Fibonacci: A Programming Language for Object Databases

Antonio Albano, Giorgio Ghelli, and Renzo Orsini

Received June, 1993; revised version received, February, 1994; accepted March, 1994.

Abstract. Fibonacci is an object-oriented database programming language characterized by static and strong typing, and by new mechanisms for modeling databases in terms of objects with roles, classes, and associations. A brief introduction to the language is provided to present those features, which are particularly suited to modeling complex databases. Examples of the use of Fibonacci are given with reference to the prototype implementation of the language.

Key Words. Database programming languages, objects with roles, data models.

1. Introduction

Fibonacci is an object-oriented database programming language characterized by static and strong typing and by new mechanisms for modeling databases in terms of objects with roles, classes, and associations.

Fibonacci was conceived to integrate features for modeling object-oriented database applications into a general purpose programming language that is statically and strongly typed. (Originally called Nuovo Galileo because it borrowed heavily from the Galileo language, which also was developed at the University of Pisa, the new language has been renamed in honor of the medieval mathematician Leonardo Pisano [Leonardo of Pisa], also known as Leonardo Fibonacci.) Since 1985, Galileo has been used extensively for teaching database programming using a strongly typed language. The language has proven to be useful for modeling complex databases in terms of objects grouped into classes that are organized in hierarchies, and it has shown the importance of static type checking in implementing database applications. This experience also highlighted that the techniques used by object-oriented languages for modeling objects and associations between sets

Antonio Albano, Dott. Ing., is Professor, Giorgio Ghelli, Ph.D., is Associate Professor, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, albano@di.unipi.it, ghelli@di.unipi.it. Renzo Orsini, Dott., is Associate Professor, Corso di Laurea in Scienze dell'Informazione, Università di Venezia, Via Torino 153, 30170, Mestre, Italy, orsini@moo.dsi.unive.it.

of objects are not satisfactory. Overcoming these limitations has been one of our major goals in the design of Fibonacci.

This article presents Fibonacci, and is organized as follows. Section 2 outlines our language design choices. Section 3 surveys the main features of Fibonacci. Section 4 presents the constructs of the language to define objects with roles. Section 5 presents the constructs to model sets of objects and associations between them. Section 6 describes the architecture of the current Fibonacci implementation. Section 7 compares the proposed solution with related works. In Section 8, we comment on our future plans.

2. Background

In this section we will discuss briefly the requirements on which the design of the language was based.

2.1 Why a Database Language?

Database languages integrate the abstraction mechanisms of a data model within a programming language. The result is an integrated programming language where data stored in the database are manipulated in the same way as any data structure of the language.

Database languages have been proposed to overcome the limitations of the traditional approach, where applications using the database are programmed by languages which host the data model operators, using ad hoc mechanisms to exchange information between a program and the DBMS. This traditional programming environment is not suitable for the growing complexity of applications, due to the limited integration of the data model into the programming language.

From the latter half of the 1970s, attention was focused on the integration of the relational data model abstraction mechanisms in the type system of a programming language, to construct an integrated programming environment to develop database applications. The term "database programming language" was used to refer to this approach, examples of which include languages such as Pascal/R (Schmidt, 1977), RIGEL (Rowe and Shoens, 1979), PLAIN (Wasserman, 1979), and DBPL (Schmidt and Matthes, 1990). A different approach to integrate the relational data model in a complete programming language is exemplified by the Datalog language, which allows relational database applications to be written in a Prolog-like logic programming language. At the end of the 1970s, new database programming languages were proposed to support a more expressive data model, the "semantic data model," to overcome the modeling limitations of traditional data models. This resulted in the development of so-called conceptual languages, such as ADAPLEX (Smith et al., 1981), Taxis (Mylopoulos et al., 1980), and Galileo (Albano et al., 1985). More recently, new database programming languages have been proposed, which are based

on the object-oriented programming paradigm (Albano and Ghelli, 1991; Zdonik and Maier, 1990; Bancilhon et al., 1992), which seems the best suited to satisfy the needs of complex database applications. The same phenomenon is taking place in the world of commercial database systems, where most of the new systems support some features of the object-oriented data model, and offer a language that is not just a DDL-DML, but is a complete language supporting application development. This trend suggests that database programming languages may become one of the basic tools to write database applications.

2.2 Why a Persistent Language?

Persistent programming languages are characterized by the following properties:

- any value, irrespective of its type, has the same rights to persistence (orthogonal persistence);
- programs are written in a way that does not depend on the persistence of data on which they operate (persistence independence).

As a consequence, the programmer uses the same operators on data, whatever their lifetime, and does not need read and write operators to control data movement. Examples of such languages are PS-Algol (Atkinson et al., 1981), Napier88 (Morrison et al., 1994), Galileo (Albano et al., 1985), and Tycoon (Matthes and Schmidt, 1994).

Orthogonal persistence is especially important in those applications where complex data types are manipulated or where the number of different types is similar to the number of different values manipulated. In these applications, orthogonal persistence allows the types of persistent data to be defined by using any type constructor of the language, and the programmer is freed from the task of finding a way to represent any persistent data structure by using only the abstraction mechanism of a specific data model (Atkinson et al., 1983; Atkinson and Buneman, 1987; Atkinson, 1991).

It is a matter of debate whether database application development would be better supported by a persistent programming language that offers a set of built-in data definition and manipulation operations, and direct support for a fixed data model (built-in approach), or by a persistent programming language that gives programmers the possibility of defining their own data model (add-on approach; Matthes and Schmidt, 1991). Fibonacci follows the built-in approach, while Napier88 and Tycoon are two examples of the add-on approach.

2.3 Why a Statically and Strongly Typed Language?

A language is strongly typed if all computations are checked for type errors. A language is statically checked if all type errors are discovered through textual analysis. Static typing is essential for languages used in the final encoding of applications, in particular for complex applications, which evolve in time, as happens with database

applications. Software systems are always evolving due to the constant evolution of requirements and to the continuous discovery of bugs, and a statically typed language provides one way to control change and, thus, helps to produce and maintain reliable software systems, since it ensures that no evolutionary step introduces run-time type errors.

Static and strong typing, however, can also be a hindrance to software evolution when it is unduly restrictive. For example, a function gimmeJohns, which takes a set of persons and returns only those whose name is John, need not be modified in a dynamically typed system when the definition of type Person is modified if the name field is not affected. However, in most typed languages (e.g., in Pascal), gimmeJohns should at least be re-checked, if not modified, whenever type Person is modified. This problem essentially disappears in polymorphic languages, such as Fibonacci, where it is possible to describe that gimmeJohns just needs a name field and returns values of the same type as its input type. In this situation, the function must be modified and re-checked only when the name field of persons would be affected, as would happen in an untyped language.

Other benefits achieved by a static and strong typing discipline are: (1) programs and data definitions are more readable; hence, they can be better maintained, (2) debugging is greatly eased by the controls made by the type checker: it is typical of database applications that once the program is well-typed very few errors (or none at all) remain in the code, (3) programs are more efficient since they do not perform run-time type checking (Cardelli and Wegner, 1985).

2.4 Why an Object-Oriented Language?

The object-oriented data model is the best suited to represent complex data types, especially when the database is a graph of interconnected entities with a structure that is not as homogeneous and regular as in traditional business applications. Other important features of this data model are the possibility of describing procedural information (methods) within the schema, and the encapsulation of object state. This allows a stricter maintenance of some classes of constraints. These features make object-oriented database languages well suited to applications that use complex data of different types, such as those in office information systems, computer-aided design and manufacturing systems, computer-aided software engineering, multimedia and hypermedia information systems, and knowledge-based systems. In the field of programming languages, the object-oriented paradigm is characterized by the fact that the combination of encapsulation, inheritance, and late binding allows the functionalities of data types to be extended without knowing their implementation. This feature of object-oriented languages explains the high level of modifiability and reusability, which characterizes object-oriented software (Atkinson et al., 1989; Dittrich, 1990; Kim, 1990; Zdonik and Maier, 1990).

2.5 Why Objects and Roles?

The standard object-oriented data model is not satisfactory when entities must be modeled, which change the class they belong to and their behavior during their life. When an object is allowed to acquire new object types, the problem arises that different and inconsistent behaviors may have been defined for these different object types. This problem can be solved by preventing these inconsistent behaviors from "mixing up." In this approach, an object has a context dependent behavior and, in each context, it exploits only a consistent subset of its possible behaviors, and this subset depends on the "role" that the object is playing in that context. A role mechanism faces this problem and, more generally, allows one to model entities that can play several roles and behave according to the role being played (Albano et al., 1993; Richardson and Schwartz, 1991).

2.6 Why an Object-Association Data Model?

The object-association data model extends the object-oriented data model with a specific construct to model associations. Usually, in object-oriented data models, objects model entities of the domain of discourse, classes¹ model sets of homogeneous entities, and an association between two objects is represented by having a method in each of them that returns the other one; one-to-many associations are modeled by methods that return collections of objects. The advantage of this traditional approach is that cardinality constraints on the associations can be enforced using types, and associations are dealt with exactly like object attributes. Nevertheless, this approach has a number of drawbacks, namely:

- associations are conceptually a higher level abstract notion whose implementation should be decided by the DBMS, while attributes force the programmer to choose a specific implementation for them;
- an association is a single conceptual entity whose semantics, in the object model, is split among several objects;
- the enforcement of the inverse relation constraint (i.e., the fact that the two methods modeling the associations are the inverse of each other) is left to the programmer;
- associations relate objects that exist independently, and it should be possible
 to define them incrementally without redefining the structure of existing
 objects;

^{1.} The term class is used with different meanings in the literature. In object-oriented languages, a class is like a type in conventional programming languages, and it is used to define the structure and behavior of a set of possible objects. Often, in object-oriented database languages, a class also is used to refer to the set of all existing instances of a class. In this article, a class is used to mean a collection of homogeneous values of any type.

