FAD, a Powerful and Simple Database Language

Francois Bancilhon
Ted Briggs
Setrag Khoshafian
Patrick Valduriez

Microelectronics and Computer Technology Corp.
Austin, TX 78759

Abstract

FAD is a powerful and simple language designed for a highly parallel database machine. The basic concepts of the language are its data structures (which we call objects) and its programs (defined in terms of operators and predicates). The primary features of the language are (i) the support of complex objects with built-in notion of object identity; (ii) an abstract data type capability; (iii) a persistent object space; and (iv) the efficient support of iteration, conditionals, and set operations. FAD is functional and uses low level operators and operator constructors. This provides for the opportunity of dataflow execution in a parallel architecture. FAD has been successfully implemented in (i) an interpreter working on a main memory database and (ii) integrated in a prototype of a database machine.

1. Introduction

The relational model [Codd 1970] provides simple and powerful features for supporting business applications. However, it is insufficient to handle at the same time new applications like Knowledge Base Systems, Office Automation and Computer Aided Design. For these types of emerging applications, there are two inherent problems. First, the relational model imposes the first normal form. With the first normal form (1) joins are necessary for the construction of hierarchical objects; (2) artificial identifiers must be introduced to perform decompositions of real world objects; (3) it is difficult to *display* hierarchical objects to a user interface. Furthermore, in the relational framework special provisions need be made to accommodatefor objects (tuples) with missing (or null) values. There have been several recent attempts to address this issue and introduce some generality to the relational model by relaxing the first normal form constraint. For example [Jaeschke and Schek 1982] present an algebra for a non first normal form model which allows attributes to be sets of atomic objects. [Zaniolo 1985] on the other hand introduces an algebra for a data model which supports nested tuples (i.e.,tuple valued attributes). More recently [Schek and Scholl 1986] have presented a model where attributes could themselves be relations, which in turn could have relation valued attributes. Other attempts to model un-normalized relations are given in [Ozsoyoglu and Yuan 1985, Ozsoyoglu and Ozsoyoglu 1983, Abiteboul and Bidoit 1984, Hull and Yap 1984, Jacobs 1982, Furtado and Kerschberg 1977, Roth et al. 1984, Thomas 1982]. Each of these models introduce *some* generality to the relational model. A more general model was given in [Bancilhon and Khoshafian 1986]. In this model, a calculus of complex objects based on a sub-object relationship among objects is presented.

Secondly, the relational model does not allow a direct representation for the *sharing* of objects. The models mentioned in the previous paragraph also suffer from this limitation. All these models lack the capability of modeling the *built-in* notion of *object identity* [Khoshafian and Copeland 1986]. With object identity, objects can be distinguished from one another regardless of their content, location, or addressing. Furthermore, objects could be shared. In other words, the object space could be a dag (directed acyclic graph) or even a graph with cycles. An interesting data model which captures object identity is the Logical Data Model [Kuper and Vardi 1984, 1985].

Note that with the relational model the object space is a collection of flat n-ary relations, and with the non-first normal form models the object space is a collection of trees. In these database models and especially the relational model [Codd 1970], objects are identified descriptively through user defined identifier keys. To model dags and graphs, one needs to specify "foreign key" constraints and to compose the hierarchical objects by performing foreign key joins. There are numerous other problems with using user defined descriptive data for object identifications (see [Kent 1978, Khoshafian and Copeland 1986]). For instance, with object identity we do not need to maintain referential and existential integrity constraints. A related semantic and performance motivated advantage is that joins will be replaced by *path traversals* in models which support object identity [Maier and Stein 1986].

A commercially available system which incorporates this notion of object identity in its model is the object-oriented database language OPAL of the GemStone object oriented database system [Maier and Stein 1986, Maier et al.1986]. The model is based on Smalltalk [Goldberg and Robson 1983] which is a representative object oriented programming system (which also supports object identity).

Another approach of modeling objects and inter-object relationships independent of the object content is the *functional data modeling* approach used in languages such as DAPLEX [Shipman 1981], EFDM [Kulkarni and Atkinson 1986], and PROBE [Manola and Dayal 1986]. The functional data model constructs the object space out of "entities" and "functions". Entities are identifiable independent of content (which in the

functional model are obtained by applying a specific set of functions to entities of a given type), and the same entity might exist in multiple relationships, thus providing the capability of object sharing.

In this paper we present a novel database model called FAD (*the Franco-Armenian Data model!*), which supports object identity and allows us to represent arbitrary complex objects built out of atoms, tuples, and sets.

In addition FAD incorporates and combines three other important characteristics. These are (i) operations which make FAD powerful, yet which are amenable to both pipelined and set oriented parallelism for database machine implementation [Boral and Redfield 1985], (ii) separation of the temporary and persistent object spaces, and (iii) the provision for a user defined *abstract data type* layer (which constitutes the atomic object space) [Ong et al. 1984, Osborn and Heaven 1986].

Concerning the *operations* of FAD, two distinct issues were considered: choosing a computational model and choosing the base operators. The computational model is functional. It offers many advantages; in particular, it is suitable for dataflow implementation, which is a natural way to implement parallelism [Ackerman 1982]. The operator constructors were selected to optimize parallelism: the *pump* represents the parallel expression of a divide and conquer strategy and the *filter* is the most elementary expression of parallelism on a set. Also, a basic *group* operation allows the efficient implementation of more complex operations (e.g., duplicate elimination and grouping) through hashing. Finally, explicit set operations (intersection, union and difference) and operations to manipulate tuples (construct and extract) were included in the language.

