A FAD for Data Intensive Applications

Scott Danforth and Patrick Valduriez

Abstract— **FAD is a strongly-typed database programming language designed for uniformly manipulating transient and persistent data on Bubba, a parallel database system developed at MCC. This paper provides an overall description of FAD, and discusses the design rationale behind a number of its distinguishing features. Comparisons with other database programming languages are provided.**

Index Terms—**Complex objects, data model, database programming language, parallel database system, query optimization.**

I. Introduction

MANY knowledge based application domains such as CAD, CAM, CASE, and office automation require efficient representation and manipulation of "complex objects"—arbitrarily nested data structures including sets and tuples. Once created, such data structures may persist between different program invocations, and may be concurrently accessed by multiple users. Although databases support persistent data, and generally support concurrency (in the sense of multiprogrammed transactions), many database data models do not support complex objects. A case in point is the relational model [1]. Because relational systems impose the first normal form constraint, complex objects must be mapped into a collection of "flat" relations. With this approach, much of the inherent semantics of complex object composition is lost, and potentially expensive foreign key joins are required to recover and use this information.

There have been several attempts to address representational deficiencies of the relational model by relaxing the first normal form constraint—for example by developing a relational algebra that allows attributes to be sets of atomic objects [2]. Another approach is to develop an algebra that supports tuple valued attributes [3]. More recently Schek and Scholl have presented a model where relational attributes may themselves be relations [4]. A variety of approaches to modeling and representing unnormalized relations have been suggested [5]–[11].

In addition to posing representational problems, traditional database languages are often not computationally complete. For example, it is not possible to evaluate the transitive closure of a binary relation in relational algebra. One response to this problem has been to embed database management calls within a general purpose programming language. Unfortunately, such couplings generally suffer from an "impedance mismatch" between the programming language data model and the database data model [12]. This problem is seen in the interface between embedded SQL products and their host languages. Another approach is to add persistence and database oriented data types to the data model of an existing general purpose language. While addressing the impedance mismatch problem, such an add-on approach may produce nonuniformities within the resulting language. Pascal/R [13], in which the Pascal data types are augmented with relations, provides an example of this [14].

The above approaches generalize the representaitonal and computational capabilities of systems based on the relational model, but do not support sharing of objects [15]. Sharing is a powerful and useful notion for data modeling, and allows a given object to be considered part of more than one data structure; when a shared object is modified, all parent structures see the result. Database data models that support sharing include the Logical Data Model [16], GEM [17], and O2 [18], as well as the Entity-Relationship model [19], and the semantic data model SDM [20].

FAD is a general purpose database programming language that uses data types representing a natural extension of the relational model in order to support complex objects with sharing. The language is a research tool that supported and became part of an investigation of issues important to efficient general purpose utilization of multiprocessor database architectures [21]. In the remainder of the Introduction, we state our overall design rationale for FAD and provide an overview of some important concepts. Section II presents the FAD data model, focusing on its type domains and the corresponding type expression language. Section III presents the FAD language, used for creating and manipulating data, and Section IV describes FAD modules, the unit of compilation. Section V discusses how our design objectives influenced our decisions, and Section VI provides comparisons of FAD with other programming languages. We conclude with a critical evaluation of FAD based on our experiences using the language.

A. FAD Overview

FAD (which stands for Franco-Armenian Data language [22]) was developed with the intention of providing an easy to use, powerful, general purpose programming language interface to Bubba, a parallel database system developed at MCC [23]. The overall design objective for FAD was that it be a simple, declarative language, incorporating support for general purpose computational mechanisms and a uniform model of transient and persistent data. It was desired that the data model include sets, tuples, shared complex objects, and null data as necessary for handling a wide variety of realistic database applications.

Manuscript received September 15, 1989; revised April 15, 1991. The work described in this paper was performed at MCC, Austin, TX.

S. Danforth is with IBM Zip 9641, Austin, TX 78759.

P. Valduriez is with INRIA Rocquencourt, 78153 Le Chesnay, Cedex, France.

IEEE Log Number 9105289.

Problems to be solved at the FAD compiler level included optimization for distributed data access, and automatic parallelization of FAD programs for dealing with horizontally declustered data within an overall dataflow model of computation. We chose this approach to demonstrate the feasibility of developing compiler technology for efficient utilization of "shared-nothing" multicomputer databases [24], [25], and also to allow the use of FAD as an intermediate target for other languages (in particular, LDL [26]) wanting an easy access path to the scalable high performance offered by the Bubba architecture.

The original FAD language was based on ideas and syntax suggested by FP [27]. A number of high level, declarative set oriented operations were provided as primitives, and major emphasis was placed on mapping these into parallel operations on Bubba. Subsequent enhancements to FAD retained the expression-based functional style of the original version, but provided additional flexibility and a more hospitable syntax. To support strong static typing, the original data model was enhanced and provided with a formal type expression language. Type expressions were then made available in FAD programs, and used as the basis for a data definition language (for describing the persistent database). Finally, a compiler was developed to implement strong static typing with type inference [28], optimization for distributed data access [29], and parallel code generation for execution on Bubba [30]. The multiprogrammed parallel system on which FAD programs now execute has been operational since early 1989 [21].

B. Data and Identifiers

The FAD data model supports arbitrary nestings of complex data structures, and is based on atomic values in conjunction with the following data structures: tuples (i.e., records), disjuncts (i.e., variant records), sets, and updatable objects. The word *object* is used in a special sense here; values and objects are distinguished as different types of data in FAD. A value may have any structure (simple or complex, as determined by its type), but may never be updated or modified in any way. An object has a modifiable state (again, of arbitrary structure, as determined by its type), and can be shared. Subsequent use of the word object in this paper should be understood in this special sense. We will use the terms *data item* and *complex data structure* when speaking of objects in the generic sense.

In FAD programs, any given data item can be named by associating an identifier with it. Such an association is lexically scoped, and is called an identifier definition. Within the scope of an identifier definition, any occurrence of the identifier denotes the data with which it is associated. We avoid using the term *variable* because it typically encompasses ideas related to data storage and updatability, as well as naming. Assignment in FAD is an operation on objects, not data identifiers.

C. Uniform Persistent and Transient Data

FAD supports a single database, identified in programs as *db*. Although some applications might require access to multiple databases, this simplification serves FAD's purpose as a research tool without precluding future enhancements. Persistence of data in a FAD program is defined by reachability (data are persistent if and only if it is reachable from db). There is no other distinction between transient (i.e., program-created) and persistent data in FAD—they are treated uniformly, and persistence is orthogonal to type [14]).

In Bubba, the database is physically distributed over a collection of "intelligent repositories" (IR's), each composed of a local processor, memory, disk, and communication interface [31]. Distribution of data within Bubba (including horizontal declustering of relations over multiple IR's in the interest of parallel execution) is invisible within FAD programs. The FAD compiler automatically optimizes and parallelizes general purpose FAD programs for execution on Bubba, using abstract interpretation for analysis of the data required by program actions [29], [30], [32].

