PINOL: A PERSISTENT INFERENTIAL OBJECT ORIENTED LANGUAGE FOR DATABASES

A HH Ngu* Limsoon Wong†

School of Computer Science & Engineering
University of New South Wales
P.O.Box 1, Kensington 2033 NSW, Australia.
email:anne@cs.unsw.oz.au

ABSTRACT: The Persistent INferential Object oriented Language (PINOL) offers a classification scheme based on canonical and non-canonical perspectives of objects. This supports economical and intuitive modeling of multiple roles of objects. PINOL also has a rule mechanism for establishing relationships between objects located in different classes. This provides a declarative and deductive capability in the framework of an object oriented system.

1 INTRODUCTION

An ability to model and support multiple independent roles of objects is desirable. Consider an example. From the perspective of the income tax department, a taxpayer is an individual person. At the same time, a taxpayer can play the role of an employer if he employs a maid at home. He can also be involved in a partnership while being employed by an organization or a company. In many database programming systems, such as O2, Gemstone, and Orion, it is necessary to artificially create several types to model the different roles played by the taxpayer. How to place a particular taxpayer in these systems can be a problem. In addition, these systems, such as O2, Gemstone, Orion, etc. do not allow objects to change type dynamically. This means that changing roles of an object cannot be modeled.

Inter-object relationships are frequently modeled procedurally as a method or as a method or an attribute in a class in an object oriented system. This makes understanding and maintaining these relationships and constraints difficult. For example, the relationship between a student and a course can be modeled by including a course-taken method in student. Then only the student knows about this information. In order to get some global information such as "who is taking what courses", it is necessary to iterate over all students. In PINOL, it is advocated that object oriented language is not declarative enough for querying.

PINOL is designed to improve the above situations. A prototype version obtained by extending C++ has been successfully implemented. In this Report, some features of PINOL that address the two issues above are presented. More details and earlier thoughts about PINOL can be obtained from our DOOD89 and DOOD91 papers.

This paper is organized as follows. Section 2 describes PINOL canonical and non-canonical classification of objects: types and classes. It uses special features called endowed attribute and method for modeling objects with multiple and independent roles. Section 3 describes the kind of inference rules that PINOL supports. The inferencing mechanism includes a feature called test predicate which retain advantages of object orientation such as encapsulation and object identities. This section also contains example declarative specifications of fairly complex recursive relationships. More example PINOL programs are given in Section 4. Finally, in section 5 we present the Summary and related research.

2 PINOL CLASSIFICATION CONSTRUCTS

We differentiate the notions of type and class. A type (PINOL type) is used for providing a basic structure of objects – the canonical perspective. A class (PINOL class) is used for collections of objects with structural extension for context sensitive information as well as information common to all the objects in the class. We call the structural extension endowed attributes and class attributes. Endowed methods and class methods are supported to manipulate them respectively. This is the non-canonical perspective. The canonical perspective of an object is defined by the type of that object. The non-canonical perspective of an object can only be viewed within a particular context, i.e. a class. The non-canonical perspective of an object in a class is defined by the type of that object and the structural extension endowed by that class.
We illustrate the modeling of canonical and non-
canonical perspectives of persons. The type person con-
taining the basic structure of a person, such as name, 
address, etc. corresponds to the canonical perspective 
of a person. A non-canonical perspective may be de-
fined by class employee of type person with endowed at-
tributes such as department, employer, and address. An-
other non-canonical perspective is defined by the class 
employee with endowed attributes employee, salary, etc. 
Yet another non-canonical perspective can be defined as 
the class student with endowed attributes department, 
courses taken, etc. There may or may not be an explicit 
class corresponding to the set of all persons. Endowed 
attributes may have the same name as class attributes as 
well as attributes in the type of the class.

Users may create objects of type person and clas-
sify them into different classes. A totally new role of a 
person can be added to the database by simply creating 
a new class for the role. This does not affect the existing 
type or class hierarchy unnecessarily. An object's role can 
be added or removed by simply classifying or declassifying 
that object from the associated class. A person gains an 
additional new role by simply allowing him to be in the 
new class. A person who ceases being a student is simply 
deleted from the student class; his or her other roles are 
unaffected.

