Adding Parallel and Persistent Sets to Modula-3

Laszlo Böszörmenyi and Karl-Heinz Eder
Institut für Informatik, Universität Klagenfurt, A-9020 Klagenfurt, Austria
e-mail: {laszlo, charly} @ ifi.uni-klu.ac.at
appeared in: Joint Modular Languages Conference JMLC´94, Ulm, Germany, 28-30 Sept ´94 (revised version: 17. Oktober 1994)

Abstract

Parallel and persistent object sets are suggested to be incorporated into general-purpose programming languages. Two alternative implementations are presented. The actual form of the proposal is an extension of Modula-3.

Keyword Codes: D 3.0; D 3.3; H 2.3
Keywords: Programming Languages, General; Language Constructs and Features; Database Management, Languages;

1. Motivation

The development of the (imperative) programming languages has always been characterized by two "concurrent" main streams: by the development of data structures and that of control structures. Control structures had for a long time always one step advantage over data structures. In concepts such as modules, abstract data types and especially objects, these two streams seem to meet each other [Möss].

On the level of entire programs, however, there is still a deficit on the data side. Programs are generally considered to be a set of modules (linked together into a program). In the Oberon system [WiGu] we do not have programs any more, instead commands, actually a chain of procedure calls. If a program (or the necessary modules of a command) is (are) loaded, then the algorithms are entirely available in the main memory. The structure, the scheme of the data is available as well, however we have no data (apart from some possible very simple initializations). Data come into existence during program execution. In other words, programs are started with an empty or very simple context.

This deficiency is caused by the traditional implicit assumption that people write programs, load them into more or less empty computers and let their calculations made. This is, however, very often not the whole truth. Rather, people have some data, often even a huge amount of data, and they want to do their calculations *in the context* of these data. Traditional general-purpose programming languages still ignore this case. The solution is to use some files if you have only few data, or to use a database system if you have more of them. In other words: *persistence* is still considered to be an exotic special case that cannot be and should not be considered by general-purpose programming languages.

One of the hopes connected to the object oriented technology was to eliminate the "impedance mismatch" between general-purpose programming languages and database languages. The actual situation is that we have powerful, efficient, "light-weight" general-purpose programming languages (such as Oberon-2 [ReWi], Modula-3 [Nels], C++ [Stro] etc.) and highly specialized, rather "heavy-weight" database manipulation languages (such as O_2Query[Deux], GOM[Kemp], Cool[Laas] etc.). We have further languages for database queries (such as SQL [SQL2]). There are a lot of combinations as well (e.g. a number of embedded SQL systems). Concepts that integrate persistence into light-weight

languages in a smooth manner are still needed.

A very similar situation can be noticed considering parallelism. Parallelism is also still considered to be a bit exotic. Almost everybody works on a network of workstations, more and more such workstations are equipped with several processors, and more and more massively parallel computers are installed. The general-purpose programming languages, however, still do not pay too much attention to this question, parallelism remains out of the language. There are a number of parallel extensions of general-purpose programming languages (such as High Performance Fortran or Vienna Fortran [Zima] , Modula-2* and Modula-3* [Phil] etc.). These languages concentrate on scientific computations, especially on computations on large arrays. Persistence is not considered.

We state that sets as "first-class citizens" are able to cope with persistence and parallelism.

2. Related work

There are a number of languages that support sets in an advanced form. In SETL [Schw] e.g. sets are at the heart of the language, however, persistence and parallelism are not considered. Starset [Gilu] is based on parallel and persistent sets and classes. It has restrictions and deficiencies in more traditional data and control structures. GOM [Kemp] provides persistent sets and all the Pascal-like features, and still some more. GOM is fairly "heavy-weight". Parallelism is not considered in GOM. Modula-R, DBPL [Matt] and O_2C [Deux] stand probably the closest to the proposal of this paper. Modula-R and DBPL extend Modula-2 with the concept of a relation, which is nearly the same as a set. They are not object oriented. O_2C is object oriented and offers tuples and lists besides sets. Persistence is provided in form of names as persistence roots while parallelism is not considered.

3. Basic Concepts

The concept of persistent and parallel sets incorporated into general-purpose programming languages is suggested. Implementations of relational databases are based on the concept of set of tuples. Object oriented databases are based similarly on set of objects, or on a hierarchy of such sets (such as classes and views [Laas]). To build a safe object oriented database, the concept of *typed set*s is required.

Persistence is orthogonal to the type of the data. Nevertheless, persistence becomes interesting only in cases, where we have a large amount of data. Therefore, we are not so much interested on single persistent variables, rather on persistent aggregates. Especially on *persistent sets*.

Sets are unordered collections of some objects. Therefore, they can be processed in many cases *in parallel* in a natural way. The most typical operation is to select a subset whose members satisfy a given constraint. As sets are unordered, no assumption can be made on the order of the constraint evaluation. As a consequence, they can be evaluated in parallel.