D. Strong Static Typing

By strong typing we mean that a given data item must always have the same type during its lifetime. Static typing guarantees that this type is known at compile time, and allows type errors to be identified before program execution while supporting generation of efficient code for data manipulation and structure access. FAD incorporates a type expression language used for describing the types of all data created and manipulated by a program. This type language provides the basis for a data definition language (DDL) used to describe the persistent database accessible to FAD programs. DDL is compiled into a persistent schema, which, in addition to maintaining the conceptual (i.e., user level) types of db data, also specifies a mapping from conceptual database types to their physical representations. Although the conceptual types of two database data items may be the same, their physical representations may be different, based on decisions expressed in FAD DDL.

A unification-based algorithm for strong static typechecking is used in conjunction with abstract interpretation to infer unstated types and identify type errors in a FAD program at compile time [28]. The type inference algorithm for FAD is not syntactically complete—there exist ambiguous FAD programs with more than one valid type assignment. Such situations are easily handled by explicitly declaring the necessary types. An example of an ambiguous untyped program would be one that accepts two arguments, x and y, and adds them using the operator $+$. In this case, x and y could be integers or floating point numbers; a solution is to explicitly indicate the types of x and y.

E. Actions and Functions

In FAD, the term *action* is used to indicate a computation that returns data after possibly accessing or updating data. Actions in FAD thus correspond to expression evaluation and command execution. User-defined functions in FAD allow specification of actions that are parameterized with respect to data. FAD provides a fixed set of higher order functions (called *action constructors*) for writing programs. These construct aggregate actions from action and function expressions. A number of FAD action constructors are provided for declar-

atively expressing powerful operations on sets. For instance, a filter action conceptually applies a function to each element of the Cartesian product of a number of sets to produce a new set composed of the application results, thus combining generalized selection, projection, and n-ary join capabilities within a single operation.

II. The FAD Data Model

We now present the FAD data domain and the corresponding type expression language. As a first step, atomic types and structured type constructors are introduced. The FAD data domain is then based on the atomic types and closed with respect to a set of recursive domain equations involving the type constructors. Types in FAD are domains (i.e., sets) of distinguishable data elements. We do not concern ourselves here with the particular representation of the individual elements within a semantic domain, but feel free to name individual elements when they are limited in number or of special importance.

A. Atomic Types

A representative sampling of the FAD atomic types includes:

bools	the set {true, false, null}
ints	a set of computer integers ∪ {null}
floats	a set of computer floating point numbers ∪ {null}
strings	a set of computer strings (of arbitrary length) ∪ {null}.

Null is an element of every FAD type (including the constructed types, introduced below), and is generally considered by FAD actions as representing an unknown element of a given type. It can be used in tuples to indicate "no-information" for fields that have not yet been assigned. By itself and without a surrounding context, the FAD constant expression null is ambiguous; it could be any type. As indicated earlier, however, static typing guarantees that the type of every data expression in a FAD program is known at compile time, either as a result of explicit typing, or as a result of type inference—otherwise a compile-time type error is issued. Using null as an argument to a function never results in a run-time error, although many functions are defined to return null if any of their arguments are null.

B. Structured Types

The structured type constructors are now introduced. They will be used in the following section to construct semantic domains for FAD tuples, disjuncts, sets, and objects. FAD does not provide a list constructor because binary tuples provide this capability. An array constructor was omitted because keyed sets provide this capability.

A *tuple* is a partial function from label values to data elements called fields. The corresponding tuple type is a domain characterized by the type of the label values (either strings or ints), and a fixed number of associations between different field labels and the corresponding field types. Square brackets are used to represent the tuple type constructor. Thus, for example, the type expression *[a:ints, b:[floats, bools]]* denotes the domain of tuples that map the string "a" to an element of ints, and the string "b" to an *ordered tuple* (with implicit integer field labels). This ordered tuple maps the integer 1 to an element of floats, and the integer 2 to an element of bools. Because of the similarity of ordered tuples to those whose fields are labeled with strings, ordered tuples will not be considered further in this section.

A *disjunct* is a tuple in which only one field is allowed to be nonnull. A tag method is provided for disjuncts that returns the label of the field stored in a disjunct. If no field is stored, a null label is returned. Vertical brackets are used to represent the disjunct type constructor. Thus, for example, the type expression *|a:ints, b:strings|* denotes the domain of all disjuncts that, if tagged with "a" store an integer, and if tagged with "b" store a string.

A *set* is an unordered collection of data elements, all of which have the same type. No two elements of a set are identical (two data items are identical if and only if they are the same element of a data domain). The corresponding set type is a domain characterized by the type of the set elements, and an optional key specification (no two elements of a keyed set are allowed to have the same key value). Curly brackets are used to represent the set type constructor. Thus, for example, the type expression *{[a:ints, b:strings]; key is a}* denotes the domain of all sets of tuples of the specified type, for which no two set elements have the same "a" field. Multisets (i.e., sets containing duplicate elements) were omitted in FAD because sets of objects provide this capability.

An *object* is a sharable data item consisting of a unique, unchanging identity and an updatable state. An object type is characterized by indicating the semantic domain of its state, and is represented in FAD semantics as the domain (object-ids $\times$ t), where t is the state domain. The identity portion of an object is a logical address that uniquely identifies the updatable state portion of an object, and therefore supports sharing. Although the primitive operations of FAD are allowed to access the identity and state portions of an object separately as necessary to perform their functions, the FAD programmer is only given access to an object as an integral combination of identity and state. The identity portion of an object is not FAD data (there are no pointers in the language), and neither in general is the state portion. This is a result of the FAD data domain equations presented in the next section. Obj is used to represent the object type constructor. Thus, for example, the type expression *obj([a: ints, b: obj(strings)])* denotes the domain of updatable sharable object tuples that map the string "a" to an integer value, and the string "b" to an updatable sharable atomic object whose state is a string. All objects support an assign operation, which replaces an object state with another from the same semantic domain. Other (incremental) changes to object state depend on the structure of the object: individual fields of an object tuple may be replaced using tupleassign; the contents of an object set may be changed by insert and delete.

C. The FAD Data Domain

Fig. 1 defines the FAD data domain in terms of the above atomic types and type constructors. Every FAD data item is an

$$
\begin{aligned}
\text{Data} &= \text{Values} + \text{Objects} \\
\text{Values} &= \text{Atomics} + \\
&\quad [\text{Values}] + \\
&\quad |\text{Values}| + \\
&\quad \{\text{Values}\langle\text{optional-key-spec}\rangle\} \\
\text{Objects} &= \text{obj}(\text{Atomics}) + \\
&\quad \text{obj}([\text{Data}]) + \\
&\quad \text{obj}(|\text{Data}|) + \\
&\quad \text{obj}(\{\text{Data}\langle\text{optional-key-spec}\rangle\})
\end{aligned}
$$

Fig. 1. The FAD data domain.

element of a FAD type, and the FAD types are exactly those that are constructed as indicated in Fig. 1. In the interest of brevity, constructor syntax is generalized in Fig. 1 to allow (for example) expressions of the form $[t]$, which denotes the domain of tuples all of whose component fields are elements of the semantic domain t.