- associations are not necessarily binary, and can have their own attributes;
 these aspects can be modeled only indirectly by means of attributes;
- operations on relationships as a whole are not possible in a straightforward way.

To overcome these problems, an extension of the object-oriented data model with an association construct was proposed (Albano et al., 1991). The association is a first class type constructor. An association is a modifiable set of tuples that is used to represent the existence of associations between classes of objects, in a way that resembles how n-m associations are represented in relational schemas: to represent the fact that two objects are associated, a tuple containing the two objects is inserted into the association. When an association is defined, some constraints can be defined on its extensions, such as referential and cardinality constraints (Section 5).

This approach solves the problems listed above. Specifically, in the object-association data model, an association is defined in a purely declarative way, its description doesn't depend on the description of the types of the associated objects, and the system is free to represent, internally, the association in the most convenient way.

3. Overview of Fibonacci

Here we present an overview of the main features of Fibonacci.

Fibonacci is an expression-based language. Each construct is applied to values and returns a value.

Fibonacci is a persistent language. All data transitively accessible from the global environment (top-level) survive automatically between different work sessions, irrespective of their type. Data are removed by a garbage collector when they are no longer reachable from any identifier in the global environment.

Fibonacci is an interactive language. The system repeatedly prompts for input and reports the result of the computation; this interaction is said to happen at the top level of evaluation. At the top level, one can evaluate expressions or perform declarations. Stand-alone applications interacting with the top-level environment can also be written.

Fibonacci is a higher order language. Functions are denotable and expressible values of the language. Therefore, a function can be embedded in data structures, passed as a parameter, and returned as a value.

Fibonacci is a safe language. Each legal expression has at least one type, which is statically checked. Each type is related to a set of operators that can be applied to values of this type (e.g., the field selectors of a tuple type). The basic types are None, Null, Bool, String, Int, Real, Any. The instances of basic types

are all disjoint, with one notable exception: the value unknown (of type None), which belongs to any type whatsoever. Besides these types, type constructors exist to define new types, from basic or previously defined types. They are divided into two categories, concrete type constructors (tuple, sequence, discriminated union, function, class, and association types) and generative type constructors (object and role types). Type constructors take types as parameters, and produce other types. Type equality is structural with concrete types (i.e., two types are equal if they are built with the same constructor applied to types recursively equal), while it is by name (more precisely by definition time) with generative types.²

Fibonacci types are first class. Fibonacci values have an equal possibility to be stored in persistent or temporary data structures, and to be parameters or results of functions independently of their type. Any type operator can be applied to any other type.

Fibonacci assignment has sharing semantics. Values of any type always are used directly, and not by copying them, when they are passed as parameters to functions, bound to identifiers in declarations, and used in constructing complex values.

Fibonacci has subtyping. A subtype relation is defined on concrete types. This relation allows the so-called *inclusion polymorphism* to be exploited: if T1 is a *subtype* of T2 (also, T2 is a *supertype* of T1), then a value of T1 is also a value of T2, consequently, it can be used in every context where a value of T2 is expected. A subtype relation with this property also exists among role types, which must be explicitly declared when a new role type is defined.³

Fibonacci supports inheritance. Inheritance is the ability to define a new object type and a new object type implementation by extending a previous type or implementation definition. In Fibonacci, as in most object-oriented languages, inheritance can be used to define object subtypes and their implementations, starting from the corresponding definitions for object supertypes.

Fibonacci has objects with roles. Objects are entities with an immutable identity and a mutable state. The state is encapsulated and can only be queried and updated by sending messages to the object. An object is internally organized as an acyclic graph of roles. A role is also an entry to access the object it belongs to: an object can only be accessed through one of its roles, and its behavior depends on this role. An object could be considered as simply being a container for its roles, since messages are handled and cooperatively answered by these roles. Fibonacci distinguishes

^{2.} While in some object-oriented languages the term "value" is used to refer to values of concrete types only, in our terminology it refers to values of every type, including objects.

^{3.} Subtyping between types with modifiable fields creates some well-known problems (Albano, 1983; Albano et al., 1985; Connor et al., 1991), which are solved in Fibonacci by having a specific type constructor Var for modifiable data, such that no subtyping relation exists between different Var types.

between object and role operations. Object operations allow one to test for the existence of certain roles in the object and to gain a reference to one of these roles, and to extend objects by giving them new roles (for example, adding the Employee role to a Person). The basic role operation is message passing. The Fibonacci role-based approach is a conservative extension of the classical object-based approach: if the special object operations are not used, and objects are never extended, then message passing behaves as in traditional object-oriented languages, and Fibonacci role types can be used as if they were traditional object types. Finally, in Fibonacci the implementation of a role type is defined separately from the type itself, and the same type can have several implementations.

Fibonacci has polymorphic functions. A function is polymorphic when its type does not prescribe a single input type, but describes the set of all acceptable input types and how the output type depends on the current input type. Fibonacci supports both inclusion polymorphism (subtyping), and parametric polymorphism, since functions can be defined with types as parameters, and these types can then be used to give the types of the other parameters and of the result.

Fibonacci has an exception-trap mechanism. Exceptions can be raised and selectively trapped, and exception handlers can be specified.

Fibonacci has a nested transaction mechanism. When a transaction fails, all its side-effects, both on persistent and transient values, are undone; transactions may be nested.

4. Objects and Roles

4.1 Overview

An object is the computer representation of certain facts about a real-world entity. An object is a software entity that has an internal state equipped with a set of 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. The state of an object can be accessed and modified only through operations associated with that object. Each object can send messages to itself (self-reference). Message interpretation (i.e., the choice of the method used to reply to a message), always depends on the object that receives that message. This means that different objects may use different methods to reply to the same message. Each object is distinct from all other objects and has an identity that persists over time, independently of changes to its state. All these features of object oriented languages are retained in Fibonacci, and are extended as follows.

Suppose that a *Person-Student* hierarchy has been defined in any object-oriented language. When a *Student "john"* receives a message, it may answer either as a *Person*, if the corresponding method has been inherited (e.g., *Age*) or as a *Student*,

if the method has been added or redefined (e.g., StudentNumber) or, finally, it may ask itself "how would I answer if I were just a Person?" before composing its answer as a Student (the super mechanism). In Fibonacci terminology, we describe this situation by saying that john contains both a Person and a Student role. From the outer world, every message is addressed to its Student role but, internally, the two roles cooperate to build up the answer, in two different ways:

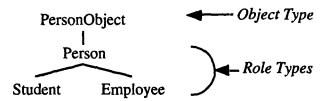
- by inheritance: when the *Student* role does not have a method, it looks for a method in its super-role *Person* (method lookup);
- by message passing (the *super* mechanism): a method of the *Student* role can send a message to the *Person* role.

The essential assumption underlying the classical model is that every object has one most specialized role, the role that receives any message sent to the object and starts the method lookup mechanism. Fibonacci extends the classical model in two directions. First of all, due to object extendibility, a single object may have different most specialized roles. For example, if Employee is a second subtype of Person, by extending john with the Employee role, we obtain an object with two unrelated most specialized roles, Student and Employee. Second, messages are not sent to objects, but they always are directed to a specific role of the object. The operations that build and extend an object, in fact, return a reference to a specific role of the object; the most specific role in the first case, and the new role in the second case. Hence, in Fibonacci, we never manipulate objects directly, but we always manipulate roles of objects; and a role is cast implicitly to its object only when an object operation, such as equality or extension, is applied. When we want to send a message to john, we must either send it to the value, say johnAsStudent, returned when john as been created as a student; or to the value, say johnAsEmployee, returned by the object extension operator. In the first case, johnAsStudent will answer as a Student (i.e., it will look for the method starting from its Student role and then will try to inherit it from its Person role). In the second case, method lookup will start from the Employee role, and then will go up to its only ancestor Person.

The role mechanism was introduced in Fibonacci to preserve cousin independence. We say that two object types are cousin types when no one type is a subtype of the other one, but they are subtypes of a common ancestor. Cousin independence means that, when programmers are defining an object type (a role type in Fibonacci), they only have to know the definition of its supertypes, and don't have to be aware of the existence of other cousin types. This principle implies that no relationship can be assumed to exist between two methods defined for two cousin types for the same message. Hence, any method lookup mechanism must ensure that no message addressed to an object of type T is answered by a method which has been designed for a cousin type S. Cousin independence holds for the standard object-oriented languages, because an object is created with a type and cannot acquire new types.

Now, suppose that two different subtypes *Employee* and *Student* are defined for type *Person*, and that in both types a *Code* field is defined, with a different

Figure 1. Object-role type hierarchy



meaning and even a different type (e.g., String and Int, respectively). If Employee and Student have no common subtype, then this situation must be legitimate, due to the cousin independence principle. This situation is not a problem in traditional object-oriented languages, where no object can be an Employee and a Student at the same time. In Fibonacci, however, such an object may exist, and this object must answer a Code request in a way that is related to the "role" it is playing. This was the main reason behind the design of Fibonacci role mechanism.