A well-defined hierarchy of sets introduces an explicit clustering, which makes efficient parallel processing still easier (see the definition of classes in section 3.2 and [Day]).

3.1 Definition of sets

We define sets (based on the definitions of Georg *Cantor* [Cant], [Deux] and of [Elma]) as a

limited, unordered collection of different, type compatible elements. All elements must be different and they must exist. A set is defined over a base type (sometimes simply called the type of the set). All elements of the set must be a subtype of this base type. The subtype relation is reflexive (and transitive), therefore this definition includes the case where elements are of the same type as the base type. Sets corresponding to this definition can be called *typed* and *polymorphic*. We are first of all interested in *sets of objects*.

3.2 Definition of a class

A class is an instance of a set of objects. It is related to other classes in a subclass or superclass relation[1]. A class may have several super- and subclasses (this does not require multiple inheritance in the type system). Since classes are defined as sets, all operations on sets are also available on classes. Classes are assignable to sets or views, and vice versa.

Is *ClassSub* a subclass of *ClassSuper*, we have the following relations:

1. The base type of *ClassSub* must be a subtype of the base type of *ClassSuper*.
2. The extension of *ClassSub* is a subset of that of *ClassSuper*. In other words, all elements of *ClassSub* are also contained in *ClassSuper*. This implies the so-called *instance inheritance*. If we insert a new element into a subclass, it becomes an element of all its superclasses. If we delete an element of a class, it is also deleted from all its subclasses. Let us suppose we have a class of students which is a subclass of persons. If we insert a new student into the class of students, then the student becomes also an element of the person class (we say, a student *is* a person). If we delete a person that is a student then it must disappear from the student class as well.

3.3 Definition of a view

A view is a set that is defined over some sets or classes or views, by means of a generator (e.g. a generator predicate). Views must be always in accordance with their generator. Therefore, they are usually not materialized, rather they are regenerated on every reference. Since views are defined as sets, all operations on sets are also available on views. Views are assignable to sets or classes, and vice versa. Views are updateable. An update of a view causes an update of the sets or classes over which it is defined. No invisible elements can be inserted into resp. deleted from a view.

4. Adding sets to Modula-3

Standard Modula-3 sets are defined as "a collection of values taken from some ordinal type" [Nels]. The following definitions relax some of the restrictions of this definition. Nevertheless, all standard Modula-3 set facilities remain unchanged.

4.1 Declaration of sets

We do not need a new type constructor for the generalized sets, the standard Modula-3 set

---

[1]Note that this definition of a class is quite different from the use of this notion by the programming language community, where a class is generally the same as an object type. The above definiton defines a class as an object container, rather than a type. This usage is very usual in the database community [Laas].

constructor can be used. The syntax of a set type remains unchanged, its form is:

SetType = SET OF Type

Type (which stands for the base type of the set) can be an ordinal type or any reference type. In the following discussion we consider only the most interesting case: where Type is an object type. Sets of non-ordinal types can only be instanced dynamically, via a reference to a set. This is analogous to dynamic (open) arrays. Sets can be instanced by the standard function *NEW*. New requires as a first parameter the type which is to be instanced (this is a standard Modula-3 feature). In the case of sets, an optional size parameter can be given, which serves as a hint for the compiler for the allocation of the set. The compiler may (and should) use this parameter for efficient allocation. The corresponding NEW-signature is:

NEW(Type [, Size])

Let us consider the following declarations (*DP, DS* and *DE* stand for display procedures):

```
TYPE
    Person = OBJECT name: TEXT METHODS display():= DP END;
    Student = Person OBJECT matriculation: CARDINAL OVERRIDES display:= DS END;
    Employee = Person OBJECT salary: CARDINAL OVERRIDES display:= DE END;
    PersonSet  = REF SET OF Person;  (*May contain Persons*)
    StudentSet  = REF SET OF Student;  (*May contain Persons and Students*)
    EmployeeSet  = REF SET OF Employee;  (*May contain Persons and Employee*)
```

Student and *Employee* are subtypes of *Person*. Both of them overrides the display method with an appropriate procedure. The sets may contain only elements which are compatible with their base type. For instancing sets consider the following example:

```
VAR
    persons  := NEW(PersonSet, pSize);  (*pSize: expected number of  persons*)
    students  := NEW(StudentSet, sSize);  (*sSize: expected number of  students*)
    employee  := NEW(EmployeeSet, eSize);  (*eSize: expected number of  employee*)
```

4.2 Operations on sets

4.2.1 Set expressions

Set expressions take compatible sets as arguments and result in a set. The type of the result depends on the operation. *Union*, *Intersection* and *Difference* are defined as usual. An additional set expression is *selection*. Let be *set1* a set of type *T1* and *set2* a set of *T2*, and *RT* the base type of the result. The general form of set expressions is as follows:

- *Union*: **set1 + set2**. RT=T1 if T2<:T1 or RT=T2 if T1<:T2 (RT is the more general one).
- *Intersection*: **set1 * set2**. RT=T1 if T1<:T2 or RT=T2 if T2<:T1 (the more special one).
- *Difference*: **set1 - set2**. RT=T1

  Following shorthands can be used: **s += s2** for s:= s+s2, **s *= s2** for s:= s*s2 and **s -= s2** for s:= s-s2. They can be (and should be) used by the compiler for a more efficient evaluation.

