An Introduction to Fibonacci:

A Programming Language for Object Databases

A. Alban, G. Ghelli, R. Orsini University of Pisa

Presented by Sudipto Ghosh

1. Object Oriented Database Programming Language

2. Static and Strong Typing

3. New mechanisms for modeling databases in terms of Roles, Classes, Associations

Design

- a lot drawn from Galileo - a strongly typed database programming language
- overcomes limitations for modeling objects and associations in OO languages

Requirements - Why a new language

1. Database Language

data stored in the database manipulated in the same way as any data structure

2. Persistent Language --- Orthogonal Persistence, Persistence Independence

Matter of debate - which is better?

Built-in Approach - P L offers a set of built-in DD and DM operations

Add-on approach - Gives programmers the possibility of defining data model

Fibonacci takes the built-in approach.

3. Statically and Strongly typed language

4. Object-oriented language

5. Objects and Roles

Standard Object-Oriented model is unsatisfactory for modeling entities that change their class and behavior

Different and inconsistent behaviors may have been defined for different types

Solution: Prevent mixing of inconsistent behaviors

Each object has context dependent behavior

In each context it exploits only a consistent subset of its possible behaviors

This subset depends on the “role” the object is playing in that context

6. Object Association Data Model

Extend OO-data model with a specific construct to model associations.

Standard OO data models

Classes - model sets of homogeneous entities

Association between Objects - method in each that returns the other one

Cardinality constraints enforced by using types

Problems:

a. associations are high level abstract notions - implementation should be decided by DBMS, not by programmer, but attributes force the programmer to choose a specific implementation

b. Association - a single conceptual entity whose semantics is split among several objects

c. enforcement of inverse relation constraint left to programmer

d. objects exist independently and it should be possible to define associations incrementally without having to redefine the structure of existing objects

e. associations are not necessarily binary and can have their own attributes

f. operations on relationships not possible in a straight forward manner

Association is a first-class type constructor.

- Modifiable set of tuples used to represent the existence of associations between classes of objects
- resembles the n-m associations represented in relational schemas
- a tuple consisting of the two objects to be associated is inserted in the association
- referential and integrity constraints can be defined

Overview

1. Expression based language - constructs apply to values and return values

2. Persistent language - all data transitively accessible from the top level environment survive during sessions. Garbage collection performed.

3. Interactive language - prompt for input and report result. Stand-alone applications can also be written.

4. Higher-order language - functions can be embedded in data-structures, passed as parameters and returned

5. Safe - expressions have at least one type, statically checked

- Basic types - None, Null, Bool, String, Int, Real, Any
  - Instances of all basic types disjoint, except the value unknown
- Type constructors -
  - Concrete type constructors
    - (tuple, sequence, discriminated union, function, class, association types)
  - Generative Type constructors (object and role types)
- Type Equality -
  - Structural equivalence for concrete types
  - Name equivalence for generative types (by definition time?)

6. All types are first-class.

7. Sharing semantics for assignment, not copying

8. Subtyping for concrete types

9. Inheritance

10. Polymorphic functions - both inclusion (subtyping) and parametric

11. Exception-trap mechanism

12. Nested transactions

13. Objects with roles

Object identity immutable but state mutable

Objects - software entity with an internal state

Methods - local operations that manipulate the state

Message - request for an object to execute an operation

When a message is sent to an object Student:

inheritance - method lookup in the super-role Person

message passing - method of the Student role can send a message to the Person role

Classical Model

Every object has one most specialized role

Role receives message and starts method lookup

Extensions in Fibonacci

Object extensibility - object can have different specialized roles

Messages sent to roles not objects

Cousin Independence preserved by role mechanism

Cousin type

no type is a subtype of the other but are subtypes of a common ancestor

Don’t have to know the existence of cousin types, need only supertypes.