The PINOL concepts of canonical and non-
canonical classification of objects are described in the fol-
lowing Subsections, especially features of PINOL classes. 
We describe most of the features through examples. A 
more extensive description of PINOL classification con-
structs is reported in 6.

2.1 Types and Classes

PINOL types correspond to types and classes in C++. 
They form the structural definition of the objects. They 
can be roughly divided into two kinds: basic types and 
object types.

Predefined types available in C++ such as inte-
gers, arrays, records, pointers, etc. are basic types of 
PINOL.

PINOL object types are essentially C++ classes. 
They provide canonical structure of objects; all objects of 
the same type have the same canonical structure.

PINOL types are concerned only with the struc-
ture of objects; collections of objects are stored in PINOL 
classes. However, PINOL classes are more than mere 
collections. They allow context sensitive information to be 
added to an object to form a non-canonical perspective 
of the object. The syntax and semantics for PINOL class 
declaration are described in subsequent Subsubsections 
with the aid of the following examples.

class NUS_employees contains objects of type 
employee

attributes:
  address: add-type; /* main campus address */
  minimum-salary: float;
  maximum-salary: float;
  increment-level: integer;
methods:
  integer set.increment.level(inc:integer);
  { return(increment.level = inc) };
  ....
endowed attributes:
  department: string;
  duties.plan:text;/* teaching duties */
  ....
endowed methods:
  void display()
  { /*display the teaching duties*/ } ;
  ....
initialization: NUS_employees()
  { address = prompt.address();
    minimum.salary = 500;
    maximum.salary = 15000;
    increment.levels = 10; } ;
end class ;

class ISS_employees contains objects of type 
employee
subclass of NUS_employees
attributes:
  department: string; /* the name of the dept. */
  address: address-type; /* dept. address*/
  director: employee*; /* the director name */
methods:
  string set.department(dept:string);
  { return( department = dept )};
  employee* set.director(np:employee*);
  { return( director = np )};
  ....
  void display(); /* print class attributes*/
endowed attributes:
  address: email.add.type;
  security.card.nointeger;
  ....
endowed methods:
  void prompt.address() { <body> } ;
  void prompt.security() { <body> } ;
  email.type set.address(new.add:email.type)
  { return( address = new.add) };
  ....
initialization: ISS_employees()
  { address = prompt.string();
    department = prompt.department();
    director = new employee;
    ISS_employees.add(director); } ;
end class ;

2.1.1 Members

Objects in a class are called members of the class. Mem-
bers of ISS_employees are objects of type employee or 
subtype of employee. We call this the type requirement 
of the class ISS_employees. The type requirement of a 
PINOL class should be thought of as the canonical per-
spective of members of that class. The rest of the PINOL 
class declaration constitutes the non-canonical perspec-
In figure 1 the object 01 is a member of ISS.employees. The canonical perspective of 01, is of type employee, the type requirement for ISS.employees. The non-canonical perspective of 01 in class ISS.employees is denoted by 01.ISS.employees. They refer to the same object, but the non-canonical perspective has extra attributes containing context sensitive information attached to it. It is important to remark here that 01 and 01.ISS.employee have the same object identifiers.

An object's membership of a class extends to all superclasses. Thus members of ISS.employees are also members of NUS.employees:

All subclasses inherit class attributes of their superclasses. They inherit both the attributes and the values in the attribute slots, except when the attribute is redefined in the subclass (as depicted in figure 1). We call this inheritance by value. In the example, NUS.employees contains a class attribute minimum.salary, which is also ISS.employees.minimum.salary, since it was not redefined. Any changes in the value of NUS.employees.minimum.salary is reflected in ISS.employees.minimum.salary and vice versa.

Class methods are methods on the class itself; they have access to class attributes. A class method is unrelated to any method of the type requirement of the class. In the example, the PINOL class ISS.employees has a class method set.department. Any mention of department within set.department refers to the class attribute ISS.employees.department. Also, all references to “self” in set.department refers to the class ISS.employees. The syntax for invoking class method is of the form ISS.employees.set.department(...), read as “invoke set.department in the context of ISS.employees.” Class methods are inherited in a similar way as class attributes.