There have been several approaches to providing users with general purpose programming capability, yet allowing and providing access to a persistent database. In some implementations "external" database calls (through subroutines) are made from a host programming language (e.g., SQL calls in COBOL). This approach suffers from the serious *impedance mismatch* problem. Another approach is extending an existing language with database capabilities (in particular persistence). The classic example of this approach is PS-Algol [Atkinson et al 1983]. It should be noted that the programming language community is recognizing the need of persistence in programming languages [Atkinson et al 1985]. On the other hand database researchers are recognizing the limitations of some of the existing data models. Attempts to design logic based languages [Tsur and Zaniolo 1986], as well as object oriented database models [OODBW 1986] are a response to this requirement.

In addition to the powerful operations and the capability of defining abstract data types in another language, FAD partitions its object space into *persistent* objects and *temporary* objects. Persistent objects are shared among users; temporary objects exist only during a transaction's lifetime and are not visible to other transactions. Although the object space is divided, the operators can be applied to both persistent and temporary objects. We found this separation very useful and powerful since it gave us the capability to impose performance and protection motivated restrictions on the persistent object space. Our implementations verified this fact.

One of the advantages of the relational model is its relative simplicity and elegance. This point could not be overemphasized. We believe many sophisticated models which attempt to offer "more and better" have either failed or will fail because of complexity and lack of robustness. We have attempted to maintain this motherhood of simplicity and elegance in FAD.

Given these basic choices, the language underwent a trial period in which many solutions were investigated. The proposed language has been validated and extensively tested by implementing a FAD interpreter. The current state is the result of numerous compromises between complexity, expressivity and efficiency. FAD is the conceptual interface model of a prototype database machine, Bubba, being built by the Database Program at MCC. Besides the interpreter, we have also integrated FAD with the Bubba's storage manager.

The rest of the paper is organized as follows. The basic concepts of the language are given in Section 2. Section 3 defines programs in terms of operators and predicates. Operators are functions (which return results) or updates (which modify the state of the system). Predicates return true/false answers. In Section 4, we describe the FAD interpreter and in Section 5 we give the summary.

2. Basic Concepts

2.1 User Defined Environment

FAD is built on top of a lower level layer of *abstract data types*. The approach is similar to the ones in [Ong et al. 1984] and [Osborn and Heaven 1986]. This layer defines a set of atomic types, the operations on these atomic types and their implementation. This layer is defined using some conventional programming language such as C. Examples of atomic types are *integer, string, boolean, matrix, degree Farenheit*, etc. With each atomic type (or set of atomic types) is associated a set of operations. For instance, with *string* we define:

concatenation: (*string, string*) -> *string*
substring: (*string, string*) -> *boolean*

Each of these operations is defined by the user and "specified" by a program which implements it. The main advantages of such an approach are:

(i) It provides extensibility. The functionality of the system can be easily enhanced and the base operations do not have to be all frozen.

(ii) It makes FAD a fully general purpose programming language. If the user wants to perform matrix multiplication in FAD, he/she does not have to simulate this feature in a more or less contorted way but can write a routine to do it in his/her favorite programming language.

(iii) It allows the execution of the entire application in the database machine. In a conventional system, the general purpose programming language is *on top* of the database language. In this approach, the database language is *on top* of the general purpose language. This allows us to store the routines *in* the database and the unique features of FAD will allow us to invoke them naturally, which a relational system cannot do.

2.2 System Defined Environment

The system knows about three special sets.

(i) A set *A* of *atomic types* which are abstract data types defined by the user.

(ii) A set *N* of *attribute names*. These are supposed to be recognizable from any other entity in the system (for instance they are distinguishable from user defined types).

(iii) A set *I* of *identifiers* which are also distinguishable from the rest of the world. Those will never be manipulated as such by FAD programs. An identifier provides a unique, invariant reference to an object. One can think about them as pointers or addresses which can be used to reference objects, but can never be printed or updated.

2.3 Objects

An *object o* is a triple (*identifier, type, value*) where:
the identifier is in I,
the type is in $A \cup \{ null\_type, set, tuple \}$

The value of an object depends on its type and is represented in the following manner:

null_type: no value

The *null object* is a unique, special object without a value. It has an identifier **null**, which is a FAD reserved word.

atomic type: an element of a user defined domain of atomic values.

set: $\{i1, i2, \ldots, in\}$
where the ij's are distinct identifiers from I.

The value of a set represents the mathematical notion of a set as an unordered collection of identifiers. The null object is considered to be an element of every set value. An empty set value is represented by { } and is equivalent to { **null** }.

tuple: $[a1{:}i1, a2{:}i2, \ldots, an{:}in]$
where the ai's are distinct attribute names from N, and the ij's are identifiers from I.

The value of a tuple represents a collection of identifiers which are labeled by attribute names. The value taken by the tuple object o on attribute aj is ij and is denoted o.aj .

A tuple is considered to take a value on every attribute name in N. In general, this value is the null object for all but a few attributes. The value of a tuple is generally written to specify only the non-null attribute values. An *empty tuple* has only null attribute values and is represented by [ ].

ordered tuple: $(i1, i2, \ldots, in)$

An *ordered tuple* is a special case of a tuple where the attribute names (ai's above) are consecutive integers starting with 1.

An *Object System* is a set of objects. An object system is *consistent* iff (i) identifiers are unique for each object (unique identifier assumption) and (ii) for each identifier present in the system there is an object with this identifier (no dangling pointers assumption). All object systems will be assumed consistent in the sequel.

2.4 Graphical Representation of Objects

An object system can be represented by a *graph*. The graph of an object system is defined as follows.

(i) Each object is represented by a node, the type of which is determined by the object type. A node representing an atomic type is denoted by ✻ its value. A node representing a set type is denoted by . A node representing a tuple type is denoted by a ◆. It is sometimes convenient to further label the nodes with the identifiers of the objects they represent.