Object and Role Type Hierarchy

PersonObject — Object Type

Person

Student Employee — Role Types (similar to Object type in traditional OO language)

Objects and Roles: Type Operators

New Object type constructor - `NewObject`

constructor of new object type needed to begin construction of a role type family

```
Let PersonObject = NewObject;
```

New Role type definition - `IsA ... With ... End`

constructs a role type as a subtype of an object type or subtype of other role types

Both are generative type constructors

Role type defined with a set of properties

(Page 12 - subtype of object type; page 13 - of role type)

Inheritance - subtype inherits all the properties of the supertype unless overridden

Multiple Inheritance

If

- Property present in more than one supertype
- there is no explicit redefinition in the subtype

then property of the last specified supertype is inherited.

- Its type has to specialize all the types of all the inherited versions.
- Allowed only if that property has been defined in a common ancestor.
- Prevents two methods with same name, different meaning being merged into a supertype

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

Contravariance - the signature of P in T is a subsignature of the one in S

Covariance - the output type of P in S is subtype of the one in T

Object Construction

- **`role`** `T <implementation>` **`end`**, creates an object with type T
- implementation specifies

    private state of the object

    body for all the methods defined in the interface.

- The role operator builds an object with one role
- returns a reference to that role of the object. (page 13)

    Private section - specifies private environment that can be accessed by role methods

    Method Section - specifies methods of the role

To create many instances of a role type, define a constructor. (page 14)

```
fun (<arguments>): <type> is <exp>
```

```
let CreatePerson = fun (name, address: String; birthyear: Int):
                        Person is
   role Person

   private
      assert stringLength(address)<=0 elsefail "incorrect address"
      let address = var (address);

   methods
      Name = name;
      BirthYear = birthyear;
      Age = currentYear() - me.BirthYear;
      Address = at(address);
      modAddress (newAddress:String) =
         if stringLength(newAddress) <= 0
         then failwith "incorrect address"
         else address := newAddress
      Introduce = "My  name is " & me.name &
                " and I was born in " & intToString(me.BirthYear);
end;
```

Function definition has different syntax from that of a **method**.

Function - first class value

Method - not a value by itself, but part of a role definition

can be evaluated only by sending a message to the object which it belongs to

Static type of "**`me`**" - the type of expression role in which it is written

when a method is executed by a role inheriting it, **`me`** denotes the role that inherited the method (like self in other OO languages)

<u>By separating object implementation from object types,</u>

<u>Fibonacci allows objects with different implementations but same types</u>

Extension operator: to acquire new roles

`ext ... to ... private ... methods ... end` (page 15)

- Example of object extension (Figure 6.)
- Example of constructor by inheritance (Figure 7.)

Method lookup

Upward lookup: `object!message`

- looked up first in the receiving role
- then in its ancestor roles.
- emphasizes difference between behavior between different roles

Double lookup: `object.message`

- first in all the descendant roles
- then in the receiving role
- finally in ancestor roles
- results in a more uniform behavior

If T and S are cousins, no message sent to S will be answered by method in T

Example: (Page 16)

Object *John* has two roles

`johnAsStudent` (type `Student`) and `john` (type `Person`)

Upward and downward lookup

coincide with `johnAsStudent` (no descendant)

differ for `john`

If `John` is extended to employee

double lookup changes as the last acquired subrole is first looked for

upward lookup remains the same

Upward and double lookup are two forms of late-binding.

Object Comparison, Role Inspection, Role Casting

Object Comparison

- equality operator (`=`) to test whether two objects are the same
- independently of the tole used to access them.
- `johnAsStudent = john;` (returns true)

Role Inspection

- infix predicate **`isAlso`** to test whether an object has a certain role.
- `john` **`isAlso`** `Employee;` (after extension, returns true)

Role Casting

- infix operator **`as`** to coerce an object to one of its possible roles (roles casting)
- Operator fails if the object does not have the specified role.
- **`let`** `johnAsEmployee = john` **`as`** `Employee`
- Combination of casting with strict lookup: `(X` **`as`** `T)!P`
- simulation of traditional send-to-super mechanism
- simulating static binding (page 18)