This mechanism, whose primary purpose is to deal with object extendibility, also enables Fibonacci to model those real world situations where a single entity can play several roles and behaves differently according to the role it is playing. In traditional object-oriented languages, where every object always is accessed through its most specialized role, it is not easy to model such situations.

Note that a Fibonacci object which is built directly in its most specialized role, and which never acquires new roles, behaves exactly like an object in other object-oriented languages. Hence Fibonacci programmers can ignore the object-roles mechanism with all its subtleties until they actually need it.

To emphasize the distinction between object and role operations, Fibonacci distinguishes between *object types* and *role types*. A typical hierarchy of role and object types is depicted in Figure 1.

A role type is characterized by the messages it accepts; hence, it is very similar to an object type in a traditional object-oriented language. An object is simply a set of roles; hence, an object type specifies the set of roles that *may* be part of an object of that type: referring to the figure, an object of type *PersonObject* may contain a *Person*, a *Student*, and an *Employee* role. This set of role types is open-ended, since new sub-role types can be added to any object type at any moment, and is called the *role type family* of the object type. Since the object type is a supertype of its role type family, object operations, such as equality and tests for the existence of a specific role, can be applied to values that denote a role.

4.2 Objects and Roles: Type Operators

NewObject is the constructor for a new object type, which is needed to begin the construction of a role type family. Since objects cannot be manipulated independently of their roles, this operator introduces a new type without giving any information

Figure 2. Person role type definition

about the messages managed by its values: this information is specified in the definition of its role type family. For example, a definition of a new object type *PersonObject* is:

Let PersonObject = NewObject;

A role type is defined with the constructor **IsA** ... **With** ... **End** as a subtype of an object type or as a subtype of other role types. In both cases, a role type always belongs to the role family of one object type, called its *object supertype*. A value belonging to a role type is called a *role value* (or simply a *role*). By subtyping, a role value also belongs to the object supertype of its role type, so that both object operations (equality, role casting, role inspection, extension) and role operations (message passing) can be applied to a role value. The messages that can be sent to a role are those specified by its role type, either by listing them in the role type definition or by inheriting them from the role supertypes.

A role type is defined by a set of *properties* (i.e., by giving the signature of the methods of that role). **NewObject** and **IsA** are *generative* type constructors (i.e., each object type and role type definition produces a new type), different from any other type previously defined. Figure 2 shows the definition of the role type *Person*.

A role type family can be extended dynamically by defining a new role type T as a *subtype* of others, called its *supertypes*. The subtype *inherits* all the properties of its supertypes, unless they are explicitly redefined in the subtype (*overriding*). In the case of multiple inheritance, if a property is present in more than one supertype, and there is not an explicit redefinition in the subtype, then the property of the *last specified supertype* is inherited (but its type must specialize all the types of all the

Figure 3. Role subtype definitions

Let Student = IsA Person With
Faculty: String;
StudentNumber: Int;
Introduce: String;
End;
Let Employee = IsA Person With
Department: String;
EmployeeNumber: Int;
Introduce: String;
End;

other inherited versions; see below). Multiple inheritance of a property is allowed only if that property has been defined in a common ancestor. This rule is enforced to prevent two methods with the same name but different meanings from being accidentally merged in a subtype. Thus, when two methods have the same name and the same meaning, this rule forces the programmer to define a common supertype where such a method is introduced. Figure 3 shows the definition of the subtypes Student and Employee of the type Person.

In a subtype definition S, for any property p of S (inherited, redefined, or added), if p also is defined in the supertype T, then the following conditions hold:

- the signature of p in T is a subsignature of the one in S (contravariance);⁴
- the output type of p in S is subtype of the one in T (covariance).

4.3 Object Construction

A role type T defines the interface of the objects with such a type, but doesn't give any information about their internal structure. An object of type T is created with the construct role T < implementation > end, where the implementation specifies the private state of the object and the body for all the methods specified in the interface; to be more precise, the role operator builds an object with one role, and returns a reference to that role of the object. Figure 4 shows an expression that creates a new object and returns a value (john) of type Person, which denotes both the new object and its only role.

^{4.} A signature is a list of zero or more Identifier: Type pairs separated by semicolons. Thus, we say that S1 is a *subsignature* of S2 if S1 extends S2 with new pairs or redefines (in the same order) the S2 pairs with more specialized types.

Figure 4. Example of object construction

```
let john = role Person
   private
        let address = var ("Darwin road, 123 - London"):<sup>5</sup>
   methods
        Name = "John Daniels";
        BirthYear = 1967;
        Age = currentYear() - me.BirthYear;
        Address = at (address);
        modAddress (newAddress: String) =
                      if stringLength(newAddress) <= 0</pre>
                      then failwith "incorrect address"
                      else address := newAddress
        Introduce =
                      "My name is " & me.Name &
                      "and I was born in " & intToString(me.BirthYear);
   end;
```

The **private** section of the **role** construct specifies a private environment that can be accessed only by the role methods. The **method** section specifies the methods of the role which, in a database example, can be as simple as a method that just returns a constant value, thus making an object very similar to a record. Once an object is created, its methods can be selected with the dot notation (e.g., john.Address). A method call causes the evaluation of an expression in the private environment with possible side-effects; this is the only way to ask an object to modify its internal state.

To create many instances of a role type, with the same internal structure and the same methods, a constructor can be defined, which is a function that returns objects of a certain type. An example of a constructor for objects of type Person is shown in Figure 5.

The expression fun (<arguments>): <type> is <exp> defines a function with type Fun (<arguments>): <type> and body <exp>. 6 When the function is applied,

^{5.} In Fibonacci, a modifiable cell is introduced with an expression var expr, and its value is read with the expression at expr.

^{6.} A function definition has a different syntax from that of a method. This is to reflect the fact that there are differences between functions and methods. A function is a first-class value, and so can be passed as a parameter or returned as a value by a function. A method is not a value by itself, but is a part of a role definition, and it can be evaluated only by sending a message to the object to which it belongs.

Figure 5. Example of constructor for objects

```
let createPerson = fun(name, address: String; birthyear: Int): Person is
   role Person
   private
      assert stringLength(address) <= 0 elsefail "incorrect address";</pre>
      let address = var (address);
   methods
      Name = name:
      BirthYear = birthyear;
      Age = currentYear() - me.BirthYear;
      Address = at (address);
      modAddress (newAddress: String) =
             if stringLength(newAddress) <= 0</pre>
             then failwith "incorrect address"
             else address := newAddress
      Introduce = "My name is " & me.Name &
                    "and I was born in " & intToString(me.BirthYear);
   end;
```

a new instance of *Person* is created. While the private data are generally different for each instance, method bodies are shared by all the instances.

In the body of the Introduce method, the special identifier **me** denotes the constructed object. The *static type* of **me** is always the type of the expression role where **me** is written (in this example Person), but when the method is executed by a role that inherits it, then **me** denotes the role that inherited the method (working as *self* in other object-oriented languages). **me** can be used only in the method bodies. intToString is a predefined function to convert an integer into a string. The infix operator & is the concatenation operator on strings.

By separating object implementation from object types, Fibonacci allows objects with different implementations but with the same type to coexist safely. This is especially useful in evolving environments, where the best implementation of an object type may change over time, and it is useful also in accommodating objects with a different nature but a unique interface.

4.4 Dynamic Object Extension and Inheritance

To model the possibility of real world entities to acquire new roles, Fibonacci provides an extension operator, which allows an object to be extended with new subroles. A new role provides the object with a new piece of private environment and a new

Figure 6. Example of object extension

set of methods, which override previously defined ones (more details will be given in Section 4.5). Figure 6 shows how the **ext-to-private-methods-end** operator can be used to extend john from Person to Student.

The object acquires the new role without changing its identity. After the extension in Figure 6, john and johnAsStudent are two different roles of the same object; the test john = johnAsStudent returns true since equality is an object operation. The semantics of the expression (me as Person)!Introduce used in the definition of the method Introduce will be explained in Section 4.5. Informally, it executes the method Introduce defined in Person.

Figure 7 shows how the ext operator can be used to define a constructor for students by inheritance (i.e., by extending the person constructor createPerson). Note, however, that in Fibonacci, a constructor for a role type with a super-role can be defined either by inheritance, using the ext operator, or directly, using the role operator.

Figure 8 shows how the ext operator can be used to define an extension constructor (i.e., a function to transform a Person into an Employee).

4.5 Method Lookup

In traditional object-oriented languages, every message is directed only to the most specialized role, which either has a method for the message, or looks for a method in its ancestors. The same happens in Fibonacci when a message is sent to one of the most specialized roles of an object. However, when a message is sent to a role that has several descendants, two different extensions of the standard mechanism are possible:

• upward lookup: the method is looked up first in the receiving role and then in its ancestor roles;

Figure 7. Defining a constructor by inheritance

• double lookup: the method is first looked for in all the descendants of the receiving role, then in the receiving role and, finally, in its ancestor roles.

Upward lookup emphasizes the difference of behavior between different roles, while the double lookup results in a more uniform behavior. Both rules, however, ensure that if T and S are two cousin roles, then no message sent to S will be answered by a method in T. Since both rules can be useful in some situations, Fibonacci gives both possibilities. Upward lookup is achieved by object!message, and double lookup by object.message; upward and double lookup coincide, and also coincide with the traditional lookup technique, when a message is sent to a most specialized role of an object.

