U_j\ is\ E_j\}^{j \in J}] \\ \rightarrow \quad C^+ \equiv (C \oplus X \mapsto l_X \oplus mkX \mapsto l_{mkX} \oplus dropX \mapsto l_{dropX}), \\ \quad \sigma' \oplus l_X \mapsto T{<}X, l_{Top}{>} \\ \quad \oplus\ l_{mkX} \mapsto F{<}\ (C^+ \oplus \$m \mapsto l_{meth}), \\ \qquad\qquad \{\$x\}, \$mk(X, [\{A_i := \$x.A_i\}^{i \in I}; \{B_j := \$m.B_j\}^{j \in J}]){>} \\ \quad \oplus\ l_{meth} \mapsto R(\{B_j \mapsto M{<}C^+, vars(\Sigma_j), E_j{>}\}^{j \in J}) \\ \quad \oplus\ l_{dropX} \mapsto F{<}\emptyset, \{\$x\}, \$drop(X, \$x){>} \end{array}}$$

The operation $\$mk$ is the one which actually builds an object. It takes a type l_X, a method suite l_R, and a new location l_h and uses them to build an object role which contains that

[8] Note that $\$m$ is an identifier which we must associate with the location l_{meth}, since the $\$mk(\ldots)$ component of the closure is a Galileo expression, which needs to go through an identifier to be able to access a value.

method suite and whose history contains only the role itself.

($\$mk$)

$$\frac{C,\sigma,X \;\rightarrow\; l_X,\sigma' \quad C,\sigma',E \;\rightarrow\; l_R,\sigma'' \quad \sigma''(l_R) = R(F) \quad newloc^2(\sigma'') =< \sigma''', l, l_h >}{C,\sigma,\$mk(X,E) \;\rightarrow\; l,(\sigma''' \oplus\; l \mapsto OR{<}l_X, F, l_h, valid{>} \oplus\; l_h \mapsto H(<l_X, l>))}$$

A subtype declaration expands $mkY(x)$ simply into $InX(mkY(x), x)$. The *In* operator is dealt with by an internal operation $\$in$, similar to $\$mk$.

(object sub)

$$\frac{newloc^5(\sigma) =< \sigma', l_X, l_{mkX}, l_{inX}, l_{meth}, l_{dropX} >}{\begin{array}{l} C,\sigma, let\ X \leftrightarrow\ is\ Y\ and\ [\{A_i : T_i\}^{i\in I}; \{B_j = meth(\Sigma_j) : U_j\ is\ E_j\}^{j\in J}] \\ \rightarrow\ \ C^+ \equiv (C \oplus\ X \mapsto l_X \oplus\ mkX \mapsto l_{mkX} \oplus\ inX \mapsto l_{inX} \oplus\ dropX \mapsto l_{dropX}), \\ \quad \sigma' \oplus\ l_X \mapsto T{<}X, l_{Top}{>} \\ \quad \oplus\ l_{mkX} \mapsto F{<}C^+, \{\$x\}, InX(mkY(\$x), \$x){>} \\ \quad \oplus\ l_{inX} \mapsto F{<}\ (C^+ \oplus\ \$m \mapsto l_{meth}), \\ \qquad\qquad \{\$obj; \$x\}, \\ \qquad\qquad \$in(X, \$obj, [\{A_i := \$x.A_i\}^{i\in I}; \{B_j := \$m.B_j\}^{j\in J}]){>} \\ \quad \oplus\ l_{meth} \mapsto R(\{B_j \mapsto M{<}C, vars(\Sigma_j), E_j{>}\}^{j\in J}) \\ \quad \oplus\ l_{dropX} \mapsto F{<}\emptyset, \$x, \$drop(X, \$x){>} \end{array}}$$

The operation $\$in$ takes a type X, an object role O and a method suite E, acquires a new location l, checks that no X role is in the old O history, builds a new object role referenced by l, adds a pointer to l into the history of O, and returns l.