Testing run-time role type

- **`isExactly`** operator on role values
- `johnAsEmployee` **`isExactly`** `Employee`

BULK TYPES: Class, Association, Sequence

Classes

- Modifiable ordered sets of homogeneous values, used to model sets of entities
- No repeated element allowed
- Possible to define constraints like inclusion and mutual disjointness

Associations

- Modifiable homogeneous ordered sets of tuples
- used to model associations between entities

Sequences

- Constant collections of homogeneous values of any type.
- A class or an association is a subtype of a sequence type.

- Query algebra defined on sequences.
- Thus also available to classes and associations.
- Operators to insert and remove data available only to updatable bulk types.

Operations

Creation

Class - **emptyClass of ... end**

```
let students = emptyClass of Student end;
```

Association - **emptyAssoc of ... end**

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

Insertion

- **insert** `Expr` **into** `Expr`
- Type of value or tuple inserted must be a subtype of the element type of class/association
- No-operation if it is already there

Removal

- **remove** `Identifier` **from** `Expr` **where** `BoolExpr`
- removes all values or tuples which satisfy the `BoolExpr`
- removed elements are not deleted until an access path exists for them.

Both **insert** and **remove** are atomic operations.

Integrity Constraints

- Inclusion Constraints
- Referential Constraints
- Surjectivity Constraints
- Uniqueness Constraints
- Constancy Constraints
- General Triggers on Insertion and Removal

Inclusion Constraints (page 20)

- Organize classes into an inclusion hierarchy

  - elements of a Class are a subset of those of its superclasses
- `are` inclusion constraint is used

  - elements inserted into a class are automatically inserted into superclasses

  - elements removed from superclass are automatically removed from subclasses
- Moreover, can define subsets of same class to be disjoint
- `butNot` disjointness constraint - insertion in a subclass fails if the element is already present in another subclass

Referential Constraints

- component of an association must belong to a given class C
- In general,

  - violation if the associated object is not a valid object

    (external key in rel. db. not associated with any tuple)

    Does not occur in Fibonacci

  - or the associated object does not belong to C

    our main concern.

- Deleting an object:

  object ceases to exist, reference becomes invalid or refers to a “tombstone”

- Removing an object from a class:

  still a valid object but does not belong to the class

- Different approaches:

  - Provide deletion operation, remove object from every class and delete it - can lead to first problem
  - Do not provide deletion. deletion only by garbage collection - can lead to second problem

- constraint specified in the emptyAssoc expression

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

- Same referential constraint

  insertion => component inserted into association must belong to the class

  removal => only components in a tuple in the association can be removed

- but different styles:

  `in`: raises failure in both cases

  `owned by`: Fails when tuple is inserted,

  forces cascade deletion of tuple when element is removed from class

`are`: removal from class forces removal from association

insertion in the association of a component forces an insertion in the class

Surjectivity Constraints

- Existence of elements in a class necessitates the existence of a tuple involving them in the association
- `label: Type` **`onto/owns`** `class`

- violation if:

  - new element inserted into class that is not a component of tuple in an association

  - tuple removed from an association and it is the last which involves an element in the class

- **`onto`**: (corresponds to **`in`**)

  fails in both cases

- **`owns`**: (counterpart of **`owned by`**)

  fails in the first case

  second case: element automatically removed from class

A problem:

If both **referential** and **surjectivity** constraints are present for a component wrt the same class, then the class element and the first tuple referring to it have to be created at the same time.