(ii) If object o, represented by node n, is of type tuple and $o.a$ is the identifier of object o', represented by *node n'*, then there is an arc labeled a going from node n to node n' .

(iii) If object o, represented by node n, is of type set and the identifier of object o', represented by *node n'*, is in the value of object o, then there is an unlabeled arc going from node n to node n' .

Consistent object systems are represented by graphs such that from each tuple node there is at most one arc with a given label and terminal nodes (i.e., nodes which do not have any out arcs) are either empty set nodes, empty tuple nodes or atom nodes.

An object system is *clean* if its graph is a forest. It is *acyclic* if its graph is a DAG.

Figure 1 illustrates the graphical representation of a FAD object where s1, s2 and s3 are set identifiers and t1, t2, t3 and t4 are tuple identifiers. This object describes two persons who have a common child.

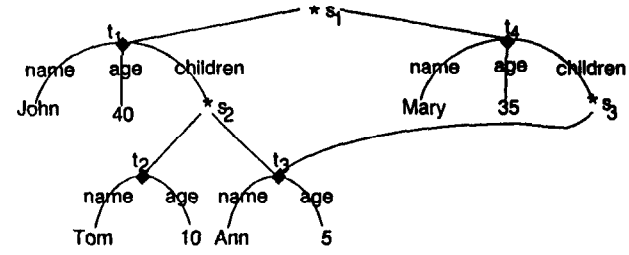


Figure 1: A FAD Object System

2.5 Linear Representation of Objects

In an acyclic object system, an object with identifier i has a linear representation denoted by *unfold(i)*, which is defined recursively as follows:

null_type:	nothing
atomic type:	value of the object
set:	{ *unfold(i1), unfold(i2), ..., unfold(in)* }
tuple:	[*a1: unfold(i1), ..., an: unfold(in)*]
ordered tuple:	(*unfold(i1), unfold(i2), ..., unfold(in)*)

A linear representation is a structured collection of atomic values. Thus an object may be specified with a linear representation purely with the atomic values which compose it, through set and tuple constructors. Note that only objects in a clean object system can be uniquely represented by their linear representations. Since a linear representation is value based, the identity of contained objects is lost and it is no longer possible to distinguish objects which have the same linear representation.

Examples of linear representations of simple objects include:

```
1, 2, 2.5, "john", "doe" are atomic objects

{1, 2, 3} is a set

[name:"john_doe", age:27] is a tuple

("john", "doe") is an ordered tuple

[name:[first:"john",  last:"doe"],
 children:{[name:[first:"joe",last:"doe"]],
           [name:[first:"ann",last:"doe"]]}]
```

2.6 Object Equality

Several languages have distinguished different forms of equality. A strongly typed language such as ML [Harper, Mac-

Queen, and Milner 1986], provides a single overloaded predicate '=' which means identity with references, equality with values, and recursive application of the record structure. In languages without strong typing the distinction between predicates must be explicit. The Lisp family (cf. [Abelson and Sussman 1985]) traditionally provides EQ, which tests addresses and is hence implementation dependent, and EQUAL, which tests for isomorphic structures. Object oriented languages must also provide predicates which distinguish between objects. Smalltalk [Goldberg and Robson 1983], provides two predicates: equivalence, '==' which is tests for the same object, and equality, '=' which is implemented separately for each class. The LDM (Logical Data Model) of Kuper and Vardi [Kuper and Vardi 1984, 1985] also distinguishes a "shallow" equality which compares the r-values of objects.

In FAD, we have three sorts of equality. The first is equality of identifiers. In a clean object system two objects have equal identifiers if and only if they are the same object (i.e. objects are *distinct* if their identifiers are not equal). We shall call this the *identical* predicate, which corresponds to '==' in Smalltalk.

Two objects are *value-equal* iff they have identical types and equal object values. Equality of object values is defined as follows: (i) two atomic object values are equal if they denote the same element in the user defined domain of values, (ii) two set values are value-equal if there exists a bijection f between the values of the two sets such that if y = f(x) then x and y are identical, and (iii) two tuples are value-equal if the values they take on every attribute are identical. Value-equal corresponds to r-value equality in LDM and it is a *shallow* equality. Intuitively, the objects might be different but their types and contents are identical.

Finally, if we are interested in "equality" of the linear representation of objects in an acyclic object system, we have the *all-equal* predicate which is defined recursively as follows: (i) two atomic object representations are all-equal iff they are value-equal, (ii) two set objects are all-equal if every element of one is all-equal to an element of the other, and (iii) two tuple objects are all-equal if the values they take on every attribute are all-equal. It can easily be seen that two objects which are all-equal have linear representations which are the same up to duplicates and permutations of elements for sets and permutations of <Attribute: Value> pairs for tuples.

2.7 Programming Environment

The programming environment is composed of two object systems: *persistent* and *temporary*. The two object systems differ only in the lifetime of the objects they contain. Temporary objects exist only during program execution, while persistent objects persist between program executions.

The database is a persistent object with identifier **database**. Initially, the persistent object system contains only this single object. Other persistent objects are created by updates to the database. FAD can support a database object represented by a directed graph structure.

3. FAD Programs

3.1 Base Concepts

A FAD program is either an *operator* or a *predicate*.

Programs take FAD object identifiers, attribute names, and abstractions as input and return FAD object identifiers (for operators) or **true** or **false** (for predicates) or cause an exception to be raised (error).

The input arity of any program is fixed at compile time. The output arity of all programs is one. Note that an operator may return multiple results by returning the identifier of an ordered tuple, the value of which are the identifiers of the results.