- *Selection*: **elem FROM set-expression : Boolean expression**

The result of a selection is a set, which contains all elements of the given *set expression* which satisfy the constraint given in the *Boolean expression*. The scope of *elem* is the

selection, its type is the base type of the given set expression.
Examples: *s FROM students : Text.Equal(s.name,"Peter")* selects all students[2], who are called "Peter". *p FROM persons - (students + employee) : Text.Equal(p.name,"Paul")* selects persons who are neither students nor employee and whose name is "Paul". Note that set expressions can be nested.

4.2.2 Statements

- The *Assignment* is defined as it is usual in Modula-3. Its general form is:

  set-expression:= set-expression;

  The set-expression on the left hand must evaluate to a writeable variable, the right-hand expression must evaluate to a set value. This value must be assignment compatible to the left-hand variable. That means that the base type of the right-hand expression must be a subtype of the type of the variable (as the subtype relation is reflexive, this includes the case when the types are equal). As a result of the assignment, the set designated by the right-hand expression is copied into the set designated by the left-hand expression. For example: *students^:= SET OF Student{st1, st2, st3, st4, st5}* initializes the student set with the given five student objects. After the assignment *persons^:= students^* the old content of the person set is lost, its new content is the set of students. Note that objects are references in Modula-3, therefore the copy of a set of objects is the copy of a set of references. The assignment of a reference to a set, such as *persons := students* has the same semantics, as any other reference-assignment, i.e. persons points after that to the student set, no copy operation is performed. If the set originally referenced by persons is not referred by any other reference the entire set is collected by the garbage collector.

- *Insert* and *Delete*.
  With insert and delete we can add and remove single elements respectively. No action is taken if an element should be inserted twice or if a non-existing element should be deleted. The type of e_i must be a subtype of the type of set. Their general form is:

  Insert: **set =+ e_1, e_2, ... e_n**
  Delete: **set =- e_1, e_2, ... e_n**

  For example *students =+ st10,st11,st2* adds three given student objects to the set. Similarly *students =- st1,st2* removes the two given objects.

- The *Set-Iteration* statement allows to iterate over all elements of a set. Its general form is:

  FOR elem FROM set-expression DO Statements END

  No action is taken if no element can be selected (e.g. since the set is empty). The scope of *elem* is the iteration statement. For example: *FOR p FROM students DO p.display() END;* displays all persons. As sets are polymorphic and methods are dynamically bound, for every member-object the proper method will be applied.
  FOR s FROM (o FROM students : Text.Equal(o.name,"John")) DO s.display() END; displays all students who are called "John". The Exit statement can be used to leave the set iteration (as all loops in Modula-3). For example, let *user aborts* stand for any requirement to stop the iteration:
  FOR s FROM students DO s.display(); IF "user aborts" THEN EXIT END END;

---

[2]A set designator may be replaced by a designator for a reference to a set, except in the assignment statement.

4.2.3 Further operations on sets

- *Relations* can be used in the usual sense. Two sets are equal (=) if their size is equal and they contain the very same elements. For example *persons^ = students^* is true after the assignment above (*persons^ := students^*). The other relations (#, <, <=, >, >=) are available with the usual semantics.

- The *all quantifier* and the *existential quantifier* give a Boolean value saying whether each element resp. any element satisfies a given predicate. Their general form is:

  ALL [x:=] elem FROM set-expression : Boolean-expression
  ANY [x:=] elem FROM set-expression : Boolean-expression

  The scope of *elem* is the selection, its type is the base type of the given set expression. The assignment part *x:=* is optional. If it is given and the result of *all* or *any* is true then an element of the result set is picked randomly and assigned to *x*. Otherwise, *x* remains unchanged. *x* must be assignment compatible with *elem*. Examples:
  ALL s FROM students : s.matriculation<x is true if all students have a matriculation number smaller than x.
  IF ANY j:= e FROM employee : e.salary>10000 THEN j.display() END; prints one of employee whose salary is greater than 10000, if there is such an employee.

  Both quantifiers are redundant to the selection, but they are in general more efficient:
  ALL e FROM S: B is equivalent to *e FROM S: B = S* (i.e. all elements are selected).
  ANY e FROM S: B is equivalent to *e FROM S: B # {}* (i.e. at least one element is selected).

- *Pick* selects an element from a set randomly. The type of its result is the base type of the set. A pre-defined exception is generated, if the set is empty. Pick is defined as a built-in function:

  PICK(set-expression)

  For example *PICK (persons - students)* gives a person that is not a student.
  FOR i:= 1 TO 10 DO PICK(persons).display() END; displays ten randomly selected persons - maybe the same person ten times.