An important aspect of the above equations concerns the way in which values and objects are stratified: objects can contain values, but not vice versa. The primary reason for defining the FAD data domain this way was our desire to

1) *provide a conservative extension to the relational data model that supports complex data structures with updates and sharing, and*
2) *enforce the idea that values do not change; if any portion of a data item can change over time, it must be an object.*

The first of these objectives is primarily a result of our belief that relational database technology is a well-developed approach for management of data. We wanted to stay fairly close to the relational model in order to assure that SQL applications would be easily mapped to FAD, and to use proven optimization techniques developed in the context of distributed relational databases.

The second objective arose from a desire to cleanly reflect updatability in the type system. Value types in a FAD program are known at compile time, and knowing that values cannot be updated is useful when generating code for efficient execution on a parallel, distributed architecture (in which special consideration for updatable objects is required). Also, when doing scavenging garbage collection in support of data clustering, the underlying system is guided by types and can make effective use of this information.

The FAD data model achieves both of the above objectives. With respect to the first objective, sets of tuples provide a natural model for relations. Also, the absence of pointers in FAD means that when an object is placed in a tuple, the operational view seen by the FAD programmer is that the complete object is there (as opposed to a pointer that must be dereferenced in order to access the object state). This coincides with the relational view of data as being stored "in place" in tuples. This view also relates to the second objective. Because the view uniformly presented is that data are stored in place, and because no part of a structured value such as a tuple is allowed to change over time, placing an object within a structured value makes no sense (since the object could change over time, and it is seen as being part of any containing structures).

As an aid to visualization of objects and sharing in FAD, Fig. 2 provides a graphical depiction of three FAD data items (named A, B, and C) from the FAD types$\{ints\}$, *obj{ints}*, and *obj([obj(ints), {ints}])*. Dashed lines surround the objects in this example, identified as id-1, and id-2. Data items B and C share the object whose identity is id-1. Note that, by itself, the state portion of the object named C is not an element of the FAD data domain defined in Fig. 1. The state of C is not a value because it is a structure that contains an object; it is not an object because it has no identity portion (id-2 is not part of the state; it is part of the overall object named C, which is FAD data). This highlights the fact that the FAD programmer is only given access to an object as an integral combination of identity and state; in general, the state of an object is not even an element of a FAD type.

D. The FAD Type Expression language

Fig. 3 presents a grammar for FAD type expressions. Within this paper, informal grammar segments are provided to summarize syntax. Nonterminals are enclosed in angle brackets, the notation $\langle xyz\rangle^*$ indicates zero or more occurrences of the xyz nonterminal; $\langle xyz\rangle^+$ indicates one or more occurences. Undefined nonterminals are given self-explanatory names.

In addition to the atomic types and the type constructors already introduced, Fig. 3 shows that FAD type expressions include *type references.* These allow referring to type domains by name, using identifiers (called type-ids) that appear on the left-hand side of a schema type declaration. In order to illustrate type-ids, type references, and the type expression language as a whole, Fig. 4 presents an example database schema. Line numbers are placed in FAD comments to aid discussion.

(*1*) associates *ages* with the type denoted by the type expression on the right-hand side of the declaration—in this case, the domain named ints.

(*2*) associates *person* with an object tuple type. Person is automatically inferred to be an object tuple type because the age field is an object type (recall that any FAD type constructed from an object type must itself be an object type). The right-hand side of the type equation for person is thus a supported shorthand for the expression, *obj([name : strings, age : object(ages), dept : insts])*. This type is the set of all object tuples that map "name" to an element of type strings, "age" to an element of type *obj(ages)*, and "dept" to an element of the type ints. As an example of a type reference, $person.age$ is the type of the age field for a person, i.e., *obj(ages)*. The domain obj(ages) is identical to the domain *obj(ints)* because ages = ints. Because person is an object tuple type, the fields of any data of type person may be replaced (using tupleassign) without loss or change of the object tuple's identity. Also, the state of the age field may be replaced (using assign) without loss or change of its identity.

(*3*) associates *people* with an object set type. This type is the domain of all FAD object sets containing elements of type

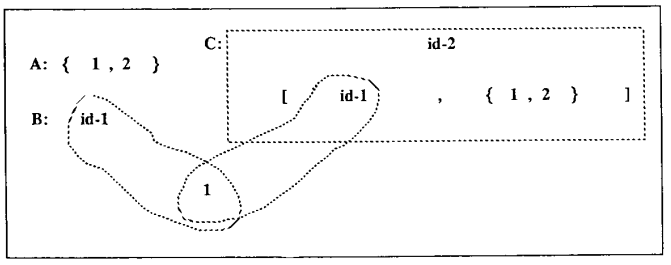


Fig. 2. Three data items.

$$
\begin{aligned}
\langle \text{type\_expr} \rangle &::= \langle \text{atomic\_type\_id} \rangle \\
&\mid \langle \text{type\_construction} \rangle \\
&\mid \langle \text{type\_reference} \rangle \\
\langle \text{atomic\_type\_id} \rangle &::= \text{bools}|\text{ints}|\text{floats}|\text{strings}|\cdots \\
\langle \text{type\_construction} \rangle &::= [\langle \text{field\_spec} \rangle^{+}] \\
&\mid |\langle \text{field\_spec} \rangle^{+}| \\
&\mid \{\langle \text{type\_expr} \rangle \langle \text{optional-key-spec} \rangle\} \\
&\mid \text{obj}(\langle \text{type\_expr} \rangle) \\
\langle \text{type\_reference} \rangle &::= \langle \text{type\_id} \rangle \\
&\mid \langle \text{type\_reference} \rangle \cdot @ \\
&\mid \langle \text{type\_reference} \rangle \cdot \langle \text{field\_label} \rangle
\end{aligned}
$$

Fig. 3. The FAD type expression language.

(*1*) ages = ints
(*2*) person = [name : strings, age : obj(ages),dep : ints]
(*3*) people = {person; key is name}
(*4*) db = [employees : people, depts : obj({ints})]

Fig. 4. An example schema.

person (person is a type reference). In addition, no element of the people domain contains two person tuples with the same name field. This is because name is a key for sets of the constructed type. As another example of a type reference, *people*.@ is the type of the elements of a set of type people, namely person. Because people is an object set type, the contents of any set of type people may be modified (using insert or delete) without loss or change of the set's identity.

(*4*) associates *db* with an object tuple type. The two fields of such a tuple, employees and depts, respectively, contain people and an updatable set of integer values. The type-id *db* is reserved, and is normally dewithoufined using FAD DDL. Although the above schema is a legal DDL persistent schema specification, other information such as indexes and physical representations are normally also provided. The following type expressions (the last three of which are type references) all denote exactly the same type: obj(ints), person.age, people.@.age, and db.employees.@.age. Although not illustrated in this example, the ordering of type declarations is immaterial, and recursive type declarations are supported (e.g. intlist = [ints, intlist]).