($\$in$)

$$\frac{\begin{array}{l} C,\sigma,X \;\rightarrow\; l_X,\sigma' \quad C,\sigma',O \;\rightarrow\; l_o,\sigma'' \quad \sigma''(l_O) = OR{<}l_Y, F, l_h, valid{>} \\ \sigma''(l_h) = H(OldH) \quad <l_X, \_> \notin OldH \\ C,\sigma'',E \;\rightarrow\; l_R,\sigma''' \quad \sigma'''(l_R) = R(F) \quad newloc^1(\sigma''') =< \sigma'''', l > \end{array}}{\begin{array}{l} C,\sigma,\$in(X,O,E) \\ \quad \rightarrow\; l,(\sigma'''' \oplus\; l \mapsto OR{<}l_X, F, l_h, valid{>} \oplus\; l_h \mapsto H(<l_X, l> \cdot OldH)) \end{array}}$$

Functions, variables, identifiers

If an identifier denotes a *derived* closure, the closure must be evaluated, in the original context C and in the current store σ. Otherwise, the value of the identifier is found in the context.

(Ide derived)

$$\frac{C(X) = l \quad \sigma(l) = D{<}C', E{>} \quad C',\sigma,E \;\rightarrow\; l',\sigma'}{C,\sigma,X \;\rightarrow\; l',\sigma'}$$

(Ide)

$$\frac{C(X) = l \quad \sigma(l) \neq D{<}\_,\_{>}}{C,\sigma,X \;\rightarrow\; l,\sigma}$$

Variable rules are simple. The update operation always returns the location l_{nil} which represent the only value of type *null*.

(var)

$$\frac{C,\sigma,E \mapsto l,\sigma' \quad newloc^1(\sigma') =< \sigma'', l' >}{C,\sigma,var\ E \;\rightarrow\; l',(\sigma'' \oplus\; l' \mapsto V(l))}$$

(at)

$$\frac{C,\sigma,E \mapsto l,\sigma' \quad \sigma'(l) = V(l')}{C,\sigma,at\ E \;\rightarrow\; l',\sigma'}$$

(update)

$$\frac{C,\sigma,L \mapsto l,\sigma' \quad C,\sigma',E \mapsto l',\sigma''}{C,\sigma,L \leftarrow E \;\rightarrow\; l_{nil},(\sigma'' \oplus\; l \mapsto V(l'))}$$

The definition of a function builds a closure which remembers the context as it was when the function has been defined. Function application uses that context, enriched with the values of the actual parameters, to evaluate the body of the function.

(function)

$$\frac{newloc^1(\sigma) =< \sigma', l >}{C,\sigma,fun(X_1{:}T_1,\ldots,X_n{:}T_n)\ is\ E \;\rightarrow\; l,(\sigma' \oplus\; l \mapsto F{<}C,\{X_i\}^{i\in 1..n},E{>})}$$

(application)

$$\frac{\begin{array}{l}C,\sigma,E \;\rightarrow\; l,\sigma_1 \quad \sigma_1(l) = F{<}C',\{X_i\}^{i\in 1..n},E'{>}\\ \forall i \in 1..n.\ C,\sigma_i,E_i \;\rightarrow\; l_i,\sigma_{i+1}\\ (C' \oplus\; \{X_i \mapsto l_i\}^{i\in 1..n}),\sigma_{n+1},E' \;\rightarrow\; l',\sigma'\end{array}}{C,\sigma,E(E_1,\ldots,E_n) \;\rightarrow\; l',\sigma'}$$

Before defining the semantics of object operations, we need an auxiliary function *hist* which, given an object, returns the concatenation of all of the histories of its base objects. The function is needed to define the semantics of the *As* and *isalso* operations. Notice how, in a *times* object, the history of the first component partially overrides the history of the second one.

$$\begin{array}{lcl} hist(OR{<}\_,\_,l_h,\_{>},\sigma) & = & \sigma(l_h)\\ hist(VE{<}\_,l{>},\sigma) & = & hist(l,\sigma)\\ hist(VR{<}\_,l{>},\sigma) & = & hist(l,\sigma)\\ hist(VT{<}\_,l_1,l_2{>},\sigma) & = & hist(l_2,\sigma) \oplus\; hist(l_1,\sigma)\end{array}$$

Record, object, and view operations:

(record)

$$\frac{\forall i \in 1..n.\ C,\sigma_i,E_i \;\rightarrow\; l_i,\sigma_{i+1} \quad newloc^1(\sigma_{n+1}) =< \sigma'_{n+1},l >}{C,\sigma_1,[\{A_i := E_i\}^{i\in 1..n}] \;\rightarrow\; l,(\sigma'_{n+1} \oplus\; l \mapsto R(\{A_i \mapsto l_i\}^{i\in 1..n}))}$$

