AN OVERVIEW OF OOPS+,

AN OBJECT-ORIENTED

DATABASE PROGRAMMING LANGUAGE

Els Laenens - Dirk Vermeir

Philips International B.V.
Corp. ISA / AIT
Building VN3
P.O. BOX 218
5600 MD Eindhoven, The Netherlands

Dept. of Math. and Computer Science
University of Antwerp, U.I.A.
Universiteitsplein 1
B2610 Wilrijk, Belgium

Abstract

This paper provides a brief introduction to the OOPS+ knowledge-representation language. While basically object-oriented, OOPS+ integrates database concepts as well as classical knowledge-representation techniques such as rule-based inference and demons. In addition, the language supports types as first-class objects, inheritance, imperative function definition, and query facilities based on logic programming.

1. Situation and motivation

OOPS+ has been designed within the framework of the Esprit KIWI project[1] . The aim of the project is to design and develop a knowledge-based user-friendly system for the utilization of information bases. The KIWI system consists of four software layers (Figure 1): the user interface (UI), the knowledge handler (KH), the advanced database environment (ADE) and the information base interface (IBI). In order to support information retrieval both directly from the KIWI knowledge base and indirectly, through the IBI, from the existing external databases, the ADE uses the relational model to represent knowledge. At present the relational model is widely accepted to represent data. However, the fact that it is restricted to flat relations has proven to be a problem in many applications. Therefore, the KH is introduced within the general architecture of the KIWI system. Its main function is to provide an environment that supports the definition and also the manipulation of knowledge using a semantically rich formalism.

This research was supported in part by the EEC Esprit program under contract P1117.

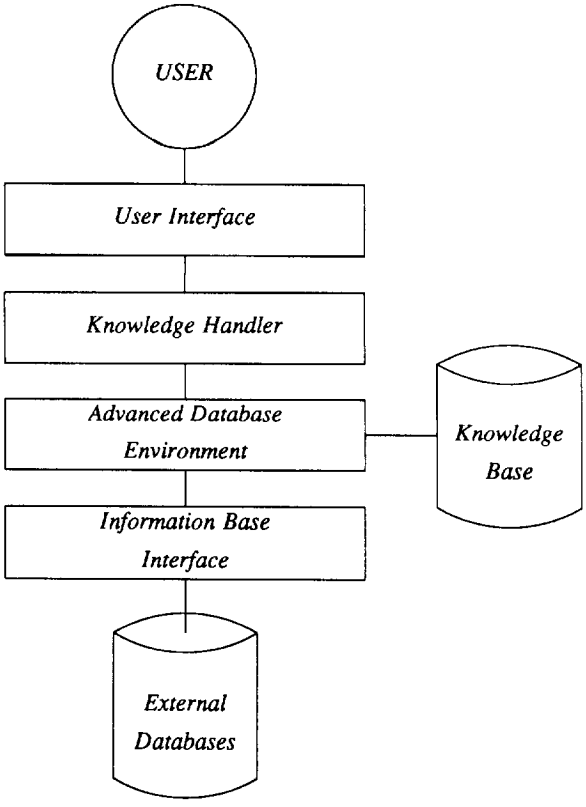

Figure 1. KIWI Architecture

The KH makes use of the facilities of the ADE for the storage and retrieval of knowledge. It is the responsibility of the ADE to perform the appropriate translations between the KH's complex objects and the ADE's underlying 'flat' relational structures. On the other hand, the KH interfaces to the User interface (UI) module which will provide a user-friendly view of the knowledge in the system using state-of-the-art graphical techniques. In this paper, we will discuss some features of the knowledge handler.

2. Requirements

It follows from the above that the KH is characterized mainly by the chosen knowledge-representation formalism, which in turn will be influenced by the intended area of application: intelligent access on information stored in large conventional databases, enriched with a local knowledge base. As a consequence of the functionality of the KH within the KIWI system, we get the following requirements for the knowledge-representation language (OOPS+) that realizes the formalism[2] .

R1 Concerning the *semantic modeling* concepts we expect organizational facilities such as classification, aggregation, generalization and specialization for structuring the knowledge base. OOPS+ will need to manipulate complex objects.

R2 Regarding *database* concepts, we want powerful query facilities as well as easy translation between the complex objects of the KH and the flat relational structures of the ADE.

R3 Another requirement is the provision of tools for *knowledge-based application programming*, which means that we want to be able to apply some knowledge representation technology to general data processing problems.