III. The FAD Action Language

The FAD action language is a lexically scoped expression language used to represent two things: actions and first-order functions (actions parameterized with respect to FAD data). Every action expression in FAD returns data, and without exception may be followed with an optional type expression enclosed in angle braces indicating the type of this data. For example, the expression $+(1, 2\langle ints \rangle)\langle ints \rangle$ denotes an action that returns an integer. In the following presentation (with occasional exceptions), explicit typing of FAD actions will be omitted. The FAD compiler infers these types, and the focus of this section concerns the action language.

A. User-Defined Functions

Function specifications appear in top level function declarations, and can also be used as arguments to action constructors that take functions as arguments (e.g., filter). Free data identifiers are allowed in function specifications as long as these identifiers are defined within an enclosing lexical scope.

$$\langle \text{fun\_spec} \rangle ::= \text{fun}(\langle \text{params} \rangle)\langle \text{action\_expr} \rangle$$

B. Actions

FAD is intended to be a simple, regular language, with an intuitive inventory of actions. The following grammar presents an overall view of the different FAD action categories.

$$
\begin{aligned}
\langle \text{action\_expr} \rangle ::= &\langle \text{simple\_action} \rangle \\
&\mid \langle \text{structure\_construction} \rangle \\
&\mid \langle \text{structure\_selection} \rangle \\
&\mid \langle \text{structure\_modification} \rangle \\
&\mid \langle \text{function\_application} \rangle \\
&\mid \langle \text{action\_construction} \rangle
\end{aligned}
$$

In examples that follow, we occasionally use simple let expressions (before presenting the let action constructor in Section III-C). These uses are intended to be self-explanatory. For example, the expression $let\ x = 1\ in + (x, 2)$ denotes an action that returns the value 3.

1) Simple Actions: A simple action is either a constant (e.g., 3.14) or an identifier that names data. The reserved identifier *db* names the persistent database.

2) Structure Construction: Structure constructions return structures that hold data. Structures in FAD are tuples, dis-

juncts, sets, and objects.

$$\begin{aligned}\langle\text{structure\_construction}\rangle ::= & [\langle\text{field\_expr}\rangle^*] \\ | & \ |\langle\text{field\_expr}\rangle| \\ | & \ \{\langle\text{action\_expr}\rangle^*\} \\ | & \ \text{new}(\langle\text{action\_expr}\rangle)\end{aligned}$$

For example, the expression *[a : "a", c : 3.14]* denotes an action that returns a value tuple of type [a:strings, c:floats]. The disjunct and set constructions are similarly straightforward, but object constructions require additional comment. The FAD function *new* is applied to a value, and returns an object whose state is the value. Thus, for example, the expression *new({1})* denotes an action that returns a new object set of type obj({ints}). The only other possibility with respect to object-construction is creation of a structure that includes an object. Such structures, when built, are automatically objects, due to FAD's stratified data model. For example, the FAD expression *let* x = *new(1) in {x}* first creates a new, unique atomic object named x whose state is an integer, and then returns a new, unique object set whose single element is the object named x—the type of the overall expression is obj({obj(ints)}).

3) Structure Selection: Structure selections are actions that return data held by structures.

$$\begin{aligned}\langle\text{structure\_selection}\rangle ::= & \ \langle\text{action\_expr}\rangle.\ \langle\text{action\_expr}\rangle \\ | & \ \langle\text{action\_expr}\rangle@\langle\text{action\_expr}\rangle \\ | & \ \text{value}(\langle\text{action\_expr}\rangle)\end{aligned}$$

The infix tuple select operator . is left associative, and is used to represent field selections from tuples and disjuncts. In most languages with tuples (or records), field labels in tuple selections are not data, but syntax processed at compile time. Although FAD avoids this restriction, the field selector is usually a constant (a special case that is optimized) rather than an identifier or an arbitrary data expression.[1] The infix set selection operator @ is left associative, and is used to represent element selections from sets that are keyed. The set being selected from is indicated to the left of the @, and the value of the search key is indicated on the right of the @. Theresult returned is the set element that has the indicated key if such an element exists, otherwise null (of the type of the set elements) is returned.

For example, the following action expression (in which a set selection is composed with a tuple selection) returns the value 42. The set in this example is given an explicit type in order to indicate an index key on the "a" attribute:

$$\{[\text{a} : 1, \text{b} : 1], [\text{a} : 2, \text{b} : 42], [\text{a} : 3, \text{b} : 1]\} < \{[\text{a} : \text{ints}, \text{b} : \text{ints}]; \text{key is a}\} > @2.\text{b}$$

The *value* structure selection operation returns a value that corresponds to the state of an object. If the state of an object is a value, then this value is returned. If the state of an object is a structure that contains objects, then a corresponding structure all of whose subobjects have been (recursively) converted to values is returned.

4) Structure Modification: Structure modification operations are provided for each FAD structure. These operations (with the exception of *assign*, which is only meaningful on objects) may be used on either objects or values. When applied to an object, they update (destructively change) the object state and return the modified object as the operation result. When applied to a value, they create and return a new value, incrementally different from the original.

$$\begin{aligned}\langle\text{structure\_modification}\rangle ::= & \ \text{tupleassign}(\langle\text{action\_expr}\rangle, \langle\text{action\_expr}\rangle, \langle\text{action\_expr}\rangle) \\ | & \ \text{insert}(\langle\text{action\_expr}\rangle, \langle\text{action\_expr}\rangle) \\ | & \ \text{delete}(\langle\text{action\_expr}\rangle, \langle\text{action\_expr}\rangle) \\ | & \ \text{assign}(\langle\text{action\_expr}\rangle, \langle\text{action\_expr}\rangle)\end{aligned}$$

Tupleassign is used to replace a tuple field. The first argument of the tupleassign operation must either be a tuple or an object whose state is a tuple. The second argument is a field label, and the third argument is the data to be placed in the designated field. As with tuple selections, the compiler will treat undefined identifiers in the second argument position as string constants. For example, the action *let x = new ([age: 1]) in tupleassign(x, age, 3)* returns the object named x, modified so it contains the value 3 in its age field.

Insert is used to add a new element to a set. The first argument of the insert operation must either be a set or an object whose state is a set. The second argument is the element to be inserted. Delete is used to remove an element from a set. The first argument of the delete operation must either be a set or an object whose state is a set. The second argument is the element to be deleted. The action *let x = new ({1,2}) in insert (x, 3)* returns the object set named x, modified so it contains the values 1, 2, and 3.

Assign is used to update an object—i.e., replace its state with another from the same semantic domain. The assign operation requires an object as its first argument, and the second argument may be either an object or a value. If the second argument is a value, then the assignment replaces the state of the first argument with this value. If the second argument is an object, the assignment replaces the state of the first argument with the state of the second. For example, the action *let x = new ([age: 1]) in assign (x, [age: 2])* returns the object tuple named x, modified so its state contains the tuple [age:2].