2.1.3 Endowed attributes and methods

Endowed attributes refer to additional attributes of each member of the class. Although all PINOL classes in the same class-subclass hierarchy contain the same type of objects, we may desire that certain objects should have extra information that is applicable only in certain contexts. Endowed attributes and methods are introduced for this reason.

In our example, in addition to attributes and methods defined in employee, each member of ISS.employees is given new local attributes, e.g. address of type email.add.type. This endowed attribute is unrelated to attributes of type employee, or the class attribute address in ISS.employees. The employee attribute address contains home addresses of employees and this property is present in all objects of type employee. Whereas the endowed attribute address refers only to electronic mail addresses of members of ISS.employees.

The local nature of endowed attributes must be stressed. Each member of ISS.employees has his own endowed attribute address, changes in one does not affect others. The syntax for accessing endowed attributes is “ISS.employees -> ref(o) -> address, read as “access endowed attribute address of object o which is a member of ISS.employees in the context of ISS.employees.”

Endowed attributes of members of a class are inherited by members of its subclasses unless they are redefined by the subclasses. Naming conflicts are resolved using standard C++ rules. If NUS.employees endows
2.2 Endowed Attributes versus Subtypes

A PINOL class is not a type — members of ISS.employees are of type employee, not ISS.employees (for ISS.employees is not a type). In a system with one class per type, this example can be simulated by creating a new subtype for ISS.employees. To motivate the PINOL class concept, we compare modeling of information using endowed attributes versus creating new subtypes.

The National University of Singapore has many departments. Each department has its own classifications of people. For example, the two departments, Institute of Systems Science (ISS) and Department of Information Systems and Computer Science (DISCS), classify their employees differently. ISS classifies its staff into software engineer I, software engineer, research associate and research member. DISCS classifies its staff into tutor, lecturer, senior lecturer and professor. These classifications are independent. If a person holds a joint appointment at these two departments, then his/her job title depends on the context. For example, a person may be a research associate from ISS and a lecturer from DISCS. To model the above situation using PINOL classification concept, we use only one type, employee, which is the type requirement of all the 10 classes. These 10 classes are: ISS.employee having four subclasses and DISCS.employee having another four subclasses. The endowed attributes are used for the extra fields to support context sensitive information under each classification.

One may combine these independent classifications into a monolithic one by converting endowed attributes in subclasses into subtypes and declaring a class for each of the types. However, this may lead to an explosion of types and classes. For example, given the two PINOL classifications ISS.employees and DISCS.employees each having four subclasses, and a given object o such that o is both a member of research.associate and lecturer; to create a single type hierarchy for the above classifications, it is necessary to declare a mixed type: lecturer-research.associate, which is subtype of both lecturer and research.associate types. We have no means of telling a priori that any combination of other types are not required (since the object o could have been a member of research.associate and senior lecturer as well). Thus the converted hierarchy needs at least 4^2 classes. In comparison, the classification with endowed attributes requires 2 * 4 classes. With such a large number of classifications in the former case, it is difficult for anyone to decide where to put a person he is interested in because he has to choose from 16 possible classes compared to looking only at the 4 classes from ISS.employ or DISCS.employ.

Having independent classifications makes life easier. It also has other uses such as for security or access control. For instance, ISS keeps some sensitive data on projects. So besides the normal classification of projects which is exported to other departments, it may have secret classifications of projects which are not shared. Thus, sensitive information on classified projects can be kept there safely.

3 PINOL RULES

In Ullman’s invited talk at DOODS91, he advocated the importance of a declarative language for querying. This is especially true for ad-hoc query such as finding relationship among data in scientific databases. He claimed that it is not possible to take object identity (OID) seriously and still retain the declarativeness enjoyed in a first order logic system. He also believes that declarative languages cannot be object oriented in a non-trivial way. We support his claim that an object oriented language is not declarative enough for querying, but do not agree that OID and declarativeness cannot co-exist. We will show in this section how to add logic (Datalog) to our PINOL system to achieve a certain level of declarativeness yet still retain the benefits of an object oriented system. We emphasize the combination of declarativeness and object orientation can be done by choosing object orientation as the base and add logic on top rather than the other way round. An earlier exposition on PINOL rules was given in 11. Here we provide only a pragmatic view of PINOL rules with emphasis on how it can be used to define complex (recursive) relationships between objects.