R4 Finally, we also want to keep *simplicity* by introducing all features in their minimal, essential form (e.g. as was done in the Amber language)[3] .

3. The resulting language: OOPS+

In order to meet all of the requirements, OOPS+ is a multi-paradigm knowledge-representation language[4] .

The query facilities (R2) are based on *logic programming* while knowledge-based application programming (R3) is supported through the facilities of *rule-based* and *access programming*: a *trigger* mechanism is available that can be used to enforce integrity constraints. As far as the knowledge-representation aspect (R1) is concerned, we will look upon the knowledge base as a collection of objects and relations defined over them. Modifications to the knowledge base occur through the insertion and deletion of objects and the manipulation of relations.

In the sequel, we will discuss OOPS+ in two layers.

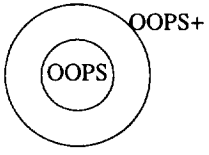

Figure 2.

The innermost layer deals with all aspects concerning knowledge representation. The purpose of this layer is to provide an environment suitable for the definition and manipulation of knowledge. In other words, this layer should meet both the semantic modeling (R1) and the simplicity requirements (R4). As this is the most fundamental part of OOPS+ we will be referring to this layer as the kernel of OOPS+, called OOPS. The other layer provides the different programming paradigms. As shown in Figure 2, this layer is build on top of OOPS as each of the paradigms makes use of the environment established by the kernel. So OOPS+ consists of a kernel (OOPS) enriched with some powerful paradigms in order to satisfy the variety of requirements (i.e. R1-R4).

4. OOPS: the OOPS+ kernel

4.1. Introduction

Different ways of structuring data have generated distinct classes of programming languages and induced different programming styles. Data is said to be taxonomically organized if it constitutes a hierarchy of classes and subclasses, and if data at any level of the hierarchy inherits all the attributes of data higher up in the hierarchy. Programming with taxonomically organized data is often called object-oriented programming, and has been advocated as an effective way of structuring programming environments, databases and large systems in general. In more conventional languages, data is organized as cartesian products, disjoint sums and function spaces i.e. the basic data type constructors in denotational semantics. Our semantic modeling requirement (R1) for knowledge representation suggests to use a taxonomically way of structuring data whereas the conventional way seems more appropriate if we want to introduce as few different concepts as possible to meet the requirement of simplicity (R4). For this reason, both the taxonomically and conventional ways of structuring data are merged in OOPS.

4.2. Objects and values

The notion of **object** is considered to be primitive in OOPS. We use O to denote the set of all objects. Each object has an internal state, called its **value.** An object retains its identity through arbitrary changes in its own state. One can look upon an object as a kind of identifier with which a value, the data represented by the object, is associated. Thus the **state** of an OOPS program is defined by a partial function

$$s : O \rightarrow V$$

where V is the domain of possible object values. Let us write *State* to denote the set of all such functions.

We distinguish primitive values (integers, strings, etc.) and complex values made up of primitive values and constructors[5,6,7] . Complex (primitive) objects are objects having a complex (primitive) value. Because of the existence of complex objects, a single entity can be modeled as a single object: entity features need not be simple data values, but can be entities of arbitrary complexity. On the contrary, in the traditional relational database approach an entity is represented as multiple tuples spread amongst several relations.

In the next sections, we will start off with three kinds of complex values. Records (cartesian products), finite sets and functions will be presented subsequently. Sets are supported explicitly, without the encoding required in the relational model, and can have arbitrary objects as elements, they need not be homogeneous. Furthermore, functions can be used to protect the integrity of the knowledge base.

As will be shown, the subset

$$Prim + Rec + Set + Fun$$

of V already supports some aspects of aggregation, classification, generalization and specialization.

Some considerations concerning the object hierarchy (inheritance) along with the introduction of three additional complex objects (meet, join and power objects) will round off our discussion on OOPS.

Recapitulating, we get the following semantic domain of object values in OOPS:

$$V = Prim + Rec + Set + Fun + Meet + Join + Power$$

4.2.1. Primitive values

Oops supports the following primitive values:

- predefined domains:
  - `Int`,
  - `Str`,
  - `Log`

which represent the sets of all integers, strings and boolean values respectively;

- atomic values:
  - integers (e.g. -1, 0, 1) having type `Int`,
  - strings (e.g. "Fred", "") having type `Str`,
  - the boolean values `True` and `False` having type `Log`,

i.e. all elements of the predefined domains;

- special values:
  - an error value `error`,
  - an undefined value * (or any),
  - a nil value `nil`

With each of the primitive values, there corresponds a constant expression which denotes the unique predefined object in the state that has this value, i.e. we assume the existence of a subset O_{Prim} of O which is isomorphic with $Prim$. Thus

$$O \supset \{o_p \mid p \in Prim\} = O_{Prim}$$

such that

$$\forall\ s \in State, p \in Prim\ \ s(o_p) = p$$

O_{Prim} is the set of primitive objects. Hence, in this case, we can identify the objects with the values. For example, the expression

```
"Fred"
```

will evaluate to the object $o_{"Fred"}$ which has as value the string constant "Fred".