The following example illustrates the mechanism. Recall that the object *John* has two roles, johnAsStudent and john, of type Student and Person. Upward and double lookup coincide for johnAsStudent, which has no descendant, while they may behave differently for john:

```
johnAsStudent.Introduce → "My name is John Daniels and I was born in 1967. I am a Science student"
johnAsStudent!Introduce → "My name is John Daniels ...
Science student"
john.Introduce → "My name is John Daniels ...
Science student"
john!Introduce → "My name is John Daniels and
I was born in 1967"
```

Once john is extended to an employee, its double lookup behavior changes, since the last acquired subrole is the first to be looked for:

Figure 8. Example of extension constructor

```
let to Employee = fun (aPerson: Person; dept: String) : Employee is
   ext aPerson to Employee
  private
        let employeeNumber = newEmployeeNumber();
  methods
       Department = dept;
        EmployeeNumber = employeeNumber;
        Introduce = (me as Person)!Introduce & ". I am an employee";
   end;
toEmployee(john; "Quality Management");
john.Introduce \rightarrow "My name is John Daniels and I was born
                       in 1967. I am an employee"
However, neither the upward lookup behavior of john nor the behavior of john as
a Student are affected:
                        \rightarrow "My name is John Daniels and I was born
john!Introduce
                                 in 1967"
johnAsStudent.Introduce → "My name is John Daniels ...
                                 Science student"
johnAsStudent!Introduce → "My name is John Daniels ...
                                 Science student"
```

Both upward and double lookup are two forms of late binding (or dynamic binding, or dynamic lookup). In object-oriented terminology, late binding of methods to messages means that the method executed to answer a message does not depend on the static type of the receiver (i.e., on its compile-time type), but on its run-time type or, in languages where different implementations are allowed for the same type, on its run-time value. For example, late binding means that if a Student is bound to a variable of type Person, or is passed to a function expecting a Person parameter, it still behaves like a Student:

```
let aPerson: Person = johnAsStudent; aPerson!Introduce \rightarrow "My name is John Daniels ... Science student"
```

4.6 Object Comparison, Role Inspection, Role Casting

Since an object in Fibonacci is a modifiable collection of roles, the language provides the following operators on objects:

• the equality operator (=) to test whether two objects are the same, independently of the role used to access them; for example

```
johnAsStudent = john; (* returns true *)
```

• the infix predicate isAlso to test whether an object has a certain role; for example:

```
john isAlso Employee; (* after extension, returns true *)
```

• the infix operator as to coerce an object to one of its possible roles (role casting). The operator fails if the object does not have the specified role:

```
let johnAsEmployee = john as Employee;
```

• the expressions x as/isAlso T are well typed if T and the type of x belong to the same role type family.

The combination of casting with strict lookup (e.g., (X as T) ! P) allows the simulation of the traditional *send-to-super* mechanism of object-oriented languages (Figure 6). The same combination also allows simulating static binding, as shown below, where another Person behaves like a Person:

```
let anotherPerson: Person = johnAsStudent as Person;
anotherPerson!Introduce → "My name is John Daniels and I was born
in 1967."
```

Finally, the **isExactly** operator is available on role values, to test their run-time role type. For example:

```
johnAsEmployee isExactly Employee; (* returns true *)
john isExactly Employee; (* returns false *)
aPerson isExactly Student; (* returns true *)
anotherPerson isExactly Student; (* returns false *)
```

5. Bulk Types: Class, Association, Sequence Types

Bulk types describe collections of values with common properties. Fibonacci supports three kinds of bulk types: class, association, and sequence type.

Classes are modifiable ordered sets of homogeneous values, used to model sets of entities in the domain of discourse. Associations are modifiable ordered sets of tuples, used to model associations between entities. Sequences are constant collections of homogeneous values of any type.

Fibonacci query algebra is defined on sequences; since a class or an association type is a subtype of a sequence type, Fibonacci's algebraic operators can be applied in the same way to the three kinds of data structures. On the other hand, operators

to insert and remove data, and to declare integrity constraints, are available only on the updatable bulk types: classes and associations.

Classes, associations, and sequences are first class types of the language; hence, it is possible to apply these type constructors to any other type of the language, at any nesting depth.

5.1 Class and Association Types

Class ElemType is a type of homogeneous ordered sets of elements of type ElemType. Classes differ from sequences because they can be updated (while sequences are constant), no repeated element is allowed in a class, and it is possible to define constraints on classes, such as inclusion or mutual disjointness, as described below.

Assoc TupleType is a type of homogeneous sets of tuples of type TupleType. Tuples are ordered associations of values with identifiers; the notation [Ide₁: Type₁; ...; Ide_n: Type_n] denotes a tuple type, while the notation [let Ide₁ = v_1 ; ...; let Ide_n = v_n] denotes a tuple value. Associations behave like classes of tuples: they can be updated, no repeated tuple is allowed, and it is possible to define constraints on them, mainly to connect association fields with classes.

5.1.1 Operations. A new empty class and a new empty association are created by the emptyClass of and emptyAssoc of operations, as in the example below, where Student and Class are previously defined types:

```
let students = emptyClass of Student end;
let classes = emptyClass of Class end;
let enrolled = emptyAssoc of [student: Student; class: Class] end;
```

Elements and tuples are inserted and removed from classes and associations using the **insert** and **remove** operators:

```
insert Expr into Expr
remove Identifier from Expr where BoolExpr
```

In the **insert** operations, the type of the value or tuple inserted must be a subtype of the element type of the class or association. **remove** removes all values, or tuples, that satisfy BoolExpr from the specified class or association, but the removed elements are not deleted until an access path exists for them. When a value is inserted into a class or into an association, a check is made to establish whether the value is already there. In this case, the insertion is a no-operation. Removal and insertion are executed atomically: if the operation cannot be completed (typically due to some constraint violation) every side effect is undone.

Besides these operators, all the operations on sequences defined below can be applied to classes and associations since, as already discussed, the following type inclusions hold, where $\{T\}$ is the type of sequences of elements of type T:

```
Class ElType \leq { ElType }
Assoc TupleType \leq { TupleType }
```

5.1.2 Integrity Constraints on Classes and Associations. The following integrity constraints can be specified when a class or an association is defined:

- Inclusion constraints
- Referential constraints
- Surjectivity constraints
- Uniqueness constraints
- Constancy constraints
- General triggers on insertion and removal

A detailed description of these constraints is reported in (Albano et al., 1991). Hereafter, the discussion will be limited to the inclusion constraint on classes, to the referential and surjectivity constraints on associations, and to general triggers on insertion and removal.

Inclusion Constraints. Classes can be organized into an inclusion hierarchy, which means that the elements of a class are a subset of those of its superclasses (inclusion constraint). Moreover, subsets of the same class can be defined as disjoint.

To this aim, the are inclusion constraint is used, which means that elements inserted into a class also are automatically inserted into their superclasses; whereas elements removed from a superclass also are automatically removed from their subclasses. Here is an example of three subclass definitions:

The **butNot** disjointness constraint means that the insertion of an element in a subclass fails if the element already is present in another subclass. Here is an example:

Referential Constraints. The referential constraint specifies the fact that a component of an association must belong to a given class C. In general, the referential constraint may be violated either because (a) the associated object is not a "valid" (in some sense) object, or (b) the associated object does not belong to C. For example, in relational databases, where the mechanism of external keys is used to model associations, problem (a) is the main concern: the external key may not be associated with any tuple in the database. In Fibonacci, every object found inside an association (or everywhere else) is a valid object, but it may not belong to the intended class; hence, problem (b) is our main concern.

More generally, the kind of referential constraint problem to be dealt with in an object-oriented database system depends essentially on how objects are removed from classes and deleted. *Deleting* an object means that the object ceases to exist (i.e., any reference to it becomes invalid), or it becomes a reference to a "tombstone." Removing an object from a class just means that the object does not belong to the class any more, but it is still a valid object. Object-oriented database systems take two different approaches to deletion and removal:

- A deletion operation is provided, which both removes an object from every class and deletes it.
- No deletion operation is provided. Removing an object from a class does not imply its deletion; the object is deleted behind the scenes by the system only when no more reference exists to the object (garbage collection).

When deletion is provided, problem (a) (invalid object references) arises: how can it be verified that no method, used to model an association, returns a reference to a deleted object? When class removal is provided, problem (b) (objects not belonging to classes) arises: how can it be verified that no method, used to model an association with class A, returns a reference to an object that has been removed from class A? If any procedure can form the body of a method, it is impossible for the system, in both cases above, to decide which kind of object a general procedure would return.

In Fibonacci, however, the referential constraint can be enforced, because associations are not represented by methods but by a specialized mechanism, associations, and the system is fully aware of association semantics. The fact that the referential constraint can be maintained in the presence of removals is an important accomplishment of the object-association data model.

Referential constraints in Fibonacci are associated with some fields of the tuples of an association. We call these fields, submitted to a referential constraint and so constrained to belong to a specified class, association components; and we call the other attributes association attributes (or just components and attributes, for short). The constraint is specified in the emptyAssoc expression, which creates the association by writing:

```
emptyAssoc of [...
    label: Type in/are/owned by class;
    ...]
end
```

in, are, and owned by class specify the same referential constraint (the field value must belong to *class*), to be maintained with different styles (i.e., either by raising failures, or by modifying the database to force its satisfaction).