Predicates return the boolean values **true** and **false** which are FAD reserved words and *not* objects.

Operators are either *functions* or *updates*. Functions return an identifier as output. Updates do the same, but also modify the value of their input objects as a side effect.

A function returns either the identifier of an existing object, the identifier of a new temporary object generated by the function, or an error. The temporary object system is modified by a function only if a new object is created. In any case, the persistent object system is not modified by the application of a function.

An update modifies the state of an object system. It either (i) modifies the value of the object whose identifier was given as input and returns the identifier of the modified object or (ii) the update fails, nothing is modified, and an error is generated. An update can be viewed as a function with side effects.

Operators and predicates may be of two kinds: *base* and *constructor*. The base operators and predicates provide the underlying functionality of FAD. Constructors allow more complicated operators and predicates to be built from simpler ones. A base predicate takes object identifiers as input. A constructor predicate takes boolean values as input. Both return boolean values. A base operator takes object identifiers and attribute names as input. An operator constructor takes as input object identifiers as well as abstractions of operators and/or predicates. The reserved word **fun** is used to denote lambda abstraction[Pingali and Kathal 84]. Abstraction in FAD is allowed only to specify anonymous operators and predicates as arguments

In the following sections, the syntax and semantics of FAD operators and predicates is given. When defining the semantics of an operator, the symbol <- denotes assignment. If the specified conditions are not met, an error is raised. Examples will be written in `typewriter font` and the symbol => will be used to indicate the result of evaluating an expression. In the examples, we will use the linear representation of objects.

3.2. Base Functions

Let *e* be an expression denoting identifier *i* and let *e1, e2, ..., en* be expressions denoting identifiers *i1, i2, ..., in*. In the sequel, when we give examples of FAD base functions, we will use the linear representation of objects. Several groups of base functions exist and are defined as follows:

Object constructors are expressions which denote the identifier of a new temporary object with the given value. They are used to create new temporary objects. There are object constructors for atomic types, tuples, ordered tuples, and sets.

(1) *<abstract data type name>* (*<value string>*) is an expression denoting the identifier of new atomic object provided the name and value are valid.

Examples of atomic constructors are:

`int (3)`	integer object with value 3
`string ("john")`	string object with value "john"
`date (July 23, 1986)`	date object with value July 23, 1986

(2) [*a1:e1, a2:e2, ..., an:en*] is an expression denoting the identifier of a new temporary tuple with value [a1:i1, a2:i2, ..., an:in]

Examples of tuple constructors are.

`[ ]`

`[name:"john"]`

```
[name:"john", age:30]

[name:[first:"john", last:"doe"]]
```

(3) *(e1, e2, ..., en)* is an expression denoting the identifier of a new temporary ordered tuple with value [1:i1, 2:i2, ..., n:in].

An example of ordered tuple constructor is:

```
( "john", "mary" )
```

(4) *{e1, e2, ..., en}* is an expression denoting the identifier of a new temporary set with value {i1, i2, ..., in}. Duplicate identifiers are eliminated.

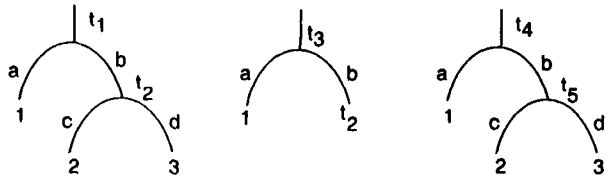
Examples of set constructors are:

```
{ }

{ "john", "mary", "john"}

{ {"john", "mary"}, {"joe", "betty"}}

{ [name:"john", age:30], [name:"mary"]}
```

Copy Functions are expressions which denote the identifier of a new temporary object of the same type and value as the specified object. There are two copy functions, corresponding to two notions of equality.

(5) **copy** *(e)* is an expression denoting identifier i' of a new temporary object which is a copy of the object identified by i. That is, the two objects have identical types and values, i.e., they are value-equal. This is a *shallow* copy of the object, i.e., if the object is a tree, only its first level will be duplicated.

In the following example, the expression *copy (t1)* generates the object identified by t3 and returns t3.



(6) **all_copy** *(e)* is a new object obtained from e by generating a new identifier i' for every sub-object of e and replacing its identifier with i'. The graphs of e and all-copy(e) are isomorphic. Furthermore, e and all-copy(e) as well as all their corresponding sub-objects are all-equal. The expression **all_copy** *(t1)* generates the object identified by t4 and returns t4 (in the example above).

Tuple Functions provide operations on tuples.

(7) *e.a*, where a is an attribute name, is an expression denoting i.a, if i is the identifier of a tuple.

Examples include:

```
[a:1].b => null

{ "john" }.a => error

[name:"john", age:30].name => "john"

("john", "joe", "betsy").2 => "joe"

[name:[first:"john",last:"doe"],age:30].name
.first => "john"
```

Set Functions provide the usual union, difference, and intersection operations on sets. The set functions union, intersection, and difference return identifiers of new temporary sets. If the argument is not a set, an error is encountered.

For notational convenience:
set (i, X) indicates *X is the value of the set object with identifier i*
Assume set (i1, X1) and set (i2, X2) for the following definitions.

(8) **union** *(e1, e2)* is an expression denoting the identifier of $\{ x \mid x \in X1 \vee x \in X2\}$

An example of union is:

```
union ({i1,i3}, {i3,i5}) => {i1,i3,i5}
```

(9) **intersection** *(e1, e2)* is an expression denoting the identifier of $\{ x \mid x \in X1 \wedge x \in X2\}$

(10) **difference** *(e1, e2)* is an expression denoting the identifier of $\{ x \mid x \in X1 \wedge x \notin X2\}$

User Defined Functions are abstract data type functions or represent a user defined fad function.