4.2.2. Record values

A record value is a finite labeled set of (references to) objects. Labels are denoted by identifiers. Formally,

$$Rec = (L \rightarrow O)$$

where L is the set of labels which is disjoint from V. Note that they are not identifiers nor strings and can not be the result of the evaluation of an expression.

Record valued objects are constructed using the '( )' operator, e.g. evaluation of

```
(name = "Fred"; birthdate = 1960)
```

will create an object with a record value containing two fields, one labeled *name* pointing to $o_{"Fred"}$, the other labeled *birthdate* referring to o_{1960}.

Using ':=' instead of '=' in a record field definition indicates that the field at hand may be *updated*, otherwise fields retain their creation time value. For example, the object created by the expression

```
(name = "Fred"; age := 3)
```

has a constant *name* field while the object pointed to by *age* can be updated.

OOPS supports *aggregation* through records. A collection of concepts can be treated as a single concept having several components. For example, an address could be thought of as an aggregation of its street, city and country.

```
(
name = "Fred";
address = (
          street = "Fifth avenue";
          city = "New York";
          country = "USA"
          )
)
```

A record object may be declared *extendible* by preceding it with a ':'. One can add/delete label-value associations to/from an extendible record value. E.g.

```
(
name = "Fred";
address = : (street=...)
)
```

creates a record object having two fields. The *address* field points to an updatable record object r. However, the association between the label *address* and r is fixed. Thus, extendability is a feature of the record and not of the reference to it.

Records also play the role of record types:

$$(name = "Fred";\ age = 25) : (name = Str;\ age = Int)$$

The latter record is a record type for the former record because "Fred" and 25 are of type Str and Int respectively.

4.2.3. Set values

A set value is a finite set of (references to) objects

$$Set = Fin(O)$$

Set valued objects are constructed using the '{ }' operator. E.g. evaluation of

```
{"Jane", 64, FALSE}
```

will create an object with a set value

$$\{o_{"Jane"}, o_{64}, o_{FALSE}\}$$

Note that different objects may have identical values i.e.

```
{(i=3),(i=3)}
```

will create an object with a set value containing two objects. This illustrates one of the principles of object-oriented programming: objects - in contrast to tuples in a relation - are not identified by their intrinsic properties. Thus two objects can be distinct even if all their defined components have the same values; objects are distinguished by reference.

Sets will be the basic constructs for providing OOPS with the *classification abstraction* method.

A set object may be declared *updatable* by preceding it with a ':'. E.g.

```
(
name = "Fred";
kids = : {(name="Pebbles";age:=1)}
)
```

The set valued object s which *kids* points to is updatable. However, the association between *kids* and s is fixed.

4.2.4. Functions

A function is an object that, when activated, takes a number of parameters and returns an object. Functions may have side effects, so

$$Fun = State * O \rightarrow State * O$$

A function definition consists of a parameter environment object (used as a record type), a function body and an optional type for the return value. The parameter environment object is a record type for the parameter object the function may be called with.

E.g.

```
(x=Int;y=Int)
    {
    if (x>y)
            return(x);
    else
            return(y);
    }
Int
```

Here, valid function calls have a parameter object containing two integer valued parameters x and y.

4.3. Inheritance

As is the case in Amber[3] , the OOPS type system exploits inheritance, introduced in Taxis and Galileo, and extends this to work on higher-order types. Unlike Amber, where inheritance is based on the notion of *type inclusion*, our basic notion is the *instance-of* relation between objects. This relation (denoted ':') captures both the notions of *having a type* and *explicit set membership*. In other words, it will support OOPS in both *type checking* and *referential integrity*. For example

$$1 : Int$$

since each integer valued object is by definition of type (the object) Int. Also

$$1 : \{1,"Fred"\}$$

because the instances of a set valued object are exactly its members.

Let us define the instance-of relation for primitive objects and set objects in a more formal way. In a given state s, an object, say o_i with value v_i=s(o_i) is an instance of an object o_m with value v_m = s(o_m) if neither v_i nor v_m is the error value and if one of the following conditions hold.