Surjectivity constraint is not checked immediately after a class insertion or association removal, but at the end of the smallest transaction that encloses the operation (defer operator)

General Triggers on Insertion and Removal

- Capability of reacting to the events
- constructs **`beforeInsert`** *`Expr`* and **`beforeRemove`** *`Expr`*
- action *`Expr`* executed every time an element is inserted/removed into the class/association
- *`Expr`* accesses elements using predefined identifiers

  `thisElement`, `thisClass` and `thisAssociation`
- Properties

  Granularity

  Coupling modes

  Atomicity of Rule Execution

  Relationship to transactions

  Conflict resolution

  Event consumption

  Transaction history inspection

Sequence Types and Query Algebra

- used in SQL fashion
- not based on select-from-where
- based on a set of atomic operators that can be combined in many ways (more flexible)
- operators not minimal
- have product, projection, selection, iteration as separate operators
- (claim) improves language usability and optimizability

Typical query form:

for (ide_1 **in** $SeqExpr_1$ **times...times** ide_n **in** $SeqExpr_n$)

where $BoolExpr(ide_1, \ldots ide_n)$

do $SeqExpr(ide_1, \ldots, ide_n)$

Query operators

1. *Label* **in** *Sequence*

Returns a sequence of tuples with a unique field named *Label*, associated with the value of the corresponding element in *Sequence*

2. *L-TupleSequence* **times** *R-TupleSequence*

L-TupleSequence **join** *R-TupleSequence*

Return a sequence of tuples obtained by concatenating each tuple in *L-TupleSequence* with every tuple in *R-TupleSequence*.

- **times** fails if attributes with the same name exist in both
- **join** only concatenates in the result those tuples that have the same value in attributes having the same name (natural join)

3. **where**: *TupleSequence* **where** *BoolExpr*

- returns the sequence of tuples in TupleSequence which satisfy the condition *BoolExpr*.

```
employees where birthYear = 1960
(e in employees) where e.birthyear = 1960
```

4. **for-do**: **for** *TupleSequence* **do** *SeqExpr*

- the expression SeqExpr is evaluated for all the elements of *TupleSequence*;
- the **for** expression returns a sequence obtained by appending the sequences produced at each iteration.

```
for employees do name
for e.employees do e.name
```

5. **`the`** *`SeqExpr`*

- returns the only element of the singleton *`SeqExpr`*
- fails if count(Sequence) is different from one.

```
the (employees where empNo = 132)
```

6. **`setof`** *`Sequence`* eliminates duplicates from *`Sequence`*

```
setof (for employees do (name.name))
```

7. **`all`** and **`some`**

- verify whether a condition holds for all elements or respectively for at least one element of a sequence of tuples

```
all Sequence have BoolExpr
some Sequence have BoolExpr
```

8. **`group by:`** *`TupleSequence`* **`group by`** `(`*`Expr`*`)`

- 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

9. **`pick: pick`** *`Sequence`*

- returns one element of *`Sequence`* chosen non-deterministically

The expression which is the second operand of the operators on sequence of tuples can always directly access the fields of the tuples in the first operand.

Architecture Of The Fibonacci System

- current implementation
- single user system
- Minimal programming environment for experimentation with the language

System has three layers

- the Compiler
- the Persistent Hierarchical Abstract Machine (PHAM)
- the Persistent Store

The Compiler

- typecheck Fibonacci expressions and generate the corresponding PHAM code
- written in Modula-3, structured in Object-oriented way.
- All information (data, environment, types) is persistent
- Not encoded as Modula-3 variables, but as PHAM data structures and stored in persistent store.

PHAM

- builds on top of the primitive general purpose data structure supported by the persistent store, the complex and specialized Fibonacci data structures.
- Executes the compiled Fibonacci code and stores it as first class data.
- Stack machine that manipulates data and closures
- linear sequence of zero-address operations which take their arguments from and put the results onto a stack.

Persistent Store

- persistent collection of data items
- data item is a string of uninterpreted bytes and of references to other data items
- mechanism for atomically saving a frozen version of state and restoring the last frozen state in case of failure.
- manage temporary data with an efficiency comparable to central memory
- Persistent store not built as part of Fibonacci project, but based on a component built elsewhere (Napier store).