A referential constraint may be violated either when a new tuple is inserted into the association, if the component does not belong to the class, or when an

Table 1. Conditions enforced and operations monitored by the constraints

Constraint	Enforced condition	Monitored operations
referential	$x \in assoc.label \rightarrow x \in class$	insert in assoc, remove from class
surjectivity	$x \in \text{class} \to x \in \text{assoc.label}$	insert in class, remove from assoc

element is removed from a class, if it is a component in a tuple in the association. The label: Type in class constraint raises a failure in both cases.

label: Type owned by class forces a "cascade deletion" of the tuple when the element is removed from the class (the association is "owned" by the class), but raises a failure when a tuple is inserted in the association. So owned by codifies a dependency constraint, more precisely a dependency of the association on the class, where dependency means cascade removal.

label: Type are class means that the projection of the association on label behaves like a subclass of class: a removal from the class forces a removal from the association (as in the owned by case); and an insertion in the association of a component which does not belong to the class forces an insertion in the class (instead of raising a failure as happens with in and owned by). For a summary, see Tables 1 and 2.

Surjectivity Constraints. While the referential constraint specifies that the existence of a tuple in an association implies the existence of a value in a class, the surjectivity (or totality) constraint enforces the converse implication: the existence of elements in a class necessitates the existence of a tuple involving them in the association. The constraint is specified in the emptyAssoc expression by writing:

label: Type onto/owns class

A surjectivity constraint may be violated either when a new element is inserted into the class, if it is not a component of any tuple in the association; or when a tuple is removed from an association, if it was the last that involves an element in the class.

The onto clause corresponds to the in clause: it fails in both cases. Suppose that both a referential and a surjectivity constraint are being defined for a component of an association with respect to the same class, which is quite common (e.g., field composite of association assembly in Appendix A). Then a class element and the first tuple referring to it should be created "at the same time," since each insertion should come before the other one (Table 1). The same is true for class and association removals. For this reason, the surjectivity constraint is not checked immediately after a class insertion or an association removal, but at the end of the smallest transaction that encloses the operation execution. Since the

Constraint	when [let label=x]	when x
	is inserted into assoc	is removed from class
	if $x \notin \text{class}$	if [let label= x] \in assoc
in	fail	fail
owned by	fail	remove assoc where label=x
are	insert (x) in class	remove assoc where label= x

Table 2. Action requested by referential constraint, before performing insertion/removal operation

language supports nested transactions, this smallest transaction can be made as short as required; the language also supports a general mechanism to define an expression to be executed at the end of the current smallest enclosing transaction (the defer operator). Note that a similar problem occurs when a couple of classes are associated by an association which is surjective (total) on both of them. In this case, two related elements have to be inserted in the two classes "at the same time." This is impossible in many languages, which support the object-oriented data model, while it is possible in Fibonacci, thanks to the delayed checking of surjectivity.

The owns clause is the surjectivity counterpart of the referential owned by clause: like onto, when an element is inserted into the class, then the operator fails if in the same transaction a tuple referring to that element has not been inserted into the association. However, when the last tuple referring to an element is removed from an association, then that element is automatically removed from the class at the end of the transaction. So owns codifies a dependency constraint, more precisely the dependency of a class on an association. Dependency of a class B on a class A through an association AB can be expressed, in this language, by saying that AB owns B and that AB is owned by A.

Tables 1, 2, and 3 summarize the precise relationships among the inclusion, referential, and surjectivity constraints.

General Triggers on Insertion and Removal. Triggers are the most popular mechanism to give a database the capability of reacting to events. In Fibonacci, it is possible to associate an unlimited number of triggers with a class or an association through the constructs **beforeInsert** Expr and **beforeRemove** Expr. The action Expr may be any expression of the language, which is then executed every time an element is inserted into (or removed from) the class or association. Expr can access the inserted (or removed) element and the class, or association, through the predefined identifiers thisElement, thisClass, and thisAssociation, as in the example below, where triggers are used to check a key constraint and to maintain a count of class elements.

```
let femaleStudentsCount = var 0;
let femaleStudents =
    emptyClass of Student are students
    beforeInsert assert no x in thisClass have x.Name = thisElement.Name
    beforeInsert femaleStudentsCount := at femaleStudentsCount + 1
    beforeRemove femaleStudentsCount := at femaleStudentsCount - 1
end;
```

Triggers also can be added to previously defined classes and associations. In Albano et al. (1991) it was shown that all the constraints (e.g., inclusion, referential) that can be declared for classes and associations may be defined using the trigger mechanism; in the current Fibonacci implementation, class and associations constraints actually are implemented in this way.

The behavior of triggers can be presented using the dimensions suggested by Fraternali and Tanca (1994) to characterize active database semantics.

- Granularity: Triggers are activated for each insertion or removal operation. If several triggers are defined for the same event, they are activated in the definition order.
- Coupling modes: Usually the action defined in a trigger is executed as soon as
 an insertion or removal operation is executed (immediate coupling). However,
 by exploiting the language construct defer Expr, it also is possible to specify
 that the action must be executed at the end of the smallest transaction
 containing the operation that has activated the trigger (deferred coupling).
- Atomicity of rule execution: The action of a trigger may generate new events which trigger other actions. When a new event is generated, the current action is suspended, and is resumed when the triggered actions have been completed.
- Relationship to transactions: The action of a trigger is executed in the same transaction where the triggering event arises.
- Conflict resolution: If an event activates several triggers, they are executed serially in the order in which they are defined.
- Event consumption: The triggered event no longer activates the processed rule, but may still trigger different rules.
- Transaction history inspection: In the action part of a trigger, the state of data before the execution of the action can be inspected with the construct old.

5.2 Sequence Type

Sequences are ordered collections of homogeneous values with duplicates. Fibonacci query algebra operators are defined on sequences, and may be applied to classes and associations, too, thanks to subtyping.

Fibonacci query algebra is characterized by the following features:

Constraint		when [let label=x]
	is inserted into class	is removed from assoc
<u> </u>	if (let label= x) $\not\in$ assoc	if $x \in \text{class}$
onto	fail	fail
owns	fail	remove y from class where $y = x$

Table 3. Action requested at commit time by surjectivity constraint, when insertion/removal operation performed

- Although it may be used in an SQL fashion, it is not based on a select-fromwhere operator (for example, as happens with O₂; Bancilhon et al., 1989), but on a set of atomic algebraic operators, which may be combined in many different ways, and which give more flexibility to the algebra.
- No attention has been paid to minimality. Although it is well known that
 essentially all of the algebraic operators may be defined in terms of the
 fold (also called pump) operator typical of functional languages with lists
 (Breazu-Tannen et al., 1991), in Fibonacci product, projection, selection, and
 iteration are defined as different operators. This improves both language
 usability and optimizability.

The notation $\{T\}$ denotes the type of a sequence of elements of type T, and $\{E_1; ...; E_n\}$ denotes the sequence containing $E_1, ..., E_n$. Several operators are defined on sequences, and the presentation will be focused on those that constitute the Fibonacci query algebra.

The typical Fibonacci query has the following form:

```
for (ide1 in SeqExpr1 times ... times iden in SeqExprn)
where BoolExpr(ide1,...,iden)
do Expr(ide1,...,iden)
```

which essentially evaluates $\operatorname{Expr}(\operatorname{ide}_1, \ldots, \operatorname{ide}_n)$ for all the tuples $\operatorname{ide}_1, \ldots, \operatorname{ide}_n$ of values from $\operatorname{SeqExpr}_1 \ldots \operatorname{SeqExpr}_n$ that satisfy the condition $\operatorname{BoolExpr}(\operatorname{ide}_1, \ldots, \operatorname{ide}_n)$. However, for-in-times-where-do is not a single construct, but just a typical way of combining different atomic operators, and each of these operators has its own elementary semantics. We now give the semantics and examples for each of these operators, using the sequences employees and departments with elements of the following tuple types:

```
Let Employee =
  [empNo: Int;
  deptNo: Int;
  name: [name, surname: String];
  birthYear: Int;
  dependents:[name: String; birthYear: Int; relationship: String]}];
```

```
Let Department =
  [deptNo: Int;
  deptName: String;
  deptBudget: Int;
  mgrNo: Int];
```

Note that classes of objects and associations would be queried in exactly the same way; a nested-relational-like schema has been used since it may be more familiar to the reader.

The query operators are defined as follows.

```
Label in Sequence
```

This returns a sequence of tuples with a unique field named Label, associated with the value of the corresponding element in Sequence.

```
L-TupleSequence times R-TupleSequence
L-TupleSequence join R-TupleSequence
```

Both return a sequence of tuples obtained by concatenating each tuple in L-Tuple Sequence with every tuple in R-TupleSequence. If the tuples in L-TupleSequence and R-TupleSequence have attributes with the same name, times fails, while join only concatenates in the result those tuples with the same value in these attributes (natural join). The tuples in R-TupleSequence to be concatenated with each tuple t in L-TupleSequence can be defined in terms of t (dependent product and dependent natural join). For example:

```
(emp in Employees) times (dep in emp.dependents)
```

creates a sequence of two-field tuples, containing one tuple for each employee-dependent pair. This is possible since, in R-TupleSequence, the labels of the tuples in TupleSequence (emp is the only such label, in this example) can be used to denote the corresponding field.