- *Membership* gives a Boolean value saying whether a given element is contained in a set:

  elem IN set-expression

  For example *st1 IN students* gives true if st1 is contained in students.

- The *cardinality* **| set-expression |** gives the number of elements in the set. | {} | is 0.

4.3 Adding classes to Modula-3

Classes have been defined as set containers that are in a sub- or superclass relation with other classes. This concept of a class can be simply mapped on set variables that are instantiated in a special way. Therefore the signature of the standard function *NEW* must be extended:

NEW(Type, {Set of superclasses} [, Size]);

After the *type* parameter the set of superclasses must be given. This parameter is mandatory for a class definition. The set may be, however, empty. The parameter *size* is optional, just as

by sets. Examples:

```
VAR
    personC := NEW(PersonSet, {},  pSize); (*a class without superclasses*)
    studentC := NEW(StudentSet, {personC}, sSize); (*subclass of person class*)
    employeeC := NEW(EmployeeSet , {personC}); (*subclass of person class*)
    myStudentsC := NEW(StudentSet , {studentC}); (*subclass of person and student class*)
    . . .
    personC += SET OF Person{p1, p2, p3, p3};
    studentC += SET OF Student{s1,s2,s3,s4}; (*s1-s4 become members of studentC and personC*)
    myStudentsC += SET OF Student{s1, s4}; (*s1 and s4 become members of myStudentsC as well*)
    myStudentsC =+ s5,s6;  (*s5 and s6 become members of studentC, personC and myStudentsC*)
```

The class hierarchy can be dynamically managed by the run-time system. Instance inheritance is provided. Due to the classes an explicit clustering is given, which can be used for optimizing the placement of classes and the processing of their operations.

4.4 Adding views to Modula-3

Views have been defined as sets generated over a set expression. They are mapped onto Modula-3 variables with a special instantiation. Objects are references in Modula-3. Therefore, a view on a set of objects, is actually a view on a set of references. This makes it easy to implement updateable views. It is prohibited to insert or modify elements which are invisible in the view. The signature of *NEW* must be extended further:

NEW(Type, set-expression)

Let us consider following example:

```
VAR
    noStudentsV := NEW(PersonSet, personsC - studentsC); (*a view on non-students*)
    youngsV:= NEW(StudentSet, s FROM studentsC : s.matriculation > 100);
                (*view selects students which have a matriculation number greater than 100*)
    myYoungsV:= NEW(StudentSet,  m FROM myStudentsC * youngsV : Text.Equal(m.name,"Ann"));
                (*view selects young students called Ann.*)
 . . .
    FOR ns FROM noStudentsV DO ns.display() END;  (*displays all non-student persons*)
    youngsV=+ s1, s2;  (* inserts s1 and s2 into personsC (superclass of studentsC) and studentsC, if their
                          matriculation number is > 100. No action is taken if they cannot be inserted *)
    myYoungsV=+ s3;      (*inserts s3 into myStudentsC, studentsC and personsC, if s3. matriculation>100 *)
```

5. Adding Persistence and Parallelism

Persistent data extend the (dynamic) scope of variables in *time* (they remain alive after program execution), parallel data (data distributed over a number of address spaces) extend the (static) scope of identifiers in *space*. Data which are both persistent and parallel are extended both in time and space. Therefore, we have to extend the normal rules for scope. We introduce the notion of a *domain*, which spans over a number of address spaces (typically realized by a number of processes or processors) and over volatile and non-volatile storage. A domain has a name and contains a set of address spaces. Names must be unique inside a domain.

5.1 Persistence

The basic idea is to make a distinction between domain *provider* and domain *user* modules. This helps us to maintain information hiding for persistent variables. Persistent variables are primarily defined by domain interfaces. Everything in a domain interface becomes persistent. The content of domain variables survive the completion of the program that created them. The names of a domain are qualified by domain name and interface name. The general form of a domain interface is:

DOMAIN DomainName INTERFACE InterfaceName;
Declarations
END InterfaceName.

Modules exporting a domain interface may define additional domain data. They do it by extending the domain interface inside the module on the outmost level. Such an extension is syntactically the same as a domain interface. The declarations inside these domain interface extensions are simply added to those, given in the true interface. All names inside a (maybe extended) domain interface must be unique. The client modules may import domain interfaces by qualifying the interface name by the domain name. The import may cause to load the persistent data - if necessary. The client has access only to data defined in the true interface, the other data are hidden from it. Listing 1 illustrates a persistent stack:

```
DOMAIN Tools INTERFACE StackProvider;
  TYPE ElemT = INTEGER;
  PROCEDURE Push(elem: ElemT);
  PROCEDURE Pop(): ElemT;
  PROCEDURE Empty(): BOOLEAN;
END StackProvider.

MODULE StackProvider;  (*Exports StackProvider by default*)
  DOMAIN Tools INTERFACE StackProvider;
    VAR
       top: INTEGER;
       stack: ARRAY [1..100] OF ElemT;
  END; (*END DOMAIN*)

  PROCEDURE Push(elem: ElemT) = ...
END StackProvider.

MODULE StackUser
  IMPORT Tools.StackProvider;
BEGIN
  Tools.StackProvider.Push(10); (*push value 10 on the persistent stack*)
  ...
END StackUser.
```

Listing 1: A persistent stack.