(11) *f (e1, e2, ..., en)*, where f is an n-ary user defined function or an abstract data type function name, is an expression denoting the identifier of the result of applying f to i1, i2, ..., in.

For example :

```
sum (10.2, 4)    => 14.2
concat ("IR", "IS") => "IRIS"
```

3.3. Base Updates

FAD has a general assignment statement plus some specialized ones for sets and tuples.

Let e be an expression denoting identifier i and let e1, e2, ..., en be expressions denoting identifiers i1, i2, .., in. For clarity, we will use the linear representation of objects in the examples. For notational convenience:
set (i, X) indicates *X is the value of the set object with identifier i*

(1) **assign** *(e1, e2)* is an expression denoting i1, if i1 is not null, with the following side effect:

type (i1) <- type (i2)
value (i1) <- value (i2)

The following example modifies the age of "john"

```
assign ([name: "john", age: 30].age, 20)
```

Even though this assignment is sufficient to do most updates, it is reasonable both for performance and expressiveness reasons to introduce assignment operating on slots of objects. These are specialized by object type.

(2) **tupleassign** *(e1, a, e2)*, where a is an attribute name, is an expression denoting i1, if i1 is a tuple, with the following side effect:

i1.a <- i2

The following example modifies the department of "john"

```
tupleassign ( [name:"john", dept:"sales"],
     dept,"marketing")
```

(3) **delete** *(e1, e2)* is an expression denoting i1, if set (i1, X1) and set (i2, X2), with the following side effect:

$X1 \leftarrow \{ x \mid x \in X1 \wedge x \notin X2\}$

The following example deletes the objects with identifiers i1 and i2 from the set

```
delete({i1, i2, i3}, {i1, i2}) => {i3}
```

(4) **insert** *(e1, e2)* is an expression denoting i1, if set (i1, X1) and set (i2, X2), with the following side effect:

X1 <- { x | x ∈ X1 ∨ x ∈ X2}

The following example inserts "john" as a new employee:

```
insert(database.emps,
       {[name:"john", age:27]})
```

(5) *f (e1, e2, ..., en)*, where f is an n-ary user defined or an abstract data type update name, is an expression denoting the identifier of the result of applying f to i1, i2, ..., in.

3.4. Predicates

FAD provides base predicates of boolean value, tests for object type, and tests for equality. The predicate constructors provided are not, and, and or.

Let e be an expression denoting identifier i and let e1, e2, ..., en be expressions denoting identifiers i1, i2, in. Let p1 and p2 be predicates with boolean values v1 and v2 respectively. For clarity, we will use the linear representation of objects in the examples.

(1) **is_tuple** *(e)*, **is_set** *(e)*, **is_atom** *(e)*, **is_singleton** *(e)* are expressions denoting true if i identifies respectively a tuple, a set, an atom or a singleton set and false otherwise.

```
is_tuple ([name: "john", age: 30]) => true
is_set ({1}) => true
is_singleton ({null}) => false
is_atom (null) => false
```

(2) **identical** *(e1, e2)* is an expression denoting *identical*(i1, i2).

(3) **equal** *(e1, e2)* is an expression denoting *value-equal(i1, i2)*. This tests for shallow equality. Considering the example in section 3.3., we have :

```
equal (t, copy (t)) => true
equal (t1, t3)  => true
equal (t1, t4)  => false
```

(4) **all_equal** *(e1, e2)* is an expression denoting *all-equal(i1, i2)*. This tests for the deep equality. Considering the example in section 3.3., we have :

```
all_equal (t, all_copy (t)) => true
all_equal (t1, t4) => true
all_equal (t1, t3) => true
```

(5) **not** *(p1)* denotes the predicate value ¬v1.

(6) **and** *(p1, p2)* denotes a predicate value v1 ∧ v2.

(7) **or** *(p1, p2)* denotes a predicate value v1 ∨ v2

(8) *p(e1, e2, ..., en)*, where p is an n-ary user defined predicate or abstract data type predicate name, is an expression denoting the identifier of the result of applying p to i1, i2, ..., in. If the type or value of the arguments is not appropriate, error is returned.

For example :

```
substring ("RI", "IRIS") => true
greater (2, "john") => error
```

3.5. Abstraction and application

Operator and predicate abstraction allow the construction of anonymous functions which are used as arguments to operator constructors.

(1) **fun** *(x1, x2, ..., xn) e* is an n-ary operator abstraction with parameters x1, x2, ..., xn. The body of the operator abstraction is the expression e. The scope of the parameters is restricted to the body of the abstraction.

For instance:

`fun(x) x` denotes the identity function.

`fun(x) x.a` denotes the function returning the a attribute value.

`fun(x,y) [a:x,b:y]` denotes a function returning a new tuple with attributes a and b.

(2) **fun** *(x1, x2, ..., xn) p* is an n-ary predicate abstraction with parameters x1, x2, ..., xn. The body of the predicate abstraction is the predicate p. The scope of the parameters is restricted to the body of the abstraction.

For instance:

`fun(x) true` denotes the constant predicate true

`fun(x) equal(x,"john")` denotes a predicate which tests for "john"

(3) **let** *r1* **be** *e1*
let *r2* **be** *e2*
...
let *rn* **be** *en*
in *e*

where r1, ..., rn are reference names and e1, e2, ..., en and e are expressions, is an expression. It denotes the expression e in which every occurrence of a reference name is replaced by the identifier denoted by it's associated expression. Within a reference declaration block, references are defined in the order written.

Application expressions are introduced to allow a name (i.e., reference) to be bound to a constant. References are useful to eliminate common sub-expressions from expressions and thus avoid duplicate work.

