A Guided Tour of Galileo 97

Antonio Albano°, Renzo Orsini•

° Dipartimento di Informatica - Università di Pisa
• Dipartimento di Informatica - Università Ca' Foscari di Venezia

January 15, 2008

Abstract

Galileo 97 is a statically and strongly typed, object-oriented database programming language incorporating new mechanisms to model databases in terms of objects with roles and classes and associations. This report presents a step-by-step tutorial showing how to create, populate and query a simple database. The purpose of this guided tour is to give a taste of the language without presenting an exhaustive description neither of the language nor of the programming environment functionalities.

Contents

1 Interacting with the System

The Galileo 97 language is implemented as an interactive interpreter. The user inputs a phrase in order to evaluate an expression or to establish a binding; then the system produces an answer.

The level of interaction at which a user enters a phrase and receives an answer is called the top-level of the system.

The following notation is used to illustrate the top-level interaction: user input is preceded by a "E:" prompt; system output is preceded by a ">" sign. The output portion is occasionally omitted when it is obvious. Once at the top-level, we can for example bind a value to an identifier, and then use that identifier in an expression. User input is always terminated by a semicolon.

```
E:  let a := 3;
>   a = 3 :int

E:  a + 4;
>   7 :int
```

The keyword **let** is used to bind identifiers to values.

Sequences of Galileo 97 top-level phrases can be stored in files and then executed by the interpreter with the `load "filename";` command.

2 Types

Language values have a type which is a predefined one (like **int** , **string** , **real**) or built by using type constructors. The following example illustrates the record type constructor.

```
let type Address := [ Street :string ; City :string ];
```

Address is a record type with components Street and City, whose values are of type string. Address is an example of "concrete" type, in the sense that it is not a new type, but it is simply an abbreviation of the type defined to the right of the ':=' symbol through the [. . .] record type constructor. Address is equal to every record type with the same components of the same types. A value of such a type can be constructed with the record constructor [. . .]:

```
let JohnAddress := [ Street := "Milano, 12"; City :="Torino" ];
```

and its components are selected with the dot operator:

```
JohnAddress.City;
    evaluates to
"Torino"
```

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. Several operators are available on sequences. Some of them are:

I **In** Q
TS **where** B
select E **from** TS
each TS **with** B
some TS **with** B

where:

- I is an identifier;
- Q is a sequence of value of type T_Q;
- B is a boolean expression;
- TS is a sequence of records, or objects, with type T_{TS};
- E is an expression with type T_E.

I In Q

returns a sequence of records with a unique field named I, associated with the value of the corresponding element in Q. The expression has type **seq** [I :T_Q].

TS where B

returns the sequence of the values of TS that satisfy the boolean expression B. The expression has type **seq** T_{TS}.

select E from TS

returns the sequence of the values of the expression E evaluated for each element in a sequence TS. The expression has type **seq** T_E.

each TS with B

tests whether every element in TS satisfies B.

some TS with B

tests whether at least one element in TS satisfies B.

Here are some simple examples of the use of sequences:

```
{2; 10; 9; 2; 5};

  {2; 10; 9; 2; 5} :seq int

x In {2; 10; 9; 2; 5} where x > 5;

  {[x := 10]; [x := 9]} :seq [ x :int ]

select   x + 1
from     x In {10; 9}
where    x > 5 ;

  {11; 10} :seq int

select   if x > 10 then x - 1 else x + 1
from     x In {8; 9; 10; 11; 12} ;

  {9; 10; 11; 10; 11} :seq int

first ({1; 2} append {3; 4});

  1 :int
```

Function are values which describes computations over parameters to produce a result. $\mathcal{T}_1 \rightarrow \mathcal{T}_2$ is the type of functions with a parameter of type $\mathcal{T}_1$ and result of type $\mathcal{T}_2$. Functions are defined as in the following example, where the parameter is an Address and the result produced is its Street component.

```
let streetOfAddress := fun(anAddress: Address): string is
                          anAddress.Street;
```

The only operation available on functions is the application, which is written with the usual mathematical notation:

```
streetOfAddres(JohnAddress);

  "Via Milano, 12" :string
```

3 Objects

Objects are denotable and expressible as “first-class” values of the language, i.e. they can be assigned as values of variables, and used as data structure components, as parameters or as results of functions. Since objects can be used as components of other objects, and can be shared, relationships among entities are modeled by relating the corresponding objects. Consequently, when objects are updated, their modification is reflected in all the other objects in which they appear as components.