- v_m is a predefined domain and v_i is one of its elements. In other words, v_m is Int, Str or Log and v_i is an integer, a string or a boolean value respectively.
- v_m is the undefined value *.
- v_i is the nil value.
- v_m is a set and v_i is one of its members.

v_m is also called a meta object of v_i.

As mentioned before, a record is an instance of another record if the instance-of relation holds for equally labeled fields of these records. For example

$$(name="Fred";sex="Male") : (name=Str;sex=\{"Male","Female"\})$$

because "Fred" and "Male" are instances of Str and {"Male","Female"} respectively.

Let us now consider the following definitions.

```
person = (name=Str; age=Int);
student = (name=Str; age=Int; number=Int);
fred = : (name="Fred"; age:=19)
```

It follows from the above that fred is an instance of person. However it is not an instance of student because of the missing number field. Suppose that we wish to extend fred so that it becomes a student:

```
fred = : (name="Fred"; age:=19; number=878)
```

We have added some information to fred to create a more informative object. Intuitively, we still want fred to be an instance of person.

In general, this gives rise to the definition of the instance-of relation between record objects. Using the same notation as above, the additional condition is defined recursively

- v_i and v_m are record values $(l_1=o_1,..,l_n=o_n,l_{n+1}=o_{n+1},..,l_k=o_k)$ and $(l_1=O_1,..,l_n=O_n)$ respectively such that equally labeled fields satisfy the instance-of relation, i.e. $o_1:O_1$ and .. and $o_n:O_n$.

The fields of person match (both in label and value) fields that are present in student. From this it is inferred that any instance of student is also an instance of person. Therefore, student is called a *subobject* (or subtype) of person.

This leads to our definition of the *subobject relation* in OOPS. We say that an object o_s is a *subobject* of an object o_m (denoted $o_s<o_m$) if any instance of o_s is also an instance of o_m. Hence this relation is both reflexive and transitive.

Note that the instance-of relationship (and consequently the subobject relationship) is inferred from the intrinsic properties (i.e. the structure) of the objects and need not be declared explicitly. As an effect of this interpretation, OOPS is a *polymorphic language* which means that an object may have more than one type.

Now consider the function age

```
age = (p=person)
  {
  return(p.age);
  } Int
```

It takes a person as argument and returns an integer valued object. We say that this function is of type (i.e. an instance of) (p=person) → Int. As fred is an instance of person, we can compute age(p=fred) which will result in an integer (i.e. an instance of Int).

So far we can write in general, if f is a function of type $\alpha \rightarrow \beta$ then for each instance, say i of α (i.e. i:α) f(i) is meaningful and of type β.

Since any object that is an instance of student is also an instance of person, age is also of type (or an instance of) (p=student) → Int.

Moreover, if we think of the parameter environment object as a whole (i.e. the record object (p=person)) we see that (p=person,s=Str) → Int is another type of age because any instance of (p=person,s=Str) is also an instance of (p=person) and the latter is exactly the domain on which the function age operates.

From this we can infer the following property. When f is a function of type $\alpha \rightarrow \beta$ then it is also of type $\alpha' \rightarrow \beta$ for all the subobjects α' of α. Note that this requires α and α' to be record objects as they are supposed to be parameter environments. This is an example of what is called *horizontal polymorphism*, i.e. polymorphism based on inheritance.

On the other hand, suppose that the following function picks the best out of a number of students.

```
bestStudent = (class=studentSet)
  {
  ..
  } student
```

Apparently, this function is of type (class=studentSet) → student. As any instance of student is necessarily an instance of person, bestStudent is also of type (class=studentSet) → person.

Thus a next property appears. When a function is of type $\alpha \rightarrow \beta$ then it is also of type $\alpha \rightarrow \beta'$ for all the *superobjects* β' of β (i.e. $\beta<\beta'$).

As a result, we can meaningfully compute

```
age(p=bestStudent(class=s))
```

where s is a set of students.

Recapitulating we get, if f:$\alpha \rightarrow \beta$ and $\alpha' < \alpha$ and $\beta < \beta'$ then f:$\alpha' \rightarrow \beta'$.

4.3.1. Power objects

Up to now we have types for primitive objects and for record objects as well as for function objects. How about set objects?