More generally, the expression which is the second operand of the operators on sequence of tuples (e.g., times, join, where, for-do, group by, all, some), can always directly access the fields of the tuples in the first operand.

```
TupleSequence where BoolExpr
e.g.: employees where birthYear = 1960
    (e in employees) where e.birthYear = 1960
```

This returns the sequence of the tuples in TupleSequence that satisfies the condition BoolExpr.

```
for TupleSequence do Expr
e.g.: for employees do name;
    for e in employees do e.name
```

This returns the sequence obtained by evaluating Expr once for each element of TupleSequence.

```
the SeqExpr
e.g.: the (employees where empNo=132)
```

This returns the only element of the singleton SeqExpr and fails if count (Sequence) is different from one.

```
setof Sequence
e.g.: setof (for employees do {name.name})
```

This eliminates duplicates from Sequence.

```
all Sequence have BoolExpr
some Sequence have BoolExpr
e.g.: all employees have birthYear > 1900;
    some e in employees have isEmpty(e.dependents)
```

all and some verify whether a condition holds for all elements or, respectively, for at least one element of a sequence of tuples.

```
TupleSequence group by (Expr) pick Sequence
```

group by returns a partition of the elements in TupleSequence in subsequences; two tuples are put in the same partition whenever the expression Expr has the same value for both of them. pick returns one element of Sequence, chosen non-deterministically. It usually is used in conjunction with group by. For example, the query:

```
for empGroup in (employees group by birthYear)
do [ let year := pick(empGroup).birthYear;
    let count := count(empGroup);
    let employees := empGroup]
```

partitions the employees according to the birth year and, for every birth year, returns a tuple containing the birth year, the number of employees with that birth year, and the sequence of those employees. The **where** operator allows us to restrict both the employees and the employee groups to be considered, as in the following example:

6. The Architecture of the Fibonacci System

The first implementation of Fibonacci currently is being completed. It is a singleuser system with a minimal programming environment, and is used to experiment with the language. The system is organized into three layers: the Compiler, the Persistent Hierarchical Abstract Machine, and the Persistent Store, described in the next three sections.

6.1 The Compiler

The main tasks of the compiler are to typecheck Fibonacci expressions and to generate the corresponding PHAM code, which is then executed by the PHAM (Section 6.2).

The compiler is written in Modula-3, and is structured in an object-oriented way. For any syntactic construct of the language, a corresponding object type exists. Hence, any node in an abstract syntax tree is represented by an object, which refers to the objects representing its children in the tree. Each object class has its own Parse, Type-Check, and Code-Generate methods, which typically use the corresponding methods of the children: for example, an object modeling an "if b then e1 else e2" node refers to the objects modeling b, e1, and e2, and uses their Code-Generate methods to generate the code for the whole construct. This structure makes it easier to add and remove new constructs to and from the language, since all the code that is related to a construct is collected in the corresponding object class.

In Fibonacci all information, not only data, but the environment (i.e., the current associations of identifiers to values and types) and the types, are persistent. For this reason, the compiler does not use Modula-3 variables to store this information, but they are encoded as PHAM data structures and stored inside the persistent store. The availability of this information inside the persistent store means that browsing tools can be written that access this information by connecting themselves directly to the store, with no need to interact with the compiler. One such tool that allows browsing data, sending messages to objects, and browsing the environment and the types, already is part of the current system. We plan to reimplement the system, in a future version, using Fibonacci itself. Thus, the compiler information would be stored inside the persistent store without having to rely on any interface to manipulate it.

6.2 The Persistent Hierarchical Abstract Machine

The PHAM has two main tasks. First, it builds the more complex and specialized Fibonacci data structures on top of the primitive and general purpose data structure supported by the persistent store. Moreover, it executes the compiled Fibonacci code, and stores it as first class data.

The PHAM is a stack machine that manipulates data and closures (i.e., compiled functions). It is derived from the Stack-Environment-Code-Dump (SECD) machine (Landin, 1964) and, more directly, from the FAM (Cardelli, 1983). A piece of PHAM code is a linear sequence of zero-address operations that take their arguments from, and put their results to, the stack. The PHAM is characterized, with respect to other similar persistent SECD-machines used to implement persistent or volatile functional programming languages, by features associated with its interface and implementation.

Most SECD-machines are designed to offer a minimal set of operations and a minimal set of very general data types, so that the machine is general and small. PHAM, on the other hand, offers higher level data types and much more complex operations, which closely correspond to Fibonacci. This difference is due to a change in perspective about optimization. In temporary functional languages, optimization is a compiler task, hence the compiler emits a low level code. In persistent systems, some kinds of optimization decisions, such as access plan generation for queries, have to be dealt with at run time, hence a query must not be transformed into a sequential program by the compiler, but a query construct must be part of the abstract machine language.

PHAM is characterized by its architecture and by the fact that it does not have any local store, but completely relies on the persistent store for this. The internal architecture of PHAM is represented in Figure 9.

The PHAM is divided into two layers. The higher layer is a PHAM interpreter, which executes the PHAM code. More precisely, the PHAM interpreter directly executes control flow operations and stack manipulation and, when it recognizes an operation that is specific to a data type, it invokes the corresponding module. This approach was chosen to make it easier to define new data types and to test various implementations for them. Furthermore, this means that we still have a small general purpose abstract machine (the interpreter) with a manageable complexity, plus a set of data type modules which are "external" with respect to this kernel.

6.3 The Persistent Store

The persistent store manages a persistent collection of data items, where a data item is a string of uninterpreted bytes, and of references to other data items. The store also offers a mechanism for atomically saving a frozen version of its state and for restoring the last frozen state in case of failure. This mechanism is used by the PHAM to realize the outer level of nested transactions.

Since the PHAM directly stores all its data inside the persistent store, without trying any kind of caching or special treatment of volatile data, the persistent store must be able to manage temporary data with an efficiency that is comparable to

^{7.} Landin's SECD has only one data type, the closure, and three operations: stack access, λ -abstraction, and application.

Figure 9. PHAM architecture

Data types modules Classes modules constraint services associations conjects ::

the one of the central memory. This is a very strong requirement. However, the alternative choice of having the PHAM realizing a buffering schema on top of the persistent store was discarded to avoid the risk of having two different and contrasting buffering schema built one atop the other, with the consequent well known problems.

Another design choice is to not build the persistent store as part of the Fibonacci project, but to base it on a component built elsewhere, to avoid duplicating efforts in a field that is being widely investigated. For this reason, the operation set required from the persistent store is minimal, to allow tests and comparisons involving different persistent stores. This minimality choice clearly has drawbacks, since it does not allow us to exploit the more advanced features of the available stores. Our current realization relies on the Napier store (Brown, 1989). We are currently experimenting with the O_2 engine.

7. Other Works

7.1 Object-Oriented Database Languages

Like other database languages, Fibonacci has been designed to overcome the limitations of traditional systems due to the lack of integration of the data model with the programming language (impedance mismatch), and the unsuitable modeling capability of the data model. Examples of these languages are Taxis (Mylopoulos et al., 1980), Galileo (Albano et al., 1985), Statice (Symbolics, 1988), O₂ (Bancilhon et al., 1992), Orion (Kim, 1990; Banerjee et al., 1987), Gemstone (1986), Vbase and Ontos (Ontologic, 1987, 1989). Below is a brief comparison of Fibonacci with two other well known object-oriented database programming languages, O₂ and ORION, to show the different solutions offered to model the following aspects: objects, object types, classes, associations between sets of objects, type hierarchies, and class hierarchies.

Objects. The languages all support the notion of objects with identity. State encapsulation is provided only in O2 and Fibonacci. In Fibonacci and O2, all the language values can be either temporary or persistent, because persistence is an orthogonal property of the type system while, in ORION, only objects can become persistent, and the management of versions is provided. In Fibonacci, objects can have roles, and an operator is provided to change the role of an object without affecting its identity. A similar mechanism has been proposed in other languages (Albano et al., 1985; Fishman et al., 1987; Richardson and Schwartz, 1991; Shilling and Sweeney, 1989; Stein and Zdonik, 1989). A detailed analysis of these proposals was made in Albano et al. (1993). The conclusion is that the proposals that support late-binding (Galileo and Iris) always assume the existence of a most specialized type to resolve the message dispatching ambiguities that can arise from type multiplicity. On the other hand, when the previous assumption is abandoned and objects are allowed to have multiple minimal types (Clovers, Views, and Aspects), late-binding never is provided. A novel aspect of Fibonacci is thus the coexistence of late-binding and multiple inheritance with role multiplicity and dynamic object extension, in a framework with strong type-checking and subtyping.

Object Types. The languages all support a construct to define object types with associated methods. Only ORION has the restriction that the state of an object can have components of the following types only: elementary, object, and set of objects. The other languages have a collection of type constructors (e.g., tuple, set) that can be used without restrictions to define a component of the object state. Only in Fibonacci, can a single object type have many implementations. Only the Fibonacci type system provides a constructor to distinguish constant values from modifiable ones, and this has important consequences on the possibility of statically typechecking definitions and applications.

ORION allows the definition of composite objects; that is, objects that have other objects as components with the possibility of enforcing the constraints that (a) an object is a part of only one object, and (b) the existence of an object depends on the existence of its parent object. Fibonacci can express the same constraints by exploiting the association mechanism.

Methods are defined differently in the three languages. In ORION and O_2 , the method signature is specified in an object type definition, and the body of a method is specified separately as a Common LISP or O_2C function with the object type to which the method belongs as the type of the first parameter. In Fibonacci, the body of a method is specified with the object constructor. Only Fibonacci is a statically and strongly typed, polymorphic language.