An object is an instance of a type defined with a *generative type constuctor*: 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 Galileo 97 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 ]);
```

A method m of an object O is activated by sending a message to O with the notation O.m(...), if the method has parameters, otherwise with the notation O.m. The following example shows the use of the John method WhoAreYou:

```
John.WhoAreYou; returns "My name is John Smith"
```

4 Classes

The word “class” is usually used with the meaning of “type” in object-oriented languages, but in Galileo 97 it is used with a different meaning.

[1]Galileo 97 syntax is not very different from those of other object database languages, and the query language is very similar to the ODMG-OQL syntax.

A *class* is the mechanism used to represent databases by means of sets of modifiable interrelated objects: A class is the collection of all the constructed objects of an object type $\mathcal{T}$, called the *member type* of the class. A class is characterized by a *name* and the *type* of its elements. The name of a class denotes the sequence of the elements of the class currently present in the database (*class extension*), while the type gives the structure of the elements (*class intension*). The class extension varies in time: When an object with the type $\mathcal{T}$ of the elements of a class is constructed by mk$\mathcal{T}$, 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.

In the following example, we define the class Persons, whose members are of the object type Person. An object has the method WhoAreYou whose body is specified in the type definition:

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

The attribute Addresses is an example of reference whose value is a sequence. Values of this attribute are modifiable and contain sequences of values of type Address.

A class can contain a *key* definition: this is an integrity constraint which specifies that no elements in the class can exist with the same value for the attribute specified as key. In this case, no two persons can be created with the same TaxCode.

A class definition influences the behavior of the constructor operator for the class member type, by imposing that every object created of that type is automatically inserted in the class. For example:

```
let John := mkPerson ([ Name := "John Daniels";
                        Addresses := var {JohnAddress};
                        TaxCode := "jhndnls23h67";
                        BirthYear := 1967 ]);

let Jane := mkPerson ([ Name := "Jane Daniels";
                        Addresses := var {JohnAddress};
                        TaxCode := "jandnls20b63";
                        BirthYear := 1963 ]);
```

inserts John and Jane in the class Persons. For this reason, and since classes are sequences, the following expressions can be evaluated:

```
(get Persons where TaxCode = "jhndnls23h67").Name;
   evaluates to
"John Daniels",

select  p.WhoAreYou
from    p In Persons
where   p.BirthYear > 1963;
   evaluates to
{ "My name is John Daniels." } :seq string
```

Relationships among entities are modeled by relating the corresponding objects. To avoid the problem of "dangling references", an object should not be removed from a class as long as it participates in a relationship. For example, the relationship between cars and persons can be modeled as:

```
let
Cars class
   Car <-> [ Plate :string ; Owner :Person ];

let JohnCar := mkCar ([ Plate:="X345Y"; Owner:=John ]);
```

Multivalued relationships can be defined with attributes of sequence type:

```
let
Apartments class
   Apartment <-> [ Location: Address;
                   Tenant :var seq Person ];
```

5 Inheritance and Subtypes

Inheritance means any 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 if a type T is a subtype of the (super)type T', written also $T \leq T'$, then a value of T is also a value of T', but not vice versa because the subtype relation is a partial order. Consequently, a value of T can be used as argument for any function defined for values of T'.

The two notions are sometimes confused because, in object oriented languages, inheritance is usually 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.

The Galileo 97 type system supports *inclusion polymorphism*: i.e. a subtype relation is defined on types. Subtyping is defined for all types, but it only holds among types defined with the same type constructor. For the concrete types previously shown, the subtype relation is automatically inferred by the type-checker according to the following rules:

- Every type is included in itself.

- Every type is included in the type any.

- The type none is included in every type.

- If r and s are record types, then $r \leq s$ iff the set of identifiers of r *contains* the set of identifiers of s, and, if r' and s' are the types of a common identifier, then $r' \leq s'$.

  For example the type

  ```
  [ Name    :string ;
    Address :[ City :string ; Street :string ] ]
  ```

  includes

  ```
  [ Name    :string ;
    Address :[ City :string ; Street :string ; Country :string ];
    Age     :int ]
  ```

- If r and s are sequence types with elements of types r' and s', then $r \leq s$ iff $r' \leq s'$.

- If $var\, r$ and $var\, s$ are updatable reference types, then $var\, r \leq var\, s$ iff $r = s$ (*only trivial subtyping among updatable references*).

- If $(r \rightarrow s)$ and $(r' \rightarrow s')$ are function types, then $(r \rightarrow s) \leq (r' \rightarrow s')$ iff $r' \leq r$ and $s \leq s'$ (*domain contravariant function rule*).

  For example the type