[1] As a notational convenience, the FAD compiler assumes that an undefined identifier found in the field label position is intended as a string constant. For example, assuming that the identifier b is undefined, both of the expressions [a:1, b:2].b and [a:1, b:2]."b" mean the same thing (an action that returns the value 2).

5) Function Application:

$$\langle \text{function\_application} \rangle ::= \langle \text{fcn-name} \rangle (\langle \text{arg s} \rangle)$$
$$\langle \text{fcn-name} \rangle ::= \langle \text{id} \rangle | \langle \text{typeclass} \rangle . \langle \text{id} \rangle$$

As shown by the above grammar segment, there are two different representations for function names. The first possibility is simply an identifier, which is the usual form. Because a number of FAD functions are overloaded, however, a typeclass can also be specified in order to remove any ambiguity. All FAD functions have a typeclass with which they are associated. This represents a grouping of functions according to the kind of data on which they act. For instance, object-sets.insert and value-sets.insert are the explicitly typeclassed insert operations for objects whose state is a set, and for value sets, respectively. Another example is ints.+ and floats.+, which are the addition functions for ints and floats.

—Example Function Applications—

$$+(2, 3)$$
$$\text{ints.}+(2.3)$$
$$\text{user-defined-fcn}(x, y)$$

a) Equality Tests: It is not the purpose of this paper to discuss all the functions provided by FAD—these include the usual set of arithmetic, logical, and string manipulation operations. But it is useful to discuss the equality tests, which closely reflect the FAD data domain structure. As with the original FAD language [22], there are three levels of equality testing available, provided by the following functions:

data.eq?
objects.equal?
objects.all-equal?

Each equality test requires two arguments of the same type. As indicated by the typeclasses, the eq? function is available on all data; the functions equal? and all-equal? are available on all objects.

Eq? tests for identity of two data items in FAD. Two data items are identical if and only if they are the same element of a FAD data domain, thus identity for objects can simply compare object-ids for equality. For example, *eq?(new(1), new(1))* returns false, while the expression *let x = new(1) in eq?(x, x)* returns true. Identity for values is based on recursive comparison of all structure levels down to atomic values, with sets being eq? if they have identical elements. Eq? treats null as known data, thus *eq?(null⟨ints⟩, null⟨ints⟩)* returns true, and *eq?(null⟨ints⟩, 1)* returns false. There is an important interaction between eq? and set elements in FAD: No two elements of a set are identical, therefore no two elements of a set are ever eq?. this is not the case for the weaker equality tests, equal? and all-equal?—two elements of an object set may be equal? or all-equal?.

Equal? tests for shallow equality of two objects in FAD by using eq? to compare object states in the case of atomic objects, or the top level state components in the case of structured objects. Thus, for example, *equal?(new(1), new(1))* returns true, and *equal?([new(1)], [new(1)])* returns false. All-equal? tests for deep equality of two objects. All equal?$(o1, o2)$ is equivalent to eq?$(\text{value}(o1), \text{value}(o2))$ with one exception: converting an object set to a value removes duplicates, but sets of objects are only all-equal? if they have the same number of all-equal? elements. Sets of objects are therefore treated by all-equal? as multisets of object states. Thus, for example,

$$\textit{all-equal?}(\{\text{new}(1), \text{new}(2), \text{new}(2)\}, \{new(1), new(2), new(2)\}) \text{ returns true, and}$$
$$\textit{all-equal?}(\{new(1), new(2), new(2)\}, \{new(1), new(2)\}) \text{ returns false.}$$

C. Constructed Actions

The last category of FAD action expression is called an action construction. These allow definition of data identifiers (let), control over the sequencing of actions (let, begin-end, do-end, if, whiledo), specialized processing of sets (filter, pump, group), and unstructured control transfers (escape, and abort).

1) Let: Let expressions are the FAD mechanism for identifier definition. A let expression represents an action that sequentially orders interspersed groups of parallel identifier definitions and actions, followed by execution of a final action whose result is returned as the overall let action result.

$$\langle \text{let} \rangle ::= \text{let} \langle \text{def\_group/act} \rangle^+ \text{ in} \langle \text{action\_expr} \rangle$$
$$\langle \text{def\_group/act} \rangle ::= \langle \text{id\_def} \rangle^+ | \langle \text{action\_expr} \rangle$$
$$\langle \text{id\_def} \rangle ::= \langle \text{id} \rangle = \langle \text{action\_expr} \rangle$$
$$\langle \text{id} \rangle \% \langle \text{id} \rangle = \langle \text{action\_expr} \rangle$$

Each definition group is a comma-separated collection of identifier definitions. Sequencing within a definition group is undefined (allowing parallel execution of the actions whose results are named). Scoping of identifier definitions extends from their introduction through the final ⟨action_expr⟩ that concludes the let expression, with later definitions masking earlier ones in the case of name conflicts. Expressions within a definition group are evaluated using the identifier definitions in effect upon entry to the definition group, and the same identifier cannot be defined twice within the same group.

A let expression need not contain identifier definitions. This is a useful special case, since it provides a mechanism for sequentially performing a number of actions for their side-effects, and then returning the result of a final action. There are two varieties of identifier definition. The first, involving a single identifier, is the usual form and requires no elaboration. The second, involving a % followed by a second identifier, allows access to action status results.

In addition to a data result, every FAD action sets an integer status code. For example, in the case of set insertions, a status code indicates whether the inserted element was already in the set. This status is normally invisible to FAD programs, but the second identifier definition syntax allows FAD code to receive and inspect this status—the second identifier names the status returned by the action indicated to the right of the equal sign.

The escape action constructor (discussed below) allows user-defined functions to escape their execution context and return status codes.

—example Let actions—

```
let x = 1, y = new(2)
  assign(y, 3)
in + (x, y)          (*returns 4*)
let x = new([a : 1, b : 2])
  y = let tupleassign(x, a, 2) in x.b
in + (x.a, y)        (*returns 4*)
```

2) Begin-End and Do-End:

⟨begin-end⟩ :: = begin⟨action_expr⟩$^+$ end

⟨do-end⟩ :: = do⟨action_expr⟩$^+$ end

These action constructors allow explicit sequencing of actions performed for their side-effects. The actions they construct return null⟨ints⟩. Actions in a begin-end construction are executed sequentially. Actions in a do-end construction are executed in no particular order, and may be executed in parallel.

3) If:

⟨if⟩ ::= if⟨test_expr⟩ then ⟨then_expr⟩

| if⟨test_expr⟩ then ⟨then_expr⟩ else ⟨else_expr⟩

If expressions support conditional execution. The constructed action first executes the action denoted by ⟨test_expr⟩ (which must be of type bools), and, if the result is true, then executes and returns the result of the action denoted by ⟨then_expr⟩; otherwise, the ⟨else_expr⟩ action is executed and its result returned (if there is no ⟨else_expr⟩, null of the same type as ⟨then_expr⟩ is returned). The types of ⟨then_expr⟩ and ⟨else_expr⟩ must be the same.