As another example let us consider a very simple (persistent) database in listing 2:

```
DOMAIN University INTERFACE ConceptualScheme;
  TYPE
    PersonSet = REF SET OF Person;
    StudentSet = REF SET OF Student;
  VAR
```

```
    personClass := NEW(PersonSet, {},  pSize);
    studentClass := NEW(StudentSet, {personClass}, sSize);
END ConceptualScheme.

INTERFACE ExternalScheme;
  IMPORT ConceptualScheme.University AS Uni;
  VAR
    studentC:= Uni.studentClass;  (*A handle on the persistent class*)
    youngs:= NEW(Uni.StudentSet, s FROM studentC: s.matriculation<100);
             (*A view on the student class*)
END ExternalScheme.

MODULE User;
  FROM ExternalScheme IMPORT studentC, youngs;
BEGIN
  FOR y FROM youngs DO y.display() END; (*display students in the view*)
  studentC:= NIL; (*resets the handle, persistent data remain unchanged!*)
END User.
```

Listing 2: A simple (persistent) database.

5.2 Parallelism

The language support for parallelism is limited to parallel sets. Since classes and views are sets, this implies parallel classes and views.

A set type can be declared to be distributed, with the help of the keyword DISTRIBUTED. Is DS a distributed set of base type T and S a non-distributed set of the same base type, then DS<:S and S<:DS, but DS # S. This implies that S and DS are assignment compatible, but a VAR or READONLY parameter of type DS cannot be substituted by an actual parameter of type S, and the way round. A distributed set is allocated on a set of address spaces, which must be defined at the instantiation of the distributed set. For this purpose, the built-in function NEW is extended with the ON-clause:

NEW(set, ..., ON [VIRTUAL] set-of-address-spaces)

The set of address spaces in the ON-clause must be a set of some ordinal type ("normal" Modula-3 set). The keyword VIRTUAL denotes a virtual address space (a process).

Sets can be distributed automatically or "manually". The ON-clause can be used to qualify any access to a distributed set. The qualification can select portions of a distributed set allocated on a set of address spaces. An example is shown in listing 3:

```
INTERFACE Configuration;  (*Defines the set of processors*)
  CONST
    MaxProcessors = 12;
    Pool          = ProcSet {0 .. MaxProcessors-1};
  TYPE
    ProcSet       = SET OF [0 .. MaxProcessors-1];
END Configuration.

MODULE DistributionExample EXPORTS Main;•
  FROM Configuration IMPORT Pool, ProcSet;
  TYPE
    Persons = DISTRIBUTED REF SET OF Person;
  VAR
    big  := NEW(Persons, ON Pool);          (*big distributed set*)
    small:= NEW(Persons, ON ProcSet{1, 5}); (*small distributed set*)
BEGIN
```

```
   FOR i:= 1 TO 100000 DO
     big^ =+ NEW(Person, name:= ...); (*automatic distribution*)
   END;
   small^:= x FROM big : x.name = "Peter";
    (*select executed on all, result placed into small pool*)
   FOR i:= 1 TO 1000 DO
     big^ ON ProcSet{3} =+ NEW(Person, name:= ...);
      (*manual distribution: all elements are inserted on processor 3*)
   END;
   FOR x FROM small DO big^ =+ x END;  (*redistribution*)
   small:= NIL;
END DistributionExample.
```

Listing 3: A distributed class.

6. Implementation

The basic idea is to keep most part of the implementation outside the compiler. The same idea has been applied in the SRC-Modula-3 compiler [Nels] in connection with Threads. The compiler does the necessary type checks and makes some preparations for optimizing the set operations. The actual implementation is given in form of user modules. The interface of this module is the same for any kind of implementation. The compiler generates calls on this "standardized" interface. The advantage of this approach is its flexibility: alternative resp. experimental implementations can be given. It fits well for a source-to-source compiler. Two implementations are actually in work: cheap sets and PPOST.

6.1 Cheap sets

Cheap sets uses a simple hash table with chaining on collision. The size of the table is determined by the hint given by the application (a default is taken, if no such hint is given, see listing 4). Classes and views are also implemented on top of sets (listings 5 and 6). This implementation is entirely sufficient for non-persistent sets.

Persistent cheap sets are simply loaded at program start (if already available) and stored at completion. Persistent sets are not updated if the program crashes. Parallelism is mapped on standard Modula-3 threads.