  ```
  seq [ Name :string ; Age :int ] → [ Name :string ]
  ```

  includes

  ```
  seq [ Name :string ] → [ Name :string ; Count :int ]
  ```

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.

6 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 languages, and it is considered to be one of their defining properties.

7 Subclasses

In Galileo 97, each class can be either a *base class* or a *subclass*. A base class is defined independently of other classes, and is used to model a primitive collection of entities. A subclass is defined in terms of other classes, and is used to model alternative ways of looking at the same entities.

Let C2 be a subclass, with member type T2, defined in terms of C1, with member type T1. C1 can be either a base class or a subclass.

The following constraints hold:

1. *Structural constraint*: T2 is defined by inheritance from T1, consequently T2 $\leq$ T1.
2. *Extensional constraint*: If c is an element in C2 then c is also an element in C1, consequently C2 $\subseteq$ C1.
3. *Integrity constraint*: C2 inherits the key constraint in C1.

Here is an example of a subclass definition with the redefinition of a method:

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

Subclasses can be populated in two ways:

- by creating elements with the member type constructor, which will also appear as elements of its superclasses, because of the semantics of the subclass relation;
- by moving objects from the superclass into the subclass, through the operator inSubtype.

For example:

```
let Mary := mkStudent([ Name := "Mary Daniels";
                        Addresses := var {JohnAddress};
                        TaxCode := "mrydnls11c58";
                        BirthYear := 1958;
                        StudentNumber := 764;
                        Faculty := "Science";
                        YearOfAdmission := 1980 ]);

let JaneAsStudent := inStudent( Jane,
                                [ StudentNumber:= 765;
                                  Faculty := "Science";
                                  YearOfAdmission:= 1985 ]);
```

Both Jane and Mary are now students (and persons too):

```
select  s.StudentNumber
from    s In Students;
  evaluates to {765; 764}

select  p.Name
from    p In Persons;
  evaluates to
{" Jane Daniels"; "Mary Daniels"; "John Daniels" }
```

In the following example, we show the late binding property of type hierarchies:

```
select  p.WhoAreYou
from    p In Persons;
  evaluates to
My name is John Daniels
I was born in 1967

My name is Mary Daniels
I was born in 1958
I am student since 1980

My name is Jane Daniels
I was born in 1963
I am student since 1985
```

8 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.

In object oriented languages usually 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. 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, in Galileo 97 has been adopted a notion of object with *role* to support objects that can evolve over time by changing type and appearance. With this approach, 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. 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. 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. Let us discuss these issues in the context of Galileo 97.

In Galileo 97, besides the operator mk$\mathcal{T}$ to construct objects of type $\mathcal{T}$, the operator 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 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.

Method lookup generally gives different results, depending on the role used. For example, 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 Galileo 97 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.

9 Views for Objects and Classes

A *virtual* object can be defined from a real one by projecting, renaming or extending its components or methods, or by combining two or more objects (respectively with the operators **project**, **rename**, **extend**, **times**). For instance:

```
let ViewOfJohn := John extend [Age: int := 2008 - me.BirthYear]
                          project [Name; Age; Addresses];
 ViewOfJohn = - : < Person > view [ Name :string ; Addresses :var seq Address; Age :int ]

ViewOfJohn.Age;
  41 : int
John.Addresses <– {[Street := "Via Dolce Vita, 6"; City := "Roma"]};
select Street from at (ViewOfJohn.Addresses);
   "Via Dolce Vita, 6"  :seq string
```

The type of a virtual object (**view**) includes the type of the base object and a record type which describes its interface, that is its components and methods.

Virtual objects can be combined with the derived bindings to produce virtual classes. A derived binding binds an identifier, instead that with a value, with an expression which is evaluated each time the identifier is referred. For instance:

```
let PersonsWithCars := derived
        ( select  p extend [Cars: seq Car := Cars where Owner = p]
          from p in Persons)
        where count(Cars)>0;