—example If actions—

```
if eq?(x, 1) then assign(y, 3)
                    (*returns data of type obj(ints)*)
if f(w) then 3 else 4
                    (*returns data of type ints*)
```

4) Whiledo:

⟨whiledo⟩ ::= whiledo(⟨loop_fcn⟩, ⟨exit_fcn⟩, ⟨start_action⟩)

Whiledo expressions support iteration without updates. The constructed action iteratively loops through applications of ⟨loop_fcn⟩ to an implicit "loop state" until the result returned is null, at which point ⟨exit_fcn⟩ is applied to the loop state, producing the final whiledo action result. The loop state is initially the result of ⟨start_action⟩, and thereafter is the nonnull result obtained from the previous application of ⟨loop_fcn⟩. In practice, the body of ⟨loop_fcn⟩ is an if expression without an else branch.

—example Whiledo actions—

```
whiledo(fun(x) if <?(x, 10) then + (x, 1),
          fun(x) - (x, 10),
          1)   (*returns 0*)
whiledo(fun(loop)
  if >?(loop.counter,1) then
    [counter : -(loop.counter,1),
    accum : *(loop.counter, loop.accum)]
  fun(exit) exit.accum,
  [counter : 5, accum : 1]) (returns 120, factorial of 5)
```

5) Filter:

⟨filter⟩ ::= filter(⟨function⟩, ⟨set_expr⟩+)

A filter action applies a function to each element of the Cartesian product of the sets given as arguments to the filter action constructor. The function must have as many parameters as there are sets to filter. The result of the filter action is a set made up of the application results. If the filter function returns an object, the filter result is an object set, otherwise the result is a value set. Filter actions are the backbone of many FAD database applications, and their optimization is essential when multiple sets are involved [29].

—example Filter actions—

```
filter(+
  {1, 2, 3}
  filter(fun(x) + (x, 1), {1, 2, 3}))   (*returns {3, 4, 5, 6, 7}*)

filter(fun(e)
     if <? (e.salary, 10000)
     then begin tupleassign(e,salary,+(e.salary, 1000)) end,
  db.employees) (*returns { } after side-effecting some employees*)
```

6) Pump:

⟨pump⟩ ::= pump(⟨unary-fcn⟩, ⟨binary-fcn⟩, ⟨identity⟩, ⟨set_expr⟩)

A pump action reduces a set to a single result by applying a unary function to each element of the set to produce an intermediate set (possibly with duplicates) which is then reduced to a single result using a binary function to combine the intermediate set members. The binary function is assumed to be both associative and commutative (with ⟨identity⟩ as the identity element) so that the reduction can be performed in any order, and in parallel. If the set to be pumped is empty, then the identity element for the binary function is returned.

—example Pump actions—

```
pump(fun(x) 1, +, 0, {5, 6, 7, 8})   (*returns 4—
                                        the set cardinality*)
pump(fun(e) if =?(e.age,30) then{e} else { }
  union,
  { }
  db.employees)   (*returns the set of employees of
                                        age 30*)
```

7) Group:

⟨group⟩ ::= group(⟨function⟩, ⟨set_expr⟩)

A group action creates a set of ordered tuples representing the equivalence classes of the argument set under application of the argument function. Within the set returned by group, each tuple is composed of two fields. The first field contains the result of applying the function to one of the set elements, and the second field contains the set of all argument set elements that are mapped to the datum in the first field by the argument function.

—example Group actions—

```
group(even?, {1, 2, 3, 4, 5})
  (*returns{[true, {2, 4}], [false, {1, 3, 5}]}*)
group(fun(x)x.1, {[1, 2], [1, 3], [2, 2]})
  (*returns{[1, {[1, 2], [1, 3]}], [2, {[2, 2]}]}*)
```

8) Escape:

⟨escape⟩ ::= escape(⟨int_expr⟩)

An escape action returns its argument as an action result status to the closest lexically enclosing action whose result status is named in a let expression, and returns null (of the appropriate type) as the result of this action. If there is no such lexically enclosing action, then the program itself is exited with a null result and a message based on the escape status.

—example Escape actions—

```
let y = "abc"
    x% s = if substring(y, "bcd")
             then append(y, "def") else escape(2)
in[x, s](*returns [null⟨strings⟩, 2]*)
```

9) Abort:

⟨abort⟩ ::= abort(⟨string_expr⟩)

A FAD program is considered a transaction by the database system, and multiple concurrent transactions are supported by Bubba although this is invisible to a FAD program. Upon program termination, any updates to the database performed by the program must be committed, and this is handled automatically by Bubba transaction management facilities. Because it is sometimes necessary to abort a transaction and avoid committing any database updates that have been performed, FAD provides an abort action, which causes an immediate exit from the program, aborts the transaction, and returns its string argument as the program result.

—example Abort actions—

abort(append("no members in", y))

IV. FAD Modules

A module (the unit of program compilation) provides a scope within which recursively defined types, recursively defined functions, and a transaction program can be declared.

```
⟨module⟩          ::= module⟨id⟩
                        ⟨dbname⟩
                        ⟨optional_transient_schema⟩
                        ⟨fcn-decl⟩*
                        ⟨program_def⟩
⟨dbname⟩          ::= db = ⟨id⟩
⟨transient_schema⟩ ::= schema⟨type-decl⟩*
⟨type-decl⟩       ::= ⟨id⟩ = ⟨type_expr⟩
⟨fcn-decl⟩        ::= define⟨id⟩fun(⟨params⟩)⟨action_expr⟩
⟨program_def⟩     ::= def prog(⟨params⟩)⟨action_expr⟩
```

A ⟨dbname⟩ is used to indicate the persistent schema that should be used when checking the module. A ⟨transient_schema⟩ enables a programmer to declare types that are useful in the function definitions that follow. Although the FAD compiler infers types, explicit type declarations can provide a convenient form of documentation in the form of assertions that are checked by the compiler. Two example modules are shown in Figs. 5 and 6.

When compiling the new-parts module in Fig. 5, the FAD compiler determines that the formal parameter p for add-part must have the same type as the tuples in the parts database relation, and therefore insures that the tuple constructed in the main transaction filter function (the tuple passed to add-part) is of this type. An equivalent new-parts module (expressed using types inferred by the FAD compiler is shown in Fig. 7 for comparison.

V. Design Objectives and Decisions

As mentioned in the Introduction, we wanted FAD to be a conceptually simple database programming language with declarative, set-oriented operations, support for general purpose computational mechanisms, and a uniform model of transient and persistent data including shared complex objects and null data. Further, we wanted this in a context of strong static typing, in order to assist reliable program development and execution efficiency. None of these objectives seemed particularly aggressive, and it would therefore be surprising if FAD were somehow a revolutionary language; this was not our purpose at all. In this section, we review our decisions with respect to 1) the overall character of the language, 2) its data model, and 3) our implementation of strong static typing. Section VI will compare our decisions with those reflected by related work.