3.1 Syntax and Semantics of Rules

An example of a datalog-like rule path has the following syntax:

path(x, y) if given
All variables must be listed in the forall part of the rule. This is to indicate that only objects in the correct category can play the roles as stated in the rule. Here point may be the name of a type or a PINOL class. If point is the name of a type point (of course!), then the corresponding variable is considered to have type points. Otherwise, point is the name of a PINOL class point (whatchevers?) containing objects of type t; then the corresponding variable is considered to have type ts. Each rule is defined once and is stratified. The keyword given is used to stand for extensionally given facts.

Once a rule R is defined, all the objects in the system instantly understand it. To reflect this understanding, these objects acquire a set of foci. The foci of a rule are the various functions which can be evaluated as methods in the underlying object oriented system. There are basically three kinds of focus for every rule: assert, retract, and query. For instance, with respect to the rule path above, the kinds of foci we get are:

\[
\begin{align*}
\text{assert.1.2.path}(z:ts, y:ts) : \text{bool} \\
\text{retract.1.2.path}(z:ts, y:ts) : \text{bool} \\
\text{1.path.2}(z:ts) : \text{set}(ts) \\
\text{2.path.1}(y:ts) : \text{set}(ts) \\
\text{1.2.path}(z:ts, y:ts) : \text{bool} \\
\text{2.1.path}(y:ts, z:ts) : \text{bool} \\
\text{path.1.2}() : \text{set}(ts) \\
\text{path.2.1}() : \text{set}(ts)
\end{align*}
\]

The literal \(\{A\}(!x)\) is a test predicate. \(A\) is a block of PINOL codes that does not invoke any other rule focus. In this example it could be a function checking that \(x\) is male and over the age of 50. A test predicate must evaluate to a boolean value and is required to terminate. The bang(!) before the \(x\) is an annotation to indicate that the test predicate cannot be evaluated until \(x\) is instantiated. A bang(!) after a variable annotates that it get instantiated after the evaluation of the test predicate. The only side effect expected in a test predicate is the instantiation of those variables with a bang after them.

Due to the presence of test predicates and negations, not every rule focus can be computed soundly and completely with respect to classical predicate calculus. Therefore, PINOL uses some simple and conservative syntactic criteria to classify foci into legal and illegal foci. Only legal foci may be used in a PINOL program and PINOL guarantees sound and complete evaluation for them. In general, if a rule is quantified over PINOL classes only, all its foci are legal. This is because a PINOL class is finite in size. On the other hand, if one of the variables is quantified over a PINOL type, some foci of the rule may not be legal.

### 3.2 Inter-object Relationships and Rules

The above description of rules assume that given facts are explicitly stored in tables. Foci are just various smart access routines to this table. In another word, the definition of rules assumes that all the base relationships (given facts) between objects are explicitly specified in somewhere. For example, in order to relate an employee \(e\) to his/her manager \(m\), we create the employee object and the manager object. When the employee \(e\) is working for a particular manager \(m\), we define the relationship by creating the fact, assert.1.2.manager(e, m). But in an object-oriented data model, relationship between two objects is suppose to be modeled as an attribute or as a method in the class. For example, with the employee-manager problem, the fact that an employee is working for a particular manager can be considered as a property of employee and might be stored in the data structure of the employee object. On the other hand, it is also possible to store it as a method in the manager's object.

Modelling relationship as attributes of objects is the philosophy of object-oriented paradigm. Here, encapsulation is enforced and data is held by the object. With this approach, however, there is no global view of relationships. For example the manager object may not know the relationship with the employee object unless the inverse is stored in the manager object. If the inverse is being stored, there is no guarantee that the implementation of the inverse relationship is consistent with the implementation of the original relationship. Another problem is that since the relationship is buried somewhere, inference rule such as the path which has to depend on the global view of...
the relationship between the points cannot be supported. On the other hand, specifying relationships using the assert focus gives a global view of all relationships between objects. This provides a straightforward method for inferencing. But, as we have pointed out earlier, it violates encapsulation.