Examples of application are: (assume *sum* and *mult* are functions returning the sum and product of two integers)

```
let x be int (3)
   in
   [a:x, b:2, c:x]   => [a:int(3), b:2, c:int(3)]
```

The following updates the total tax paid by an employee, Emp, and the payroll of department, Dept, based on the amounts paid to and withheld fom the employee:

```
let withhold be mult(0.10, Emp.salary)
let paid be sub(Emp.salary, withhold)
in
   (add(Emp.tax, withhold),
    add(Dept.payroll, paid)
```

(4) *p -> e*, where p is a predicate denoting a value v and e is an expression which denotes an identifier i, is an expression denoting i if v is true and null otherwise.

For example, the following expression returns the age of "john" or null:

```
equal(x.name, "john") -> x.age
```

3.6. Operator constructors

In this section, we will base our examples on the following database which contains a set of departments (called dept) and a

set of employees (called emps). The employees are of the form:

```
Employee =
     [ename: string, enum: integer,
      salary: real, department: Department ]
```

and departments are of the form:

```
Department =
     [dname: string, dnum: integer,
      budget: real, employees: {Employee}]
```

The database schema is (assuming a typing capability in FAD):

```
define type database =
   [dept:{ Department }, emps: { Employee }]
```

Let f and g denote n-ary operator abstractions. Let e be an expression denoting identifier i and let e1, e2, ..., en be expressions denoting identifiers i1, i2, ..., in. For clarity, we will use object values instead of object identifiers in the examples. For notational convenience:

set (i, X) indicates *X is the value of the set object with identifier i*

(1) **ifthenelse** *(p, f, g, e1, e2, ..., en)*, where p is an n-ary predicate abstraction, is an expression denoting

f(i1, i2, ..., in) if p(i1, i2, ..., in) denotes true
g(i1, i2, ..., in) if p(i1, i2, ..., in) denotes false

(2) **whiledo***(f, e)* is an expression, if i is an ordered tuple, denoting:

i if f (i.1, i.2, ..., i.n) is null
whiledo (f, f (i.1, i.2, ..., i.n)) otherwise

The whiledo construct is primarily used for fixpoint computations. The transitive closure of a binary relation R of ordered tuples is:

```
whiledo( fun(x,y) not(equal(y,{})) ->
          (union(x,y),
          filter(fun(z,w) equal(z.2,w.1)->
                          (z.1,w.2),
                 R, y),
       R, R).1
```

(3) **filter** *(f, e1, e2, ..., en)*, is an expression denoting, if set (i1,X1) and ... and set (in,Xn), the identifier of
{f(x1, x2, ..., xn)| x1 ∈ X1, ..., xn ∈ Xn }

The filter constructor is the most elementary expression of parallelism on a set, i.e., each x of X can be applied "f(x)" in parallel.

The following example gives the name and salary of employees who make more than 50K. The result is a set whose elements are of the type {[en: string, sal : integer]}.

```
filter (fun(x)  greater(x.salary, 50K)
            ->[en:x.ename, sal: x.salary],
        database.emps)
```

The following example gives the department name and the names of employees of all departments whose budget is greater than 100K. The result is a set whose elements are of type {[name: String, employees: { Employee }]}.

```
filter(fun(x) greater(x.budget, 100K) ->
          [name: x.dname,
           employees:
              filter(fun(y)  y.ename,
                     x.employees)],
       database.dept)
```

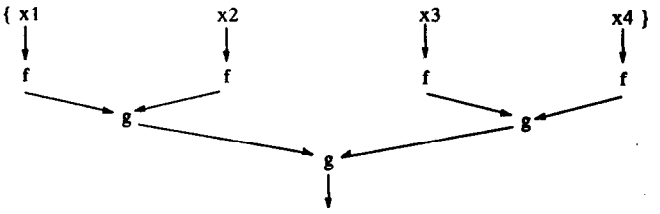
The following example gives the employee *name* of all employees who work in department toy and the employee *number* of all employees who do not.

```
filter (fun(x)
          ifthenelse (fun (z)
             equal(z.department.dname,"toy"),
             fun (z) z.ename,
             fun (z) z.enum,
             x) ,
        database.emps)
```

(4) **pump** *(f, g, e)*, where f is a unary operator abstraction and g is a binary function abstraction which is associative and commutative, is an expression, if set(i,X), denoting:

f(x) if X = {x}
g(pump(f, g, A), pump(f, g, B)) if X = A ∪ B ∧ A ∩ B = ∅

The pump operator supports parallelism by a divide and conquer strategy as shown on the following graph :



The pump operator's main application is aggregate functions. The average employee salary can be found as follows, where totals is an ordered tuple (count, sum of salaries).

```
let totals be
  pump (fun(x)(1, x.salary),
        fun(x,y) (sum(x.1, y.1),
                  sum(x.2, y.2) )
        database.emps ) in
quotient(totals.2, totals.1)
```

(5) **group** *(f, e)*, is an expression, if set(i, X), denoting:

identifier of { (a,{x}) | x ∈ X ∧ all_equal(f(x), a)}

In other words, the group operator returns a set of pairs representing the equivalence classes of the set S under application of the function abstraction f. The pair consists of the representative value for the class and the set of identifiers of objects in the class. It gives a general framework for handling uniformly such functions as hashing, grouping, duplicate elimination.

The following illustrates the use of group to eliminate the duplicate values in a set X of abstract data types as follows :

```
filter (fun(x) x.1, group (fun(y) y, X))
```

The following example groups employees by salary:

```
group (fun(x) x.salary, database.emps)
```

The following example produces a set of employee names without duplicates :

```
filter (fun(x) x.1, group (fun(y) y.ename,
        database.emps))
```

4. The FAD Interpreter

We implemented the FAD interpreter (in C), primarily to test the operational semantics of the language, but also to gain ex-

perience in the use of FAD itself by writing large application programs.