6.2 PPOST

PPOST is a memory-resident parallel object store [Bösz], which stores sets or classes of objects in the main memory of a number of processors. It fits very well for implementing parallel and persistent sets and classes.

6.2.1 The Architecture of PPOST

PPOST's main components are (figure 1): "object store" (consisting of a number of object storage machines), "log machine", "checkpoint machine", "archive machine" and "users" (consisting of a number of user machines). All the data of the stored objects (i.e. their attributes and methods) lie in the memory of the storage machines. Every transaction that reads or changes the data is executed on those machines. Transactions are initiated by the user machines and processed by the object store. Changes of the data in the object store are reported to the log machine which saves the information onto a logfile in nonvolatile memory.

The checkpoint machine reads the log produced by the log machine and saves all commited

changes to the disc-based database. Only the checkpoint and the log machine are involved in producing the disc image. The user transaction can go on as soon as the information about the changes is transmitted to the log machine.

The archive machine saves the disc-database to a secondary storage, like a magnetic tape. This is considered as a normal activity of the data-store and again is done in background without interrupting the user-transactions.

We call this pipeline-like way to decouple user-transactions from issues of persistence "vertical parallelism".

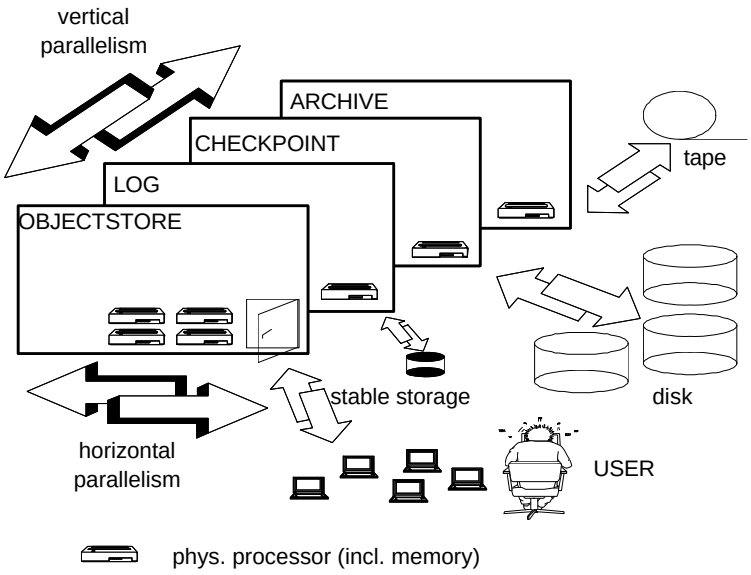

Figure 1: The architecture of PPOST

6.2.2 Scalability

If we have more than one physical storage machine, we can use "horizontal parallelism" either to speed up operations or to scale up the size of the database without loosing performance. When the databases increases in size, we add nodes to the object store. This means, we not only add storage capacity but also computational power. We can show that in many cases it is possible to scale up the size of an object set without degrading the performance of a certain operation on that set by adding nodes [Bösz].

6.3 State of the implementation

Cheap sets is almost fully implemented and the corresponding interfaces can be used directly. A preprocessor has been built that maps the new language features into appropriate definitions and calls of the set, class and view interfaces. It checks the set expressions and statements syntactically, it does not do type checks yet. The development of PPOST is in progress.