Classes. Fibonacci and O_2 provide two separate mechanisms to deal with objects and sets of objects, here called classes; objects are explicitly inserted and removed from a class. As in most object-oriented database languages, in ORION an object type definition entails automatically defining a variable with the same name to denote all the objects of that type.

Associations. In O_2 and ORION, associations between objects are modeled only by aggregation (i.e., by the definition of objects that have other objects as components). In Fibonacci, associations relate objects that exist independently, and they can be defined incrementally without redefining the structure of existing objects.

Type Hierarchies. In all these languages, object types can be organized into hierarchies, and the benefits of the inheritance mechanism can be exploited. However, only the Fibonacci type system has been designed for a statically and strongly typed language.

Class Hierarchies. Classes can be organized into a subset hierarchy with the following properties: (a) the type of the subclass elements is a subtype of the type of the superclass elements, and (b) the extension of a subclass is a subset of the extension of the superclass. A subclass can be defined from a single superclass, or from several superclasses, and can be populated by creating new elements, and also by moving objects from a superclass into the subclass. Class hierarchies are treated differently in object-oriented database languages. O₂ does not provide this mechanism. ORION provides disjoint subclasses, except when a class is defined by multiple inheritance, and subclasses cannot be populated by moving objects from a superclass into the subclass. A construct is provided to access the element of a class without considering those elements that also belong to a subclass. Fibonacci provides subclasses in the most general form with the option of moving objects from a superclass into a subclass.

7.2 Persistent Languages

Persistent languages are languages where persistence is a property that is orthogonal to types. Fibonacci differs from other persistent languages, such as PS-Algol, Napier and Tycoon, since it is also a database language (i.e., it directly supports a specific data model). The scope of Fibonacci research currently is twofold: in the field of data modeling, to design a data model that is richer than current object-oriented data models, and in the field of type systems, to extend the expressivity of the type system of current languages.

The design goal of the Napier and Tycoon persistent languages is slightly different. Both support the add-on approach for data modeling (i.e., they do not provide direct support for a specific data model, but provide programmers with the tools they need to define their own data model). The main tools provided by the Tycoon language (Matthes and Schmidt, 1994) for this aim are:

- a rich type system, which is indeed very similar to the Fibonacci type system, in that both derive directly from the Quest type system (Cardelli, 1990);
- a rich sublanguage for module and library management;
- a mechanism to extend Tycoon syntax;
- a linguistic mechanism and an implementation style, which allow Tycoon to be used as a gateway to operate on different systems; a similar mechanism

was developed for the Galileo system and is present in Fibonacci, but the Tycoon project deals with this aspect with much more attention.

The Napier project has a yet different aim. While Tycoon focuses on openness (i.e., extendibility and interoperability), the Napier project tries to define a complete Napier based environment where the persistent Napier store becomes the repository of all the data, even replacing the file system, and every tool is written in Napier88 itself, taking the maximum advantage from the persistence of the language and from the integration of the environment. The main mechanisms offered by the Napier88 language to fulfill this project are (Morrison et al., 1994):

- a rich type system. It lacks inclusion and parametric polymorphism; however, it has an environment construct that provides for a form of controlled dynamic typechecking, which is very useful for dealing with database evolution.
- A reflection mechanism, which gives the system the ability to generate and then evaluate the Napier88 code.

8. Conclusions

This article is an introduction to the Fibonacci database programming language. We have presented the most important language features for modeling databases, but we have omitted other language features, as well as several details on the concepts discussed. Further information on the object mechanism may be found in Albano et al. (1993), on the class mechanism in Ghelli (1990) and Albano et al. (1991), and on the association mechanism in Albano et al. (1991b). The informal definition of the language may be found in Albano et al. (1994b).

The implementation of the language is in progress. The language is implemented as an interactive compiler written in Modula-3 and running on Sun workstations and on MS-DOS machines. The present implementation does not support concurrent access to persistent data. The persistence of data is achieved using the persistent object store of the Napier 88 language, developed at St Andrews University. A description of the Fibonacci system and the features of the programming environment may be found in Albano et al. (1994a).

The research in progress on the Fibonacci language addresses the following aspects: a modularization mechanism, constructs for viewing and schema evolution, and mechanisms to connect Fibonacci with other languages.

A modularization mechanism is needed to be able to partition big software projects into manageable units (modules) with an interface that indicates only the types of the values that are exported by each unit. The Fibonacci proposal will be based on the idea that modules are first class values and module interfaces are first class types. In this way, it is possible to define a very flexible module mechanism by adding a small amount of new constructs to the language.

Viewing and schema evolution mechanisms, such as the view and alter table constructs of SQL, must be present in every database programming language,

because the same piece of information may best be viewed in different ways in different contexts, and the structure of a database naturally evolves with time. The traditional SQL approach cannot be directly transposed in a persistent language such as Fibonacci for many different reasons. The most important is that, in a persistent language, there is not an exact equivalent to the notion of a "database schema," which only describes the structure of classes. The "environment" of a persistent language merges the definition of types, classes, functions, and simple variables. Hence, a schema evolution mechanism must manage, in a linguistically consistent way, the coherent evolution of all of these pieces of information. Moreover, in a statically and strongly typed language, it is important that the viewing and evolution mechanisms do not break the strong typing of the system. First steps in this research direction have been taken by Albano et al. (submitted), where a view mechanism for objects is described.

Finally, mechanisms to connect Fibonacci to other languages are needed to allow Fibonacci programmers to access various software packages and data managed by different systems. The current Fibonacci system contains a mechanism that allows any package with a C language interface to be dynamically linked to the Fibonacci system, and its constants, variables, and functions to be accessed as if they were predefined Fibonacci functions. The mechanism will be generalized, mainly to allow more languages to be accessed, more parameter passing techniques to be exploited, and to allow Fibonacci functions to be passed as call-back parameters.

Acknowledgements

This work was supported in part by grants from the CEC under ESPRIT BRA No.6309 (FIDE2: Fully Integrated Data Environment), the Progetto finalizzato "Sistemi informatici e calcolo parallelo" of CNR under grant No.93.1502.PF69, and by Ministero dell'Università e della Ricerca Scientifica e Tecnologica. The design and implementation of Fibonacci were carried out jointly by the University of Pisa and Engineering S.p.A. In addition to the authors, other project members were R. Bergamini, M. Diotallevi, A. Marrara, C. Brasini, R. Rossi, M. Pieragnoli, and L. Vinciotti. D. Blakeman gave an essential contribution to the success of the implementation.

References

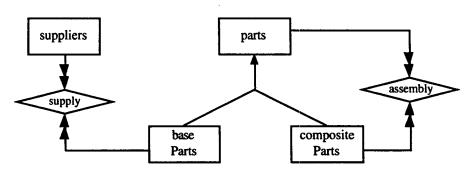
- Albano, A. Type hierarchies and semantic data models. ACM SIGPLAN Symposium on Programming Languages Issues in Software Systems, San Francisco, 1983.
- Albano, A., Antognoni, G., and Ghelli, G. View operations on objects with roles. Submitted for publication.
- Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. An object data model with roles. *Proceedings of the Nineteenth International Conference on VLDB*, Dublin, 1993.

- Albano, A., Brasini, C., Diotallevi, M., Ghelli, G., Orsini, R., and Rossi, R. A guided tour of the Fibonacci system. FIDE2 Technical Reports, FIDE/94/103, 1994a.
- Albano, A., Cardelli, L., and Orsini, R. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230-260, 1985. Also in: Zdonik, S.B. and Maier, D., eds. *Readings in Object-Oriented Database Systems*, San Mateo, CA: Morgan Kauffman, 1990, pp.147-161.
- Albano, A. and Ghelli, G. Object-oriented data base programming languages, invited paper. *IEEE EuroComp Conference*, Bologna, 1991.
- Albano, A., Ghelli, G., and Orsini, R. A relationship mechanism for a strongly typed object-oriented database programming language. *Proceedings of Seventeenth International Conference on VLDB*, Barcelona, 1991.
- Albano, A., Ghelli, G., and Orsini, R. Fibonacci reference manual: A preliminary version. FIDE2 Technical Reports, FIDE/94/102, 1994b.
- Atkinson, M.P. A vision of persistent systems. Invited paper. Proceedings of the International Conference on Deductive and Object-Oriented Databases, Munich, 1991.
- Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R. An approach to persistent programming. *Computer Journal*, 26(4):360-365, 1983.
- Atkinson, M.P., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S.B. The object-oriented database manifesto. *Proceedings of the DOOD*, Kyoto, Japan, 1989.
- Atkinson, M.P. and Buneman, O.P. Types and persistence in database programming languages. ACM Computing Surveys, 19(2):105-190, 1987.
- Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. PS-algol: An algol with a persistent heap. ACM SIGPLAN Notices, 17(7):24-31, 1981.
- Bancilhon, F., Cluet, S., and Delobel, C. A query language for the O₂ objectoriented database system. *Proceedings of the Second International Workshop on Database Programming Languages*, Salishan Lodge, Oregon, 1989.
- Bancilhon, F., Delobel, C., and Kannelakis, P., eds. Building an Object-Oriented Database System: The Story of O₂. Menlo Park, CA: Morgan Kaufmann, 1992.
- Banerjee, J., Chou, H., Garza, J.F., Kim, W., Woelk, D., and Ballou, N. Data model issues for object-oriented applications, *ACM Transactions on Office Information Systems*, 5(1):3-26, 1987.
- Breazu-Tannen, V., Buneman, P., and Naqvi, S. Structural recursion as a query language. *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, 1991.
- Brown, A.L. Persistent object store. Persistent programming, Research Report 71-89, Department of Computer Science, University of Glasgow and St. Andrews, Scotland, 1989.
- Cardelli, L. The functional abstract machine. AT&T Bell Labs Technical Report N. 107, Murray Hill, NJ, 1983.