The FAD interpreter provides an environment in which the user may run and debug FAD programs. It is implemented with a main memory database which resides on a file between sessions. Within the interpreter, the user has capability to manipulate objects directly through both commands and fad programs. The overall architecture of the interpreter is given in Figure 2.

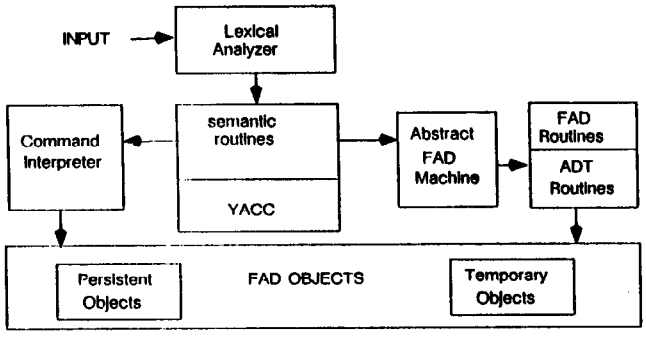


Figure 2: Architecture of the FAD Interpreter

Both temporary and persistent FAD objects are implemented as entries in the Temporary and persistent Object Tables respectively. Each entry contains a type, reference count, and a value. The identifier of an object is given by a bit, to indicate which table it belongs, plus an index into the table. The value of set object is a pointer to a list of identifiers. The value of a tuple object is a pointer to a list of attribute, identifier pairs.

Abstract Data Types are implemented as a table of user supplied routines given at compile time. The user must supply certain routines for the interpreter to manipulate each new abstract data type. These include: converting to and from the string representation used in a FAD program and the actual internal storage representation, testing for equality, and returning the size in bytes of a given value. Abstract Data Type Values may be stored either in the object table (small, fixed length data) or as a pointer to the actual data.

The interpreter automatically reads from and writes to a file persistent object linear representations at the beginning and end of a session. During a session, persistent objects reside in the persistent Object Table in main memory. Temporary objects reside in the Temporary Object Table which exists only during an interpreter session. Reference counts are used to remove unneeded objects.

To facilitate debugging, development, and testing object "handles' which are basically object identifiers are visible to the user within the interpreter. The value and linear representation of an object may be printed using interpreter commands. FAD programs may be written using object identifiers directly. This allows a user to interactively construct a complex query and examine the intermediate results.

Interpreter commands supplement FAD by allowing the user to print object values and store object values in files. Files may also be used to store and edit programs and command sequences. FAD programs may be debugged using breakpoint, trace, and step commands.

Thus, the FAD interpreter consists of several components: a command interpreter, FAD program executor, temporary and persistent object tables. The parser, implemented using YACC, generates both requests to the command interpreter and code for the FAD abstract machine.

The FAD abstract machine is a stack based machine which executes FAD base operators and predicates as single instructions. Functions constructors are compiled as sequences of code in which the functional arguments are evaluated via subroutine calls. The stack is used to pass arguments and results.

5. Summary

In this paper, we have presented FAD, a language designed to be the interface to a highly parallel. Some of FAD's more relevant features are summarized here.

FAD provides a rich set of built–in structural data types and operations. Instead of normalized relations, FAD operates on sets and tuples which can be nested within each other to an unlimited degree, so that hierarchical structures can be directly represented as either internal data or results. One of the strongest points of FAD is the explicit existence of the concept of set, which is inherently parallel (all elements in a set can be processed in parallel). Furthermore, the FAD objects have an identity enabling support of graph structures and replacement of most joins by path traversals. Instead of relational algebra operations, FAD provides a lower–level set of functions which are more general and yet allow maximum parallelism. The FAD functions allow for efficient implementations of set operations.

FAD is a pure functional language except for updates. It consists of a set of base operators and operators constructors. All functions return identifiers of either new objects or previously existing objects. FAD uses nested functions to model dataflow dependencies and includes a function which allows common sub–expression elimination. These dataflow dependencies exhibit parallel and pipelined execution possibilities of FAD programs in a database machine. FAD partitions the object space into persistent and temporary. Update functions are applied to the persistent objects and they have side effects since they cause state changes to previously defined objects.

Most data languages provide a closed set of built–in atomic data types which are not extendable by users. Instead, FAD is built on top of a user–defined domain of abstract data types so that users can define their data types using another programming language such as C. Thus, any number of atomic data types can be added or modified to satisfy changing requirements for functionality or compatibility.

FAD was designed in 1985–1986. It has been successfully implemented and extensively tested first using a FAD interpreter working on a main memory database on UNIX. It is now being implemented as the interface to the Bubba database machine.

As defined in this paper, FAD is an untyped language. Current work includes the support of disjunctive, prescriptive and recursive typing in FAD and of a general ADT capability where ADT operations can be defined in a FAD data definition language.

Acknowledgements

Many thanks are due to Haran Boral, George Copeland and Shamim Naqvi who provided invaluable comments and suggestions on the FAD language. The authors are also grateful to Brian Hart for his great help in the implementation of the FAD interpreter.

References

[Abiteboul and Bidoit 1984] *"An Algebra for Non Normalized Relations"*, S. Abiteboul and N. Bidoit, ACM Int. Symp. on PODS, March 1984.

[Abelson and Sussman 1985] *Structure and Interpretation of Computer Programs*, H. Abelson and G.J. Sussman with J. Sussman, The MIT Press, Cmabridge, MA, 1985.