We would like to specify that the set object {1,2,3} is a set of some integer valued objects. One way of doing so is through the subobject relation defined in a previous section.

$$\{1,2,3\} < Int$$

However, we do not feel like introducing a direct representation of this subobject relation as it is based on the more primitive instance-of relation which is already explicitly available in OOPS. To this end, a new form of objects is introduced, called power objects. A power object is a complex object constructed by using the power operator, denoted '[ ]'. Now we can write

$$\{1,2,3\} : [Int]$$

Similarly, a possible definition of the set of students studentSet (used in an earlier example) is

```
studentSet = [student]
```

because this requires that each instance of studentSet is a set of instances of student.

In general we can write: the power operation takes an object, say x and constructs a new object [x] which has the property that

$$y : [x] \ \ iff \ \ y < x$$

Hence the power object of x has all x's subobjects as instances.

4.3.2. Meet

As OOPS supports the notion of multiple inheritance, one may require to specify that an object is an instance of several other objects simultaneously. Therefore, we introduce another constructor, namely *meet* (or intersection), denoted '&'.

For example, reconsider the example where fred was a person with an additional integer valued field labeled number. Hence

$$fred : person \ \& \ (number{=}Int)$$

Note that the earlier definition of person and student are equivalent to the following.

```
person = (name=Str; age=Int);
student = person & (number=Int);
```

The & construct arranges two declarations to be simultaneously effective, and consequently allows the additional fields to be introduced simultaneously with those 'inherited' from the 'superobject'. This is equivalent to *specialization*.

However, the meet operator also operates on other objects. A more complicated example is the following. Let men be the set of all male persons, i.e. {fred,bill,..}. Then

$$student \ \& \ men$$

which is the meet between a record object (student) and a set object (men) will have all male students as instances. In other words, its instances are exactly the members of men that are of type student.

Generally we have: the meet operation takes 2 objects, say x and y and constructs a new object x&y which has the property that

$$z : x\&y \ \ iff \ \ z : x \text{ and } z : y$$

4.3.3. Join

It is often desirable to specify that an object is an instance of at least one of a number of other objects. To this aim, we introduce a new object constructor called *join* (or union), denoted 'I'.

For example, fred is an instance of the join between student and employee (whatever this may be) since fred is an instance of student.

$$fred : student \ | \ employee$$

In general: the join operation takes 2 objects, say x and y and constructs a new object xIy which has the property that

$$z : x \,|\, y \ \ iff \ \ z : x \ or \ z : y$$

4.3.4. Extents

We have discussed the instance-of relation with respect to programming languages. E.g. person denotes the set of all possible objects that are instances of (i.e. of type) person. However this set is not actually available to the programmer as an object. Let us now have a look at the instance-of relation from a different angle. In database programming, person not only describes a type (a set of possible persons) but it is also used to describe the set of all persons that are currently in the database. This set is called the *extent* associated with the 'type' person.

Similarly, in OOPS we can think of an object as its extent being the set of all its instances (excluding nil) that are currently present in the knowledge base. Note that this means that OOPS merges the notions of 'type' and 'class'. We distinguish implicit and explicit extents. Explicit extents or user defined extents are set objects constructed using '{ }', whereas implicit extents are system created extents (i.e. through a scan of the knowledge base).

It is worth noting that OOPS is a persistent language which means that any value may persist. In other words persistence is not determined by type. Rather, any object that can be referenced directly or indirectly from the program object is persistent. Objects which cannot be refered are automatically deleted.

Some consequences of the definition of extent are:

1. The extent of a set valued object is the object itself;
2. The extent of a meet object is the intersection of the extents of its component objects;
3. The extent of a join object is the union of the extents of its component objects;
4. The extent of an atomic object is the empty set;
5. The extent of * (i.e. any) is the set of all objects in the knowledge base.

Some examples of the use of extents will be shown later on.

4.3.5. Typed environments

As OOPS contains a number of object constructors, it is mostly desirable that it is strongly typed.

A record object can be thought of as an environment in which both constant and variable (field) declarations are allowed. Hence, to our feeling, it is obvious to allow an instance-of specification for each field, as this makes it possible to specify the type of a constant-field value or to restrict the range of possible variable-field values. In order to get such a *typed environment* we need to extend the definition of a record value. For example,

```
(name="Fred"; Int age:=19)
```

defines a record object of which the age field is updatable, but restricted to refer to an instance of Int

(i.e. an integer valued object). Note that types in OOPS are 'first class' objects[8] .

It is also possible to specify *referential integrity* rules in this way. E.g.

```
person = : {fred,jane,bill};
anEmployee = (
             employee = jane;
             Int salary := 1000;
             person boss := fred
             )
```

At any time, the boss field of anEmployee should refer to an instance of the set person i.e. to an existing person. Hence an error may occur upon deletion of an element of this set.