(dot / !)

$$\frac{C,\sigma,E \;\rightarrow\; l,\sigma_1 \quad C,\sigma_i,E_i \;\rightarrow\; l_i,\sigma_{i+1} \quad C,\sigma_{n+1},l(l_1,\ldots,l_n) \stackrel{?A}{\rightarrow} l',\sigma'}{C,\sigma,E?A(E_1,\ldots,E_n) \;\rightarrow\; l',\sigma'}$$

(dot / !, no parameters)

$$\frac{C,\sigma,E \;\rightarrow\; l,\sigma_1 \quad C,\sigma_i,E_i \;\rightarrow\; l_i,\sigma_{i+1} \quad C,\sigma_{n+1},l() \stackrel{?A}{\rightarrow} l',\sigma'}{C,\sigma,E?A \;\rightarrow\; l',\sigma'}$$

(As)

$$\frac{C,\sigma,E \;\rightarrow\; l,\sigma' \quad C,\sigma',X \;\rightarrow\; l_X,\sigma' \quad <l_X,l'> \in hist(\sigma'(l),\sigma')}{C,\sigma,E\ As\ X \;\rightarrow\; l',\sigma'}$$

(isalso true)

$$\frac{C,\sigma,E \;\rightarrow\; l,\sigma' \quad C,\sigma',X \;\rightarrow\; l_X,\sigma' \quad <l_X,l'> \in hist(\sigma'(l),\sigma')}{C,\sigma,E\ isalso\ X \;\rightarrow\; l_{true},\sigma'}$$

(isalso false)

$$\frac{C,\sigma,E \;\rightarrow\; l,\sigma' \quad C,\sigma',X \;\rightarrow\; l_X,\sigma' \quad <l_X,l'> \notin hist(\sigma'(l),\sigma')}{C,\sigma,E\ isalso\ X \;\rightarrow\; l_{false},\sigma'}$$

The extend operation evaluates all the non-method fields, while the methods are stored as closures.

(extend)

$$\frac{C,\sigma,E \;\rightarrow\; l_b,\sigma_1 \quad C,\sigma_i,D_i \;\rightarrow\; l_i,\sigma_{i+1} \quad newloc(\sigma_{n+1}) =< \sigma'_{n+1},\{l_j\}^{j\in J} >}{\begin{array}{l} C,\sigma,E\ extend\ [\{A_i := D_i\}^{i\in 1..n};\{B_j = meth(\Sigma_j) : U_j\ \ is\ E_j\}^{j\in J}] \\ \quad \rightarrow\; l,(\ \sigma'_{n+1} \\ \qquad \oplus\ l_j \mapsto M{<}C, vars(\Sigma_j), E_j{>} \\ \qquad \oplus\ l \mapsto\ VE{<}\{A_i \mapsto l_i\}^{i\in 1..n} \oplus\ \{B_j \mapsto l_j\}^{j\in J}, l_b{>}) \end{array}}$$

(rename)

$$\frac{C,\sigma,E \;\rightarrow\; l_b,\sigma' \quad newloc^1(\sigma') =< \sigma'', l >}{C,\sigma,E\ rename\ (\{A_i \Rightarrow B_i\}^{i\in I}) \;\rightarrow\; l,(\sigma'' \oplus\ l \mapsto VR{<}\{B_i \mapsto A_i\}^{i\in I}, l_b{>})}$$

The semantics of the times operation depends on the static typing of the two components. All the messages which are statically detected in the type of E are redirected to the corresponding base object ($\{A_i \mapsto 0\}^{i\in I}$), and the same for E'. This is essential to ensure the type safety of the language, i.e. to avoid that one component may capture a message which is type-safe for the other component only.

(times)

$$\frac{C,\sigma,E \;\rightarrow\; l_b,\sigma' \quad C,\sigma',E' \;\rightarrow\; l'_b,\sigma'' \quad newloc^1(\sigma'') =< \sigma''', l >}{\begin{array}{l} C,\sigma,E\ times^{X_1..X_n,X'_1..X'_m}\ E' \\ \quad \rightarrow\; l,(\sigma''' \oplus\ l \mapsto VT{<}\{X_i \mapsto 0\}^{i\in 1..n} \oplus\ \{X'_i \mapsto 1\}^{i\in 1..m}, l_b, l'_b{>}) \end{array}}$$