[Ackerman 1982] *"Dataflow Languages"*, W.B. Ackerman, Computer, February 1982.

[Atkinson et al. 1983] *"An Approach to Persistent Programming"*, M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm and R. Morrison, Computer Journal, Vol. 26, November 1983.

[Atkinson et al. 1985] *Persistence and Data Types Papers from the Appin Workshop*, M. Atkinson, P. Buneman, and R. Morrison, Editors, University of Glasgow, 1985.

[Bancilhon and Khoshafian 1986] *"A Calculus for Complex Objects"*, F. Bancilhon and S. Khoshafian, ACM Int. Symp. on PODS, March 1986.

[Boral and Redfield 1985] *"Database Morphology"*, H. Boral and S. Redfield, Int. Conf. on VLDB, Stockholm, Sweden, August 1985.

[Codd 1970] *"A Relational Model of Data for Large Shared Data Banks,"* E.F. Codd, CACM, Vol. 13, No. 6, June 1970.

[Furtado and Kerschberg 1977] *"An Algebra of Quotient Relations,"* A. L. Furtado and L. Kerschberg, ACM SIGMOD Int. Conf., Toronto, Ontario, June 1977.

[Goldberg and Robson 1983] *Smalltalk-80: The Language and Its Implementation*, A. Goldberg and D. Robson, Addison-Wesley Publishing Co., Reading, MA, 1983.

[Harper, MacQueen, and Milner 1986] "Standard ML", R. Harper, D. MacQueen, and R. Milner, ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[Hull and Yap 1984] *"The Format Model: A Theory of Database Organization,"* R. Hull and Chee K. Yap, In JACM, Vol. 31, No. 3, July 1984.

[Jacobs 1982] *"On Database Logic"*, Barry E. Jacobs, JACM, Vol. 29, No. 2, April 1982.

[Jaeschke and Schek 1982] *"Remarks on the Algebra of Non First Normal Form Relations,"* G. Jaeschke and H. Schek, ACM Int. Symp. on PODS, Los Angeles, 1982, 124-138.

[Kent 1978] *Data and Reality*, W. Kent, North-Holland Publishing Co, New York, 1978.

[Khoshafian and Copeland 1986] *"Object Identity"*, S. Khoshafian and G. Copeland, Proc. of 1st Int. Conf. on OOPSLA, Portland, Oregon, October 1986.

[Kulkarni and Atkinson 1986] *"EFDM: Extended Functional Data Model"*, K. G. Kulkarni and M. P. Atkinson, The Computer Journal, Vol. 29, No. 1, 1986.

[Kuper and Vardi 1984] *"A New Approach to Database Logic"*, Gabriel M. Kuper and Moshe Y. Vardi, ACM Int. Symp. on PODS, Waterloo (April 1984).

[Kuper and Vardi 1985] *"On The Expressive Power Of The Logic Data Model,"* G. M. Kuper and M. Y. Vardi, SIGMOD 1985, Austin, Texas, 180-187.

[Maier et. al. 1986] *"Development of an Object-Oriented DBMS,"* D. Maier, J. Stein, A. Otis, A. Purdy, Proc. of 1st Int. Conf. on OOPSLA, Portland (Oregon), October 1986.

[Maier and Stein 1986] *"Indexing in an Object-oriented DBMS"*, D. Maier and J. Stein, in [OODBW 1986].

[OODBW 1986] Int. Workshop on Object Oriented Database Systems, Pacific Grove, CA, September 1986.

[Ong et al. 1984] *"Implementation of Data Abstraction in the Relational Database System INGRES"*, J. Ong, D. Fogg, M. Stonebraker, ACM Sigmod Record, Vol. 14, No. 1, 1984.

[Osborn and Heaven 1986] *"The Design of a Relational System with Abstract Data Types of Domains"*, Sylvia L. Osborn and T.E. Heaven, TODS Volume 11, Number 3, September 1986.

[Ozsoyoglu and Yuan 1985] *"A Normal Form for Nested Relations,"* M.Z. Ozsoyoglu and L.Y. Yuan, In Proc. Fourth Annual ACM Symposium on Principles of Database Systems, ACM, Portland, OR, 1985, pages 251-260.

[Ozsoyoglu and Ozsoyoglu 1983] *"An Extension of Relational Algebra for Summary Tables,"* M.Z. Ozsoyoglu and G. Ozsoyoglu, Proceedings of the 2nd International (LBL) Conference on Statistical Database Management, Los Angeles, 1983, 202-211.

[Pingali and Kathail 84] *"An Introduction to Lambda Calculus"*, K. Pingali and V. Kathail, Laboratory for Computer Sciences, MIT, July 1984.

[Roth et al 1984] *"Theory of Non-First-Normal-Form Relational Databases,"* M. Roth, H. Korth, and A. Silberschatz, TR-84-36, Department of Computer Science, University of Texas at Austin, 1984.

[Schek and Scholl 1986] *"The Relational Model with Relational Valued Attributes"*, H. J. Schek and M. H. Scholl, Information Systems, Volume 11, No. 2, 1986.

[Shipman 1981] *"The Functional Data Model and the Data Language DAPLEX"*, ACM TODS, Vol. 6, No. 1, 1981.

[Thomas 1982] *"A Non-First-Normal-Form Relational Database Model,"* S. Thomas, PhD Dissertation, Vanderbilt University, 1983.

[Tsur and Zaniolo 1986] *"LDL: a Logic Based Data Language"*, S. Tsur and C. Zaniolo, Int. Conf. on VLDB, Kyoto, August 1986.

[Zaniolo 1985] *"The Representation and Deductive Retrieval of Complex Objects"*, Int. Conf. on VLDB, Stockholm, Sweden, August 1985.