7. Appendices

The listings 4, 5 and 6 represent the interfaces of our actual Modula-3 implementation:

```
INTERFACE OIDSet;
IMPORT    OID; (* OID.T = ROOT *)
EXCEPTION SubtypeError; SizeLessOne;
CONST     MinSize  = 64;
TYPE TypeCode = CARDINAL; (*use built-in function TYPECODE*)
     OrderT   = {ASC, DESC, NONE};
     F        = PROCEDURE(e: OID.T);
                  (* Function: apply F on element e *)
     CondF    = PROCEDURE(e: OID.T): BOOLEAN;
                  (* Condition: true if CondF is true for element e *)
     CmpF     = PROCEDURE(e1,e2: REFANY): [-1..1];
                  (* Compare: -1 if e1<e2, 0 if e1=e2, or 1 if e1>e2 e1>e2 *)
     GetF     = PROCEDURE(e: OID.T): REFANY;
                  (* Get: returns either the element e or a value of it *)
     JoinF    = PROCEDURE(e1,e2: OID.T): OID.T;
                   (* Join: returns an element as join of e1 and e2 *)
  T <: Public;
  Public = OBJECT METHODS
         new(type: TypeCode; size:=MinSize) RAISES {SubtypeError};
           (* Initializes self with a type and a hint for the hash table *)
         member(e: OID.T): BOOLEAN RAISES {SubtypeError};
           (* True if element e is in self *)
         exist(c: CondF; VAR x: OID.T): BOOLEAN;
           (* Existential quantifier: true if condition c is true for at
              least one element. If true x is the first hit. *)
         forall(c: CondF; VAR x: OID.T): BOOLEAN;
           (* Universal quantifier: true if c is true for all elements. *)
         insert(e: OID.T): RAISES {SubtypeError};
           (* Inserts element e into self. *)
         delete(e: OID.T): RAISES {SubtypeError};
           (* Deletes element e from self. *)
         equal(s: T): BOOLEAN RAISES {SubtypeError};
           (* True if self and s have the same cardinality and elements. *)
          subset(s :T): BOOLEAN RAISES {SubtypeError};
            (* True if s is a subset of self. *)
         size(): CARDINAL;
           (* Returns the cardinality of self. *)
         copy(s: T; size:=MinSize) : RAISES {SubtypeError};
           (* Self is a copy of s and size is again a hint. *)
         type(): TypeCode;
           (* Returns the type of self. *)
         pick(): OID.T RAISES {SizeLessOne};
           (* Returns randomly an element of self. *)
         apply(f: F);
           (* Sequential iterator: apply f sequential to each element. *)
         parApply(f: F; nodes: CARDINAL:=12);
           (* Parallel iterator: apply f in parallel on n-"nodes". *)
         select(c: CondF): T;
           (* Returns a set of elements in self for which c is true. *)
         add(s: T): RAISES {SubtypeError};
           (* Add all elements of s to self. *)
         union(s: T): T RAISES {SubtypeError};
           (* Returns a set of elements which are in self or in s. *)
         intersection(s: T): T RAISES {SubtypeError};
           (* Returns a set of elements which are in self and in s. *)
         difference(s: T): T RAISES {SubtypeError};
           (* Returns a set of elements which are in self but not in s. *)
         join(s: T; c: JoinF; resTC: TypeCode): T;
           (* Returns a set of members joined by join condition c. *)
         list(o:=OrderT.NONE;cmp:CmpF:=NIL;g:GetF:=NIL) REF ARRAY OF OID.T;
            (* Returns an array of elements, can be sorted by o and cmp. *)
       END;
END OIDSet.
```

Listing 4: The set interface of the set implementation.

```
INTERFACE Class;
IMPORT    OIDSet;
EXCEPTION InvalidInheritance; InvalidBasetype;
CONST     maxSuperC = 5; (* maximum number of super classes per class *)
TYPE      IsA = ARRAY [1..maxSuperC] OF T;

          T <: Public;
          Public = OIDSet.T OBJECT METHODS
               new(type: OIDSet.TypeCode; superCl:= IsA{NIL,..};
                   size:= OIDSet.MinSize; copyUp:= TRUE)
                   RAISES {InvalidInheritance, InvalidBasetype};
          END;
(* new inserts a class into the class hierarchy. Type must be a subtype of
OID.T and of all the types of its super classes, otherwise the exception
InvalidBasetype is raised. The super classes are specified in superCl as an
array. If any super class is already a sub class of self then the exception
InvalidInheritance is raised. This is necessary to avoid cycles within the
class hierarchy. size is the expected size of the class. If copy is true
all the members will be copied into its super classes. The last two
parameters have only performance reasons. *)
END Class.
```

Listing 5: The class interface of the set implementation.

```
INTERFACE View;
IMPORT    Class, OIDSet;
EXCEPTION InvalidBase;
TYPE QueryT =  {SELECT, UNION, DIFF, INTERSECT, JOIN};
     T      <: Public;
     Public =  Class.T OBJECT METHODS
               new(base1,base2: OIDSet.T; (* the base set/classes/views *)
                   query: QueryT;         (* which kind of query *)
                   qExpr: OIDSet.CondF  (* the query expression *)
                  ) RAISES {InvalidBase};
               END;
END View.
```

Listing 6: The view interface of the set implementation.

View.T is a subtype of Class.T. Since Class.T is itself a subtype of OIDSet.T, for which all set operations are defined, all operations are also available for Class.T and View.T. Sets are instances of the type OIDSet.T, while classes are instances of Class.T, and views are instances of View.T.

In listing 7 the median matriculation number of a set of students is computed (m is the abbreviation for the matriculation number):

```
MODULE Median EXPORTS Main;
IMPORT IO;
VAR studs: REF SET OF Student;
    st   : Student;
BEGIN
   IF ANY st:=s FROM studs: |x FROM studs: x.m<s.m|=|y FROM studs: y.m>s.m|
      THEN IO.PutText("the median matriculation# is:"); IO.PutInt(st.m)
      ELSE IO.PutText("no median, studs has an even number of elements.")
   END
END Median.
```

Listing 7: The median matriculation number.