5. OOPS+

As mentioned before, besides the OOPS layer of OOPS+ - which is mainly responsible for the definition of knowledge - there is a second layer that should provide the programmer with a number of tools for querying the various available databases and in general for knowledge manipulation and application programming. We will now introduce these tools in a rather informal way.

5.1. Logic programming

OOPS+ uses predicates to incorporate the logic programming paradigm. In most object-oriented languages, it is rather difficult and tedious to reference objects by contents: the programmer has to "hand code" support for such references explicitly by providing methods. In OOPS+, predicates can be used to query the object space. A predicate object consists of a set of defining clauses (in the logic programming sense).

A predicate can be activated in two ways: as a logical function that verifies that the predicate holds for the parameter object or as a set containing all tuples satisfying this predicate. It is then possible to further qualify this set using a *where* condition (see below).

$$Pred = State * O \rightarrow State * O$$

The following defines a predicate called parent. Let person be a set of persons.

```
* parent =
  {
  ? parent(child=_c, parent=_p) :-
    person(father=_p, self=_c)
    ;
    person(mother=_p, self=_c)
  };
```

E.g.

```
parent(child=fred)
```

will return a set object containing all record objects that satisfy the parent predicate. Such record objects are equipped with properties as specified in the predicate parameters, i.e. child and parent.

```
parent(child=fred,parent=jane)
```

will return true if jane is a parent of fred.

Predicates can refer to both objects with instances (like person in the previous example) and other predicates. The following predicate refers to the earlier defined parent predicate.

```
* ancestor =
  {
  ? ancestor(older=_x,younger=_y) :-
      parent(child=_y,parent=_x)
      ;
      parent(child=_y,parent=_z)
      ancestor(older=_x,younger=_z)
  }
```

An object may also be queried using *set restriction*, i.e. the *where* condition. For example

```
parent() where (child.age<1)
```

will only select those record objects that satisfy the parent predicate and of which the child field refers to a baby. The object to be queried by the where condition need not be a set object since its extent is used in the selection process. E.g.

```
student where (age=fred.age)
```

will make a set object consisting of all existing students which are of fred's age.

Using both predicates and set restrictions makes it possible to easily formulate rather complex queries.

5.2. Access programming

A powerful programming paradigm is 'access-oriented programming' where actions are triggered by the occurrence of events. Originally conceived as part of AI and knowledge-representation languages[9], similar mechanisms have recently been proposed also for database programming[10] where they can be used e.g. to support constraint checking. OOPS+ supports two forms of access-oriented programming namely triggers and laws. Triggers are demons that get activated whenever a particular event occurs. Laws are similar except that their execution may be postponed which, together with a transaction mechanism, makes them suitable to implement integrity rules.

Like functions, triggers take a parameter environment upon activation and may have side effects. However, triggers can only be activated indirectly e.g. through the occurrence of an assignment.

$$Trig = State \times O \rightarrow State$$

A trigger consists of a specification of the events that will cause its activation and a body describing what actions are to be executed upon activation.

The next example defines a trigger called updatePosition.

```
movingObject = (position = point);
updatePosition = trigger on position
        (x = movingObject)
        {
        updateDisplay(x);
        }
```

The function updateDisplay is called whenever the position field of an instance of movingObject is updated.

Laws are used to provide support for consistency rules.

```
CheckSalaryChange = law on salary
        (e = employee)
        {
        check (e.salary > e..salary)
        }
```

The main difference between laws and triggers is that the activation of a law can be postponed using a transaction mechanism. Also, the activation of a law results in either the commitment or the rejection of a transaction, depending on the value of the boolean expression to be checked. Note that access to the previous field binding in a record is possible through the '..' operator.

5.3. Rule-based programming

Rule-based programming is convenient for describing flexible responses to a wide range of events characterized by the structure of the data. This helps to explain the wide popularity of this paradigm, e.g. to construct expert systems. In OOPS+, rule-based programming is supported through functions: the function body can be either a statement, supporting procedural programming, or a ruleset, supporting rule-based programming.

A ruleset consists of a set of production rules, each specifying a condition-action pair. A condition is a logical expression, while an action is a statement.

```
thermostat = (r = room)
        {
        rule tooHot:
           (r.temp > MAXTEMP) ->
                lowerTherm(r);
        rule tooCold:
           (r.temp < MINTEMP) ->
                higherTherm(r);
        }
```

The call thermostat(r=A212) causes rules to be fired as long as the temperature is not comfortable in room A212.