A. Overall Language Character

By overall language character, we mean the general style of expressing individual operations, and the mechanisms used to combine individual operations into complete programs. A number of different possibilities are available: purely functional (no update operations), functional-style (expression-based), procedural, logical, and object-oriented. Also, we feel an important aspect influencing the character of a database programming language concerns the relationship between transient data and persistent data: Is there a division between these two that requires translation between different formats, the use of a special query subsystem, the use of load/store operations, or are these two classes of data uniformly presented [14] so that persistence is orthogonal to type, and based on reachability as opposed to declaration?

Because a primary goal was showing the feasibility of automatic, efficient utilization of a Bubba-style architecture for scalable support of general purpose database programs, an initial language focus concerned the need to program with declarative, set-oriented operations on a uniform model of transient and persistent data. It was felt that this style of programming would handle a wide variety of realistic applications, be simple enough to allow the necessary compile time analysis, and allow direct manipulation of persistent data by object code executing in a virtual single level store. Unfortunately, although filter, group and pump operations (or restrictions thereof) are the bread and butter operations of SQL, we were aware of no general purpose database programming languages that provided these operations within a uniform model of persistent and transient data. FAD was therefore developed to provide such a language in support our research objectives.

The initial designers of FAD [22] chose an expression-oriented, functional style of programming, similar to that suggested by Backus [27], because of its conceptual and syntactic simplicity. The declarative, set-oriented operations important to FAD fit nicely within this framework, which included conditionals and iteration as necessary for computational completeness. Unfortunately, although it was computationally complete, the initial version of FAD was difficult to use. The absence of data identifiers, and the required use of function composition as the sole program building mechanism were problematic.

In order to include data identifiers, support common subexpressions, and allow specification of sequentially sequenced actions, FAD was enhanced with ideas found in more traditional functional-style languages. In particular, function parameters and a let action constructor were added. Another enhancement was the idea of a module, to provide a scope for naming and referring to user-defined functions. Thus, although the current version of FAD retains an FP flavored whiledo (in which an internally maintained loop state supports iteration without requiring updates to objects), the overall style of programming supported by the language is now closer to that seen in environment-based functional-style languages such as Common Lisp [33].

Persistence in FAD is based on reachability from a single root, *db*, and is orthogonal to type. Bubba allows FAD programs to execute in parallel, as concurrent, multiprogrammed database transactions. This is essential for efficient utilization of the bubba architecture, but does not directly impact FAD (the abort statement is the only place where FAD recognizes the connection between programs and transactions). Bubba includes a specially designed virtual memory OS that supports a single level store (into which the persistent address space is mapped), page-level locking of the persistent space, and transaction-private address spaces for updated persistent data. At the close of a transaction, a distributed two-phase commit is used to incorporate updates into the persistent database [21].

B. The Data Model

The initial FAD data model included no values—only sharable, updatable, atomic and structured objects [22]. Early application experience with FAD, however, indicated a number of somewhat problematic aspects related to an absence of values in the data model. For example, FAD originally had "predicates" that returned true or false values as needed to control conditionals and iteration. But the results of predicates were not available to the user as data (because true and false were not objects). While this had not seemed restrictive with the original language, in which neither data nor functions could be named, enhancements to allow named data (which support

```
(*
  —parts-suppliers db—

  part     = [part-num: ints, part-name: strings, supplier: ints]
  supplier = [supplier-num: ints, parts: obj({ints})]
  db       = [parts: obj({part; key is part-num}),
              suppliers: obj({supplier; key is supplier-num})]
*)


module new-parts (*add new parts for an existing supplier*)
db = parts-suppliers
define add-part fun(p) (*add a new part*)
  do
    insert(db.parts,p),
    insert(db.suppliers@(p.supplier).parts, p.part-num)
  end
define prog(s-code, new-parts)
  filter(fun(p) add-part([part-num:p.1, part-name:p.2, supplier: s-code]),
    new-parts)
```

Fig. 5. Parts–suppliers example.

```
(*
  —ancestors db—
  db = [parents: obj({[person: strings, child: strings]})]
*)


module get-ancestors (*return the ancestors of a given person*)
db = ancestors
define ancestors fun(parents) (*start w. partents, and extend frontier*)
  whiledo(
    fun(loop) (*try to extend frontier*)
      let frontier = loop.frontier, accum = loop.accum
      in if non-empty?(frontier) then
        let new_front =
          filter(
            fun(parent, frontier)
              if =? (parent.child,frontier.ancestor)
              then [person:frontier.person
                    ancestor:parent.person],
            db.parents,frontier)
          in [frontier:new_front, accum:union(new_front,accum)]
        fun(exit) exit.accum, (*return accumulated ancestors*)
        [frontier:parents, accum:parents])
define prog(name)
  let parents =
    filter(
      fun(p) if =? (p.child, name)
        then [person:name, ancestor:p.person]
      db.parents)
  in ancestors(parents)
```

Fig. 6. Ancestors example.

factoring out common subexpressions in the interest of both readability and efficiency) suggested that the result of a test performed by a predicate might be named to allow its use more than once. A dual problem, highlighted by adding modules to support named functions (another factoring operation useful to program development), was that complex predicates built up from primitive comparisons and Boolean connectives could not be implemented as user defined functions (since functions should return data, and values were not FAD data).

Adding values to FAD offered a number of benefits. In addition to cleaning up problems like those mentioned above, supporting values strengthened the relationship between FAD and the value-based relational model (in which relations and tuples may be viewed as FAD objects, and attributes as FAD values [34]). In addition, the notions of identity, sharing, and updating do not apply to FAD values, so they can be implemented more efficiently than objects in an underlying database management system.

```
module new-parts (*showing types inferred by the FAD compiler*)
db = parts-suppliers
schema
T$100 = [T$101]
T$101 = [db.parts.@.part-no, db.parts.@.part-name]
define add-part fun(p⟨db.parts.@⟩)
  do
    object-sets.insert(db."parts",p)
    object-sets.insert(db."suppliers"@(p."supplier")."parts", p."part-no"
  end
define prog(s-code⟨db.parts.@.supplier⟩, new-parts⟨T$100⟩)
  filter(
    fun(p⟨T$100⟩)
      add-part([part-no: p.1,
        part-name: p.2
        supplier:s-code]⟨db.parts.@⟩),
    new-parts⟨T$100⟩)
```

Fig. 7. Inferred types.

The decision to add values to FAD required us to address two related questions: Should structured values be supported, and, if so, should structured values be allowed to contain objects? We chose to allow structured values because they seemed appropriate within the overall FAD context. As mentioned above, FP was an initial source of inspiration for the language, and FP operations (including whiledo) were specifically designed for handling structured values. Although our approach of supporting object updates prevented FAD from being a purely functional language like FP, adding structured values to FAD produced a computationally complete, purely functional sublanguage well-suited to expressing queries. This would not have been possible had FAD employed the more usual approach to iteration (which is based on testing data changed by updates), so we felt that we were simply making good use of the initial FAD language framework.