- Cardelli, L. The Quest language and system (tracking draft). Technical Report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, 1990.
- Cardelli, L. and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471-523, 1985.
- Connor, R., McNally, D., and Morrison, R. Subtyping and assignment in database programming languages. *Proceedings of the Third International Workshop on Data Base Programming Languages*, Nafplion, Greece, 1991.
- Dittrich, K. Object-oriented database systems: The next miles of the marathon. *Information Systems*, 15(1):161-167, 1990.
- Fishman, D.H., Beech, D., Cate, H.P., Chow, E.C., Connors, T., Davis, J.D., Derrett, N., Hoch, C.G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M.A., Ryan, T.A., and Shan, M.C. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48-69, 1987.
- Fraternali, P. and Tanca, L. A structured approach for the definition of the semantics of active databases. Politecnico di Milano, Dipartimento di Elettronica e Informazione, Report 94-073, 1994.
- Gemstone Product Overview, GemStone Version 1.0, Servio Logic Development Corporation, March 1986.
- Ghelli, G. A class abstraction for a hierarchical type system. *Third International Conference on Database Theory*, Paris, 1990.
- Kim, W. Introduction to Object-Oriented Databases, Cambridge, MA, The MIT Press, 1990.
- Landin, P. The mechanical evaluation of expressions. *Computer Journal*, 6:308-320, 1964.
- Matthes, F. and Schmidt, J.W. Bulk-types: Built-in or add-on. *Proceedings of the Third International Workshop on Data Base Programming Languages*, Nafplion, Greece, 1991.
- Matthes, F. and Schmidt, J.W. Persistent polymorphic programming in Tycoon: An introduction. FIDE2 Technical Reports, FIDE/94/106, 1994.
- Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., and Munro, D.S. The Napier88 Reference Manual (Release 2.0), FIDE2 Technical Reports, FIDE/94/104, 1994.
- Mylopoulos, J., Bernestein, P.A., and Wong, H.K.T. A language facility for designing interactive database-intensive systems. *ACM Transactions on Database Systems*, 5(2):185-207, 1980.
- Ontologic Inc., VBASE System Documentation, Billerica, MA., 1987.
- Ontologic Inc. ONTOS Object Database Documentation, Release 1.5, Burlington, MA, 1989.
- Richardson, J. and Schwartz, P. Aspects: Extending objects to support multiple, independent roles. *Proceedings of the International Conference on the Management of Data*, Denver, CO, 1991.

- Rowe, L.A. and Shoens, K.A. Data abstraction, views and updates in RIGEL. *Proceedings of the ACM SIGMOD Conference*, Boston, MA, 1979.
- Schmidt, J.W. Some high level language constructs for data of type relation. ACM Transactions on Database Systems, 2(3):247-261, 1977.
- Schmidt, J.W. and Matthes, F. Language technology for post-relational data systems. Database Systems of the 90s, Vol. 466 of Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1990.
- Shilling, J.J. and Sweeney. P.F. Three steps to view: Extending the object-oriented paradigm. OOPSLA 89, ACM SIGPLAN Notices, 24(10):353-361, 1989.
- Smith, J.M., Fox, S., and Landers, T. Reference Manual for ADAPLEX, CCA-81-02, Computer Corporation of America, January, 1981.
- Stein, L.A. and Zdonik, S.B. Clovers: The Dynamic Behavior of Type and Instances. Brown University Technical Report No. CS-89-42, November, 1989.
- Symbolics Inc., "StaticeTM," Cambridge, MA 02142, USA, 1988.
- Wasserman, A.I. The data management facilities of PLAIN, *Proceedings of the ACM SIGMOD Conference*, Boston, MA, 1979.
- Zdonik, S.B. and Maier, D. Fundamentals of object-oriented databases. In: Zdonik, S.B. and Maier, D., eds. *Readings in Object-Oriented Database Systems*, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

Figure 10. The Parts example



Appendix A

A database example

Let us consider the classical example of parts and suppliers, with parts defined recursively by other parts, either composite or base. The scheme is shown in Figure 10.

Here is the Fibonacci scheme:

```
(* object and role type definitions *)
   Let PartObject = NewObject;
   Let SupplierObject = NewObject;
(* role types *)
   Let Part =
      IsA PartObject With
         Name: String;
         Cost: Int;
         Mass: Int
      End;
   Let BasePart =
      IsA Part With End;
   Let CompositePart =
      IsA Part With
         AssemblyCost: Int
      End;
   Let Supplier =
      IsA SupplierObject With
         Name: String;
```

```
Address: String
      End:
(* classes *)
   let parts =
       emptyClass of Part
       key Name
         elsefail "parts key: duplicated Name"
       end:
   let baseParts =
       emptyClass of BasePart
         are parts
       end:
   let compositeParts =
       emptyClass of CompositePart
         are parts butNot baseParts
       end;
   let suppliers =
       emptyClass of Supplier
       key Name
         elsefail "suppliers key: duplicated Name"
       end;
(* associations *)
(* a base part cannot exist without a supplier for it *)
   let supply =
       emptvAssoc of
        [ basePart :BasePart are baseParts
           onto baseParts
             elsefail "supply: basePart onto baseParts";
         supplier : Supplier in suppliers ]
       end:
   let assembly =
       emptyAssoc of
         [ composite : CompositePart are compositeParts
           onto compositeParts
             elsefail "assembly: composite onto compositeParts";
         component : Part in parts
             elsefail "assembly: component in parts";
         quantity : Int ]
        key composite, component
        elsefail "assembly key: duplicate pair composite component"
        end;
```

```
(* auxiliary constructors used to create values *)
   let createBasePart =
       fun(aName :String; aCost, aMass :Int) :BasePart is
        begin
         assert not (isUnknown aName)
          elsefail "createBasePart: Name cannot be unknown";
         role BasePart
         methods
          Name = aName;
          Cost = aCost:
          Mass = aMass
         end:
        end:
   let createCompositePart =
       fun(aName :String; anAssemblyCost :Int) :CompositePart is
        begin
         assert not (isUnknown aName)
          elsefail "createCompositePart: Name cannot be unknown";
         role CompositePart
         methods
         Name = aName;
          AssemblyCost = anAssemblyCost;
          Cost = anAssemblyCost +
             sum(for assembly where composite = me
              do component.Cost*quantity);
          Mass = sum(for assembly where composite = me
              do component.Mass*quantity)
         end:
        end:
   let createSupplier =
       fun(aName, anAddress:String):Supplier is
         assert not (isUnknown aName)
          elsefail "createSupplier: Name cannot be unknown";
         role Supplier
         methods
          Name = aName;
          Address = anAddress
         end
        end:
(* functions to insert elements in classes which are not
domain of surjective associations *)
   let insertIntoSuppliers =
```

```
fun(aName, anAddress :String) :Supplier is
         let newSupplier = createSupplier(aName; anAddress);
         insert newSupplier into suppliers;
         newSupplier
        end:
(* functions to insert elements in classes which are domain
of surjective associations *)
   let insertIntoBasePartsAddSuppliers =
       fun(suppliers:{Supplier};aName:String;aCost,aMass:Int):BasePart
is
        atomic
         let bp = createBasePart(aName; aCost; aMass);
         loop s in suppliers do
          insert [bp; s] into supply
         end:
         bp;
        end;
   let insertIntoCompositePartsAddComponents =
       fun(someComponents :{[component: Part; quantity: Int]};
         aName :String; anAssemblyCost :Int): CompositePart is
        atomic
         let newCompositePart =
          createCompositePart(aName; anAssemblyCost);
         loop someComponents do
          insert [newCompositePart; component; quantity] into assembly
         end;
         newCompositePart
        end:
(* function to add a new supplier for a base part *)
   let addSupply =
        fun(aBasePart :BasePart; aSupplier :Supplier): Null is
        insert [aBasePart; aSupplier] into supply;
(* query examples *)
(* find the base parts which cost more than 100.000 *)
baseParts where Cost > 100000;
(* result type :{BasePart} *)
(* find Name, Cost and Mass of the base parts which cost more than 100000
   for baseParts
   where Cost > 100000
   do [let Name = Name; let Cost = Cost; let Mass = Mass];
```

```
(* result type :{[Name :String; Cost :Int; Mass :Int]} *);
   (* Find the cost of part "FunBike" *)
   (the (parts where Name = "CityBike")).Cost;
   (* result type :Int *);
   (* find the names of base parts supplied by Alfred *)
   for supply
   where supplier.Name = "Alfred"
   do basePart.Name;
   (* result type :{String} *)
(* a recursive function to find all the base parts composing a part *)
   let findBaseComponents = fun(aPart: Part): { BasePart } is
         if aPart isAlso BasePart
         then { aPart as BasePart}
         else setOf (flatten(for assembly where composite = aPart
              do findBaseComponents(component) ))
         end:
(* find the base parts of the part named "FunBike" *)
findBaseComponents(the (parts where name = "FunBike"));
```