6. A test case

6.1. The problem

In this subsection we will introduce four tasks - that Atkinson and Buneman[11] believe are characteristic of database programming - together with their solutions in OOPS+.

The example they use is an illustrative fragment of a manufactering company's parts database. The database represents among other things the inventory of a manufacturing company. In particular, it presents the way certain parts are manufactured out of other parts: the subparts that are involved in the manufacture of a part, the cost of manufacturing a part from its subparts, the mass increment or decrement that occurs when the subparts are assembled. A manufactured part may themselves be subpart in a further manufacturing process. The relationship between parts is therefore hierarchical, but it is a directed acyclic graph rather than a tree, for part D may be used in the manufacture of parts B and C, which are both used in the manufacture of part A. In addition, certain information must be held on the parts themselves: their name and, if they are imported, (i.e. manufactured externally) the supplier and purchase cost.

The first task is

> Task 1: Describe the database.

The next task is simple:

> Task 2: Print names, cost and mass of all imported parts that cost more than $100.

Task 3 is somewhat more complicated and defeats many query languages:

> Task 3: Print the total mass and total cost of a composite part.

The last task requires an update by adding some information to the database in order to examine where in the program or type system integrity constraints are implemented.

> Task 4: Record a new manufacturing step in the database, i.e. how a new composite part is manufactured from subparts.

In their paper [11] Atkinson and Buneman show different approaches to meet these tasks. Pascal as well as SQL mainly failed because of inadequate persistence and lack of computational power respectively. They also discuss several other languages and in particular database programming languages with respect to these tasks.

6.2. The OOPS+ solution

In this section, the OOPS+ solutions to the tasks are presented together with some remarks.

Task 1

Figure 3 shows the declarations corresponding to task 1.

```
dollars = Int;
grams = Int;
partType = (name=Str);
basePartType = partType &
               (
               cost=dollars;
               mass=grams;
               suppliedBy=[suppliers]
               );
compositePartType = partType &
                    (
                    assemblyCost=dollars;
                    massIncrement=grams;
                    components=[useType]
                    );
useType = (
          subPart=parts;
          quantity=Int
          );

[basePartType] baseParts =:{};
[compositePartType] compositeParts =:{};
parts = baseParts | compositeParts;
```

Figure 3: Task 1 in OOPS+

We define among other objects, four objects that will be used for typing: partType, basePartType, compositePartType and useType.

PartType isolates the common features of both kinds of parts.

This partType is then used in the definition of basePartType and compositePartType which are two meet objects. It follows from the definition of meet objects that each instance of basePartType must be an instance of partType and of the record (cost=dollars;mass=grams;..). Hence basePartType is a subobject or a specialization of partType.

Similarly, compositePartType is a subobject of partType.

As a basepart may be supplied by several suppliers we define the suppliedBy field in basePartType as a power object: suppliedBy=[supplier]. This means that the suppliedBy field of any instance of basePartType must be a set of (instances of) suppliers (note that we do not examine suppliers in order to concentrate on the rest).

BaseParts and compositeParts are sets that will contain instances of basePartType and of compositePartType respectively. This is specified by the power objects. Both sets are declared updatable (:{}) and initialized to the empty set.

Note the difference between basePartType and baseParts. basePartType is used as a type in the classical sense whereas baseParts plays the role of a class as in other object-oriented languages. In other words, basePartType is an intentional type whereas baseParts is an extensional type.

Parts is defined as the join between baseParts and compositeParts. Thus, at each moment an object is an instance of parts if it is an instance of either baseParts or compositeParts. In other words, each base and composite part is also a part.

UseType represents the use of a part in a composite part. Writing (subPart=parts;..) instead of (subPart=partType;..) requires that a compositePart is made of existing parts.

It follows from the definition of useType that the components field of compositePart also keeps track of the quantity used of a subPart.

Task 2

Figure 4 shows an OOPS+ solution to task 2.

The let statement allows the declaration of local variables. E.g. 'let useType use' declares use as a local variable of type useType.