The source above in listing 7 and could be mapped by the compiler to the following Modula-3 source code in listing 8:

```
MODULE Median EXPORTS Main;
IMPORT OID,OIDSet,IO;

PROCEDURE cond001(st: OID.T) : BOOLEAN =
   PROCEDURE cond002(x: OID.T) : BOOLEAN =
   BEGIN  RETURN x.m<s.m  END cond002;
   PROCEDURE cond003(y: OID.T) : BOOLEAN =
   BEGIN  RETURN y.m>s.m  END cond003;
BEGIN
   RETURN studs.select(cond002).size() = studs.select(cond003).size()
END cond001;

VAR studs: OIDSet.T:=NEW(OIDSet.T);
    st   : Student;
BEGIN
   IF studs.exist(cond001,st)
      THEN IO.PutText("the median matriculation# is:"); IO.PutInt(st.m)
      ELSE IO.PutText("no median, studs has an even number of elements.")
   END
END Median.
```

Listing 8: The median in "compiled" form.

Listing 9 shows the declarative power of views:

```
MODULE SocialStatistics EXPORTS Main;
IMPORT IO;
CONST  ManagerSal = 1000.0;
TYPE   PersonS = REF SET OF Person;
       EmployeeS = REF SET OF Employee;
VAR
   persons: PersonS:= NEW(PersonS,{});
   employee: EmployeeS:= NEW(EmployeeS,{persons});
   managers: EmployeeS:= NEW(EmployeeS,
                          m FROM employee: m.salary>=ManagerSal);
   fathers: PersonS:= NEW(PersonS,
                       f FROM persons: f.sex=male AND |f.childrens|>0);
   marriedFathers: PersonS:= NEW(PersonS,
                          fathers
                          * (husband FROM employee:
                          (wife FROM persons: wife.marriedWith=husband)));
   sum,avg: REAL:=0.0;
   anyFather: Employee;
BEGIN
    FOR e FROM employee DO sum:=sum+e.salary END;
    IO.PutText("The average salary: " & Fmt.Real(sum / |employee|) & "!");

    sum := 0.0;
    FOR f FROM marriedFathers DO sum:=sum+f.salary END;
    avg := sum / |marriedFathers|;
    IO.PutText("Married fathers earn" & Fmt.Real(ManagerSal-avg)
               & "less than our managers.");
    anyFather := PICK(marriedFathers);
    IO.PutText("For example, father " & anyFather.name & " earns only " &
               Fmt.Real(anyFather.salary) & "!");
END SocialStatistics.
```

Listing 9: The declarative power of views.

8. References

Bösz L.Böszörmenyi,K.-H.Eder,C.Weich, PPOST - A Persistent Parallel Object Store, Proceedings of the International Conference Massively Parallel Processing Applications and Development ´94, Delft, Netherland, 1994

Cant G.Cantor in: Handbuch der Mathematik; VEB Bibliographisches Institut Leipzig; 1986

Day M.Day: Object Groups May Be Better Than Pages; in the Proceedings of the 4th Workshop on Workstation Operating Systems ´93; Napa, California; IEEE Computer Society Press, 1993

Deux O.Deux et.al.: The Story of O2; IEEE Transactions on Knowledge And Data Engineering, 2(1), 91-108

Elma R.Elmasri and S.Navathe: Fundamentals of Database Systems; 2nd edition; Benjamin/Cummings Comp.; 1994

Gilu M.Gilula: The Set Model for Database and Information Systems; Addison-Wesley; 1994

Kemp A.Kemper: Zuverlässigkeit und Leistungsfähigkeit objekt-orientierter Datenbanksysteme; Informatik Fachberichte 298; Springer Verlag; 1992

Laas Ch.Laasch, Ch.Rich, H.-J.Schek, M.Scholl et.al.: COCOON and KRISYS - A Survey and Comparison; March 1993; Department Informatik; ETH Zürich

Matt F.Matthes, J.Schmidt et.al.: The Database Programming Language DBPL; Bericht Nr. 159; FBI-HH-B-159/92; Dec.92; Universität Hamburg; Fachbereich Informatik

Möss H. Mössenböck: Object-oriented Programming in Oberon-2; Springer Verlag, 1993

Nels G.Nelson: Systems Programming With Modula-3; Prentice Hall Inc.; 1991

Phil M. Philippsen and W. F. Tichy: Modula-2* and its compilation; in Parallel Computation, First International ACPC Conference, Salzburg, 1991

ReWi M. Reiser and N. Wirth: Programming in Oberon - Steps beyond Pascal and Modula-2; Addison-Wesley; 1992

Schw J.Schwartz et.al.: Programming With Sets - An Introduction to SetL; Springer Verlag; 1986

Stro B.Stroustrup: The C++ Programming Language; Addison-Wesley; 1986

SQL2 American National Standard Institute; ANSI X3.135-1989; Database Language SQL; 1989

WiGu N.Wirth and J.Gutknecht: Project Oberon - The design of an Operating System and Compiler; Addison-Wesley; 1992

Zima H.Zima et.al.: Vienna Fortran - A Language Specification Version 1.1; March 1992; Technical Report of the Austrian Center for Parallel Computation ACPC/TR 92-4