select [Name := Name; NumOfCars := count(Cars)]
from PersonsWithCars;
    evaluates to
{ [ Name := "John Daniels"; NumOfCars := 1 ] }
```

In this example, PersonsWithCars is the sequence of all elements with at least a car of class Persons, seen with an additional component Cars. The derived binding makes the values of the sequence always updated, so that a query to PersonsWithCars is in effect a query to Cars.

10 The Query Language

A query is an expression which can be formed using the **select** clause, similar to the the standard SQL SELECT, universal quantification (**each**), existential quantification (**some**), membership testing (**In**), unary set operators (**min**, **max**, **count**, **sum**, **avg**), and the group-by operator (**group by**).

Examples will be given using the database in Figure 1. For simplicity, in the schema definition shown in Figure 2 the inverse of the relationships are not represented.

1. The following query retrieves the name of the persons who were born in 1980:

   ```
   select   p.Name
   from     p In Persons
   where    p.BirthYear = 1980;
   ```

2. Path expressions: The following query retrieves the street of the hauses in Pisa with a category ‘A3’:

   ```
   select   h.Address.Street
   from     h In Houses
   where    h.Address.City = "PI" And h.Category = "A3";
   ```

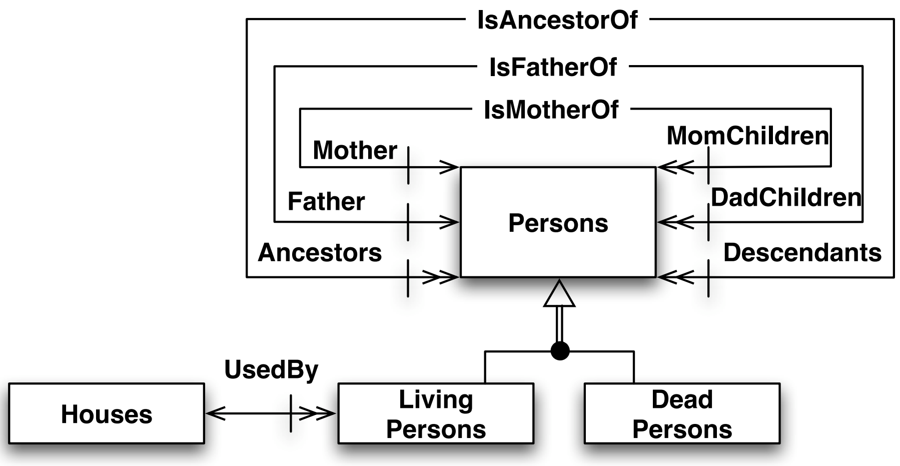

Figure 1: A database schema example

3. Path expressions: The following query retrieves the residence street of the living person with name John Simth:

```
select   p.Residence.Address.Street
from     p In LivingPersons
where    p.Name = "John Simth";
```

4. Join expressions: The following query retrieves the names of the living persons with the same residence:

```
select   [ first := p.Name; second := q.Name ]
from     p In LivingPersons times* q In LivingPersons
where    p <> q And p.Residence = q.Residence;
```

5. Join expressions: The following queries retrieves in two different ways the names of the living persons and the names of their ancestors:

```
select   [ personName := p.Name; ancestorName := q.Name]
from     p In LivingPersons times* q In p.Ancestors;
```

```
select   [ personName := p.Name;
           ancestorName := select q.Name from q In p.Ancestors ]
from     p In LivingPersons;
```

6. Join expressions: The following query retrieves the names of the living persons with the same mother that have the same residence:

```
let CurrentYear := 2002;
let type Address := [ Street :string ; City :string ];

let rec
Persons class
  Person <->
    [ Code          :string ;
      Name          :string ;
      BirthYear     :int ;
      Age           := meth () :int is CurrentYear - self.BirthYear;
      Father        :Person;
      Mother        :Person;
      Ancestors     := meth () :seq Person is
                         (if isunknown(self.Mother)
                           then {} :seq Person
                           else self.Mother :: self.Mother.Ancestors )
                         append
                         (if isunknown(self.Father)
                           then {} :seq Person
                           else self.Father :: self.Father.Ancestors )
    ]
and
LivingPersons subset of Persons class
LivingPerson<-> is Person and
    [ Residence     :House ]
and
DeadPersons subset of Persons class
DeadPerson <-> is Person and
    [ DeceaseYear   :int ]
and
Houses class
House <->
    [ Category      :string ;
      Address       :Address ];
```

Figure 2: A Galileo 97 Example

```
select  [ personName := p.Name; brotherOrSisterName := q.Name ]
from    p In LivingPersons times* q In LivingPersons
where   p <> q
        And Not isunknown(p.Mother) And Not isunknown(q.Mother)
        And p.Mother = q.Mother
        And p.Residence = q.Residence;
```

7. Subqueries: The following query retrieves the names of the living persons, and the number of their ancestors, which are father of more than 6 children:

```
select  [ personName := p.Name; numberOfAncestors := count(p.Ancestors) ]
from    p In LivingPersons
where   count( select  c
               from    c In LivingPersons
               where   Not isunknown(c.Father) And (c.Father = p) > 6;
```

11 Conclusions

We have presented the most important Galileo 97 features for modelling databases, but we have omitted other language features as well as several details on the concepts discussed. Further information may be found in the language reference manual.