(parts where cost>100 and suppliers!={}) selects among the instances of parts those objects that cost more than $100 and have a nonempty set of suppliers (i.e. they are imported). The expression returns a set containing these selected objects. Then the print statement is executed for each element in this resulting set.

```
expensiveParts =
        {
        let parts part;
        foreach part in (parts where cost>100 and suppliedBy!={} ) do
                print(part.name,part.cost,part.mass);
        }();
```

Figure 4: Task 2 in OOPS+

Task 3

Figure 5 presents an OOPS+ solution to task 3.

```
totalMassAndCost =
        (p=part)
        {
        if (p:basePart)
                return((totalMass=p.mass;totalCost=p.cost));
        else
                {
                let useType use;
                let grams resultMass:=p.massIncrement;
                let dollars resultCost:=p.assemblyCost;
                foreach use in p.components do
                        {
                        let tmpMandC := totalMassAndCost(p=use.subPart);
                        resultMass += tmpMandC.totalMass*use.quantity;
                        resultCost += tmpMandC.totalCost*use.quantity;
                        }
                return((totalMass=resultMass;totalCost=resultCost));
                }
        } (totalMass=grams;totalCost=dollars);
```

Figure 5: Task 3 in OOPS+

(p=part) is the parameter environment object or the formal parameter object of which the actual parameter must be an instance.

The ':' denotes the instance-of operator. Thus the condition p:basePart is true if p is an instance of basePart and false otherwise.

Note that local variables may be initialized: 'let Int resultMass := p.massIncrement'.

return((totalMass=..;..)) creates a record object which is an instance of the formal return object (totalMass=grams;totalCost=dollars).

Task4

Figure 6 sketches an OOPS+ solution to task 4.

```
newCompositePart =
        (name=Str;assemblyCost=dollars;
                massIncrement=grams;components=[useType]);
        {
        compositeParts += param;
        }
```

Figure 6: Task 4 in OOPS+

`Param` is a reference to the actual parameter object.

Note that the definition of useType along with the parameter type checking (i.e the actual parameter object must be an instance of the formal parameter object) forces the components field to refer to existing objects.

'+=' denotes a set update operator: `'compositeParts += param'` inserts the actual parameter object (`param`) in the set object compositeParts.

The constraint that the relationship between parts is acyclic i.e. no part can be directly or indirectly a subpart of itself is also expressed by this definition of the function object newCompositePart. `Param` always refers to the actual parameter object which is newly created at the moment of the function call. Thus param cannot refer to a component of another object.

7. Conclusion

OOPS+ is object-based and multi-paradigm, supporting procedural as well as logic, rule-based and access (demon) programming. In addition, it unifies the notions of type (types are first-class objects) and collection in a general instance-of relationship. Also, the concept of name space (environment) and tuple (record) object are confused (i.e. unified), both being a set of typed and labeled references to other objects. Persistence is defined by preserving only those objects that can be referred to, using either names or (operations on) collections from a set of designated initial objects, making an explicit

delete operation unnecessary and motivating an approach in which a query results in the creation of new objects which are automatically discarded unless action is taken by storing a reference to the collection in some referable object.

8. Acknowledgement

Thanks to Prof. S. Ceri for his constructive criticism which led to the writing of this paper.

References

1. D. Sacca, D. Vermeir, A. D'Atri, J. Snijders, G. Pedersen, and N. Spyratos, "Description of the overall architecture of the KIWI system," in *Proceedings of the Esprit Technical Week*, Elsevier Publ. Co., 1985.

2. D. Vermeir and E. Laenens, *Requirements document of the knowledge handler (main features)*, 1986. B3 report, Esprit project P1117 - KIWI

3. L. Cardelli, "Amber," in *Proceedings of the Treizieme Ecole de Printemps d'Informatique Theorique*, 1985.

4. D. Vermeir and E. Laenens, *Formal description of the OOPS language*, 1987. B2 report, Esprit project P1117 - KIWI

5. F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," in *Proceedings of the fifth ACM Symposium on Principles of Database Systems*, 1986.

6. L. Cardelli, "A Semantics of Multiple Inheritance," in *Lecture Notes in Computer Science*, vol. 173, pp. 51-67, Springer, 1984.

7. D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," in *OOPSLA'86 conference proceedings*, pp. 472-482, 1986.

8. K. J. Lang and B. A. Pearlmutter, "Oaklisp: an Object-Oriented Scheme with First Class Types," in *Proceedings of the OOPSLA'86 conference*, 1986.

9. D. G. Bobrow and Stefik, *The LOOPS Manual*, Tech. Report Xerox Park, 1981.

10. M. Stonebraker and L. A. Rowe, "The design of POSTGRES," in *Proceedings of the ACM Sigmod International Conference on Management of Data*, ed. C. Zaniolo, 1986.

11. M. P. Atkinson and O. P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys* , to be published in 1988.