Query operations:

To evaluate $select^{X_1..X_n}\ F\ from\ E$ we first evaluate E and obtain a sequence l^s. Then, for every element l^s_j of the sequence we evaluate F in a context where every label X_i of the sequence element type is associated with a derived expression which, whenever is used inside F, triggers the evaluation of $\$x.X_i$, where $\$x$ is an identifier bound to the location l^s_j. The semantics does not require the sequence elements to be accessed in any specific order; at each step j, F is evaluted w.r.t. the element $l^s_{b(j)}$. The condition $b : \{1..m\} \leftrightarrow \{1..m\}$ (i.e., b is a bijection over $\{1..m\}$) ensures that every sequence element is accessed exactly once. This makes the semantics non-deterministic, and gives the optimizer the possibility to exploit access structures which may be defined for the sequence.

(select from)

$$\frac{\begin{array}{l} C,\sigma,E \;\rightarrow\; l^s,\sigma' \quad \sigma'(l^s) = S(l^s_1 \cdots l^s_m) \quad newloc^{n+1}(\sigma') =< \sigma_1, l^r, l^d_1, \ldots, l^d_n > \\ b : \{1..m\} \leftrightarrow \{1..m\} \\ \forall j \in 1..m\ .((C \oplus\ \{X_i \mapsto l^d_i\}^{i\in 1..n}), \\ \qquad (\sigma_j \oplus\ \{l^d_i \mapsto D{<}(C \oplus\ \$x \mapsto l^s_{b(j)}), \$x.X_i{>}\}^{i\in 1..n}), F \;\rightarrow\; l^r_j, \sigma_{j+1}) \end{array}}{C,\sigma,select^{X_1..X_n}\ F\ from\ E \;\rightarrow\; l^r, (\sigma_{m+1} \oplus\ l^r \mapsto S(l^r_1 \cdots l^r_m)}$$

To evaluate $E\ where^{X_1..X_n}\ B$ we behave essentially as in $select^{X_1..X_n}\ B\ from\ E$ and we put in the result every l^s_j which makes B equal to l_{true}.

(where)

$$\frac{\begin{array}{l} C,\sigma,E \;\rightarrow\; l^s,\sigma' \quad \sigma'(l^s) = S(l^s_1 \cdots l^s_m) \quad newloc^{n+1}(\sigma') =< \sigma'', l^r, l^d_1, \ldots, l^d_n > \\ b : \{1..m\} \leftrightarrow \{1..m\} \\ \forall j \in 1..m\ .(\ (C \oplus\ \{X_i \mapsto l^d_i\}^{i\in 1..n}), \\ \qquad (\sigma_j \oplus\ \{l^d_i \mapsto D{<}(C \oplus\ \$x \mapsto l^s_{b(j)}), \$x.X_i{>}\}^{i\in 1..n}), B \;\rightarrow\; l^b_j, \sigma'_{j+1} \\ \qquad \text{where } \sigma_1 = \sigma'' \oplus\ l^r \mapsto S(()) \\ \qquad \text{and } \sigma_{j+1} = \sigma'_{j+1} \oplus\ l^r \mapsto S(l^s_{b(j)} \cdot Res) \\ \qquad\qquad\qquad\qquad \text{if } \sigma_{j+1}(l^r) = S(Res),\ l^b_j = l_{true} \\ \qquad\quad \sigma_{j+1} = \sigma'_{j+1} \qquad\quad \text{if } l^b_j \neq l_{true}\ ) \end{array}}{C,\sigma,E\ where^{X_1..X_n}\ B \;\rightarrow\; l^r, \sigma_{m+1}}$$

The $X\ In\ E$ operation creates one record with just one field X associated to l^s_j for every

element l^s_j of the value of E, and collects all these records in a sequence l^r.

(In)
$$\frac{C, \sigma, E \ \rightarrow \ l^s, \sigma' \quad \sigma'(l^s) = S(l^s_1 \cdots l^s_m) \quad newloc^{m+1}(\sigma') =< \sigma'', l^r, l^r_1, \ldots, l^r_m >}{C, \sigma, X\,In\ E \ \rightarrow \ l^r, (\sigma'' \oplus \ l^r \mapsto S(l^r_1 \cdots l^r_m) \oplus \ \{l^r_j \mapsto R(\{X \mapsto l^s_j\})\}^{j \in 1..m})}$$