We want to keep to the philosophy of modeling relationship as in object-oriented paradigm to retain structural object-orientation. But we also wish to be able to specify complex relationships between objects by rules to achieve declarativeness. The key to achieve our goal is to integrate rules with PINOL classes and types in such a way that it is always possible to derive the facts required by the rule through calling the object's methods. In this way encapsulation is respected and inferencing can be supported.

It is found that test predicate is a feature adequate for the specification of methods invocation in the rule. So for every rule which requires the base relationships to be extracted from the class, a test predicate specification is used instead of the keyword given. So despite storing relationships with objects, the inference engine has a way to access it.

The program transformation technique, used to generate the foci for the normal rules is adopted to generate a set of query foci for rules having test predicate specification. We explain the mechanism for deriving the test predicate foci using the following example.

```
object type employee
attributes are:
  workfor: manager;
methods:
  (manager*) get-work-for();
end object type

object type manager
attributes are:
  projects2managed: set(project*);
  employee2managed: set(employee*);
methods:
  set(employee*) get-employee-managed();
end object type

class employee-class contains objects of type employee
class manager-class contains objects of type manager

/* Querying who is the boss of whom */
rule manager(m, e) if
  { m == employee-class -> e -> get-work-for() }!e,!m
forall m: manager-class, e: employee-class;

rule boss(m, e) if
manager(m, e) or boss(m, x) and boss(x, e)
forall e: employee-class, x, m: manager-class;
```

The relationship between the manager and the employee is stored with the respective object. When making a query on who is the boss of whom, it would be more declarative to be able to state the query as boss(x, y)? This query is really a Datalog rule whose evaluation depends on the manager rule. The manager rule is defined with the test predicate to access the relationships from the employee's object by the method get-work-for. To generate the set of foci for the manager's rule, the test predicate specification is first translated into a function. For the above example, we can introduce a function foo more or less by making a verbatim copy of the test predicate:

```c
foo(employee* e, manager* m)
{
  return (m == employee-class -> e -> get-work-for())
}
```

A set of foci (functions) is generated for the manager rule. Each of the foci will be calling the above test predicate function with the correct mode of input and output. For example, the focus _2manager-1(m, e), roughly corresponds to the implementation of the following function,

```c
set(manager*) _2manager-1(employee* e)
{
  return{m:manager* | manager-class.contains(m)
  and foo(e, m)}
}
```

The body of this function is expressed using the relative set abstraction, which is another useful feature of PINOL, similar to ML's list comprehension. The other foci associated with manager rule are: _1manager-2(m), _1manager-2(m, e), _2manager(e, m), manager-1-2(), manager-2-1().

4 EXAMPLES

More examples are given in this Section to illustrate how objects of an employee database of the National University of Singapore (NUS) are created and manipulated. The declaration of the necessary types and classes are assumed.

Creation of objects can be done through the PINOL class operator new or the system new operator in C++. We first show the creation of employee for ISS using the class operator new. This operator is invoked with the name of the class in which the object is to be created. The constructor or initialization procedure for the type requirement of the PINOL class is invoked automatically when the class operator new is called. PINOL programs follow as much as possible the syntax of C++. This example shows how objects in the class ISS.employees are
created. We assume that the class attributes are initialized by the class initialization procedure at compilation time. The object P is initialized by the employee object type constructor.

```cpp
employee* p;
int i;
for (i=0; i<10; i++)
    { p = ISS.employees.new();
      ISS.employees + p -= promptsec.card.no();
      ISS.employees + p + add.duties.plan();
      ISS.employees + p -= prompt.address();
      ISS.employees + p + prompt.projects();
    }
```

Objects can also be created using the system (C++) command new. The object acquires default values set up by the constructor of the object type. The following codes illustrate the creation of an employee which we wish to put it in the class research.associates which is a subclass of ISS.employee.

```cpp
employee* createaresearchassociate();
{
    employee* p;
    p = new employee;
    research.associates.add(p);
    research.associates + p -= prompt.room.no();
    research.associates + p + prompt.sec.card.no();
    research.associates + p + add.duties.plan();
    research.associates + p -= prompt.address();
    research.associates + p + prompt.projects();
    return(p)
}
```

The next example shows how relationship between objects in two different classes are set up.

```
project* pj;
employee* p;
p = createaresearchassociate();
pj = research.projects.new();
research.projects + pj -= set-topic();
/* assigning new project leader for the project */
pj + setprojleader(p);
```

Information can be retrieved either by writing the imperative C++ codes or using the higher level relative set abstraction. For example.

```
/* retrieve projects whose duration is > 3 years */
set(project*) three-years.proj();
{
    return((pj.project* | research.projects.contains(pj) and
            pj.duration > 3 ) )
}
```

Some information is more naturally expressed using rules. We show how some queries from the employee database are expressed using the datalog-like rules.

```cpp
rule same.team(x,y) if common.manager(x,y) and
same.project(x,y).
forall x,y:ISS.employe.
```

```cpp
rule works.on(x,y) if
    { s = ISS.employe + y + get.project(); return
      true)(y,s) and (s-member(x)(ts,t))
    forall x:research.projects, y:ISS.employe,
    s:set(project*).
```

```cpp
rule same.project(x,y) if works.on(z,x) and
    works.on(z,y)
forall x,y:ISS.employe, z:research.projects.
```

```cpp
rule manager(x,y) if { x = ISS.employe + y + get.manager(); return
true}(y,x).forall x:manager, y:ISS.employe.
```

```cpp
/* query whether two ISS employees have the common manager */
rule common.manager(x,y) if manager(x,y) and
    manager(y,z).
forall x,y:ISS.employe, z:manager.
```

5 SUMMARY AND RELATED RESEARCH

In this paper, we have described some features of PINOL as implemented in the prototype. We have highlighted the two unique features of PINOL, its canonical and non-canonical classification constructs and inference capability.

A major advantage of the PINOL class concept is that it provides independent classification of objects without leading to type explosion (2.2) when an object takes on several roles. Furthermore, the behaviour of an object is sensitive to the context in which the object is placed.

Systems such as O2 and Gemstone do not support multiple classifications of an object in a convenient way. In particular, objects cannot dynamically change classes. Iris2 do allow an object to gain or lose types throughout its lifetime. However, objects in Iris do not have context dependent behaviours.

Aspects8 is a proposal that fully supports independent and dynamic classification of objects. In this system, an object can be given many aspects, which corresponds to gaining memberships in PINOL classes. Object behaviour is also context sensitive. In this system, aspects are first class values and are part of the type system. In addition, it inherited from Emerald7 the ability
to have several alternative implementations per type. In contrast, PINOL classes are not part of the type system. Furthermore, each PINOL class has exactly one implementation as it is our belief that objects should behave in the same way in the same context. However, Aspects has potential of being a more flexible system.

Rules provide a way of expressing knowledge declaratively in object oriented systems. The inferential strength of our rule is no more than that of Datalog, hence it is too weak for general programming, but it is suitable for expressing static complex relationships between objects in the object oriented system. The test predicate provides a mean to easily extract inter-objects relationships from an object store. Our approach has the benefit that features of object orientation such as object identity and encapsulation can be retained while still achieve a reasonable level of declarativeness. We have shown that by choosing the object oriented system as the base and add logic on top of it, it is possible to get some logic programming into object oriented system. This has the benefit that OIDS and declarativeness can co-exist which has been pointed by Ullman as being one of the open problems in trying to combine deductive database and object oriented paradigm. We don’t deny that the introduction of test predicate affect the declarativeness of the programming style. But provided it is used in a restricted way, the expressive power that it can bring far outweighs this deficiency.

Our design of PINOL is influenced by our goal to simplify the development of object oriented database applications through a closer integration of databases and object oriented programming language constructs. Thus we have emphasised more on the design of high level language constructs as constrained with E and CO2 which emphasis more on the system level details such as management of persistency and concurrency. A preprocessor for PINOL to C++ which supports PINOL types, class and rules has been implemented.

ACKNOWLEDGEMENT

We thank Yan Ling Ling for her contributions to the section on rules. We also want to thank Ho Lye Kuan and Tan Chee Yong for the implementation of PINOL.

References


12) Zdonik S.B., Maier D., editors. Development and Implementation of An Object Oriented DBMS