Another reason for supporting structured values had to do with duplicate elimination in sets of structured data. FAD sets do not contain duplicates, but objects are identical only if they have the same identity. Thus, for example, insert({new([1, 2])}, new([1, 2])) returns a set with two elements. Originally this was viewed as an asset for the language, since it was a way of handling multisets, but in our applications, data placed in sets were often more appropriately viewed without the concept of identity because duplicate elimination was desired. This problem could have been solved by providing different kinds of sets in FAD (perhaps, by parameterizing the set constructor with a comparison test), but the implications of such an approach for strong static typing seemed unclear. Instead, supporting structured values in addition to structured objects provided a simple solution to this problem while allowing us to retain and make good use of the traditional concept of sets as not containing duplicates. With this approach, the expression *insert({[1, 2]}, [1, 2])* returns a set with a single element, which was what many applications required.

The question of whether structured values could contain objects was a difficult one, because either approach can be rationalized. We were ultimately guided by the specific concept of values that we wanted to maintain for the programmer, the underlying database system, and portions of the compiler concerned with parallel execution: no observable portion or aspect of a value should ever change. Allowing updatable objects in structured values would allow values to change over time (i.e., a test performed on a value at one point during program execution might return a different result when performed on the "same" value at a later time). An example may help clarify this issue.

$$\begin{aligned}&\text{let } x = [1,\ \text{new}(1)]\\ &\quad y = +(x.1, x.2)\\ &\quad \text{assign}(x.2, 2)\\ &\quad z = +(x.1, x.2)\\ &\text{in eq?}(y, z)\end{aligned}$$

The fundamental guarantee we wanted FAD to offer concerning values was that y and z must be the same if x is a value (because values do not change over time, and both y and z depend only on x). But in this example y is 2 and z is 3. Therefore, in FAD (according to the simple view of values and objects we wanted to support), the data named by x in the above example is not a value, but rather an object—of type *obj([ints, obj(ints)])*. The fact that y and z differ is explained by the fact that portions of the object x were changed by an update in between the definitions of y and z.

Coining a term, we characterize our decision to support this programmer view as one of supporting an "identity-based" semantics, as opposed to the more traditional "reference-based" approach (in which pointers of one form or another explicitly appear as data). A distinguishing aspect of our approach is that object identifiers are not FAD data, and neither in general are object states.

C. Strong Static Typing

What we mean by strong static typing is that the data created and manipulated by a FAD program have the same type throughout its lifetime, and this type is known by the compiler as a result of static analysis of the program text and the persistent database schema.

The original FAD language was not strongly typed; operations such as tuple selection required run-time examination of

whatever fields happened to be stored in a tuple at the time of the select. If the requested field was not present, null was returned [22]. In addition to not being able to generate efficient code for structure access, run-time type errors were possible (e.g., a tuple select operation might be applied to a set during program execution). Because a major objective of our research was efficient use of the Bubba architecture, such an approach was deemed unacceptable. Strong static typing was therefore required to provide efficiency and prevent run-time type errors.

Initially, we had to decide whether the types of data used in a program would be explicitly declared, or automatically inferred by the compiler. We decided to attempt support for both approaches. We felt that explicitly declared types would be a useful documentation technique whose consistency could be checked by the compiler, and, although automatic inferencing was very attractive, we were initially unsure as to whether complete type inferencing was possible for a language such as FAD (it was clear that some degree of inferencing was possible).

After having designed the FAD data model, our first step towards supporting strong static typing was to develop the language framework necessary for explicit type declarations (automatic inferencing was viewed as a means of supplying declarations wherever they were omitted). Our solution combined the transient schema section of a module, the FAD type expression language presented in Fig. 3, and the use of type expressions within angle brackets following action expressions.

A central issue for type languages concerns the question of when two type expressions represent the same type. Two different approaches to this question are based on structural equivalence, and name equivalence. Because our approach to the FAD data model was based on considering types to be domains, and because we wanted to do unification-based type inferencing (which is based on structural equivalence), we chose structure-based equivalence as opposed to a name-based approach. Within this context, however, type references turned out to be extremely useful (if not essential), because they allowed us to refer to types defined in the persistent schema by name.

The type information associated with persistent data is held in a persistent schema, which, in addition to reflecting the "conceptual" types in the database, also includes concern for many of the physical level aspects related to storage of data within a database (e.g., ordering of fields in tuple structures is invisible to conceptual level tuple types—a fact that is useful as type constraints concerning the fields that are in a tuple are incrementally accumulated during inferencing). Although there are default physical level representations for conceptual level types, in general the same conceptual level type may be implemented in different ways within the Bubba data storage and access system. Generating efficient code for data manipulation requires that the physical representation be known.

This distinction between conceptual and physical level type information represented a subtle complication for type inferencing of FAD program modules. The result was that unification of types (as used in type inferencing) ultimately reflected the need to be aware of both conceptual and physical levels, giving precedence to persistent schema types. The explicitly typed parts-supplier example given in Fig. 7 illustrates this. None of the database programs we have written have required explicit type declarations, and these programs include fairly complicated decision support algorithms. Because these programs ultimately deal with the database, all the necessary type information is ultimately found in the persistent schema. Type inferencing thus provides a reliable and useful service for the FAD database programmer, even though it is conceivable that explicit type declarations might be required in some situations.

VI. Comparisons with Related Work

A. Overall Language Character

Aside from supporting persistence, one difference between FAD and more sophisticated, higher order functional-style languages (e.g., Scheme [35] and ML [36]) is that functions are not data in FAD. Although the FAD actions constructors accept functions as arguments, user-defined functions in FAD can neither accept functions as arguments or return functions as results.

Amber [37] is a higher order functional-style language with persistence. Persistence in Amber is somewhat orthogonal to type, and is not based on reachability. Explicit import and export statements are required to load and store persistent data from a file system, and the data imported and exported must have a "dynamic type" (which means that the data carries its type description with it [38]). Galileo [39] is a higher order functional-style language with persistence, based on ideas of ML extended with support for inheritance. Persistence in Galileo is orthogonal to type, and is based on reachability from a global identifier binding environment (to which new identifiers can be added using an explicit declaration statement). PS-Algol [40] is a procedural-style language with a persistent heap. This provides uniform support for transient and persistent data, as well as reachability-based persistence. PS-Algol appears to have been the first language to support reachability-based persistence and uniform transient/persistent data. It has been the focus of a sustained development effort. Like Amber and Galileo, PS-Algol is a higher order language with the ability to treat functions as data.

None of the above languages includes an operation equivalent in power to FAD's filter operation. This includes Machiavelli [41], a higher order functional-style language that incorporates typing ideas that seem promising for database programming. Although, as shown in [41], it is possible to write an expression in Machiavelli that produces the overall result of a filter operation, this expression makes use of very general combining forms and higher order functions—an approach that would currently result in disastrous performance penalties for an application.

This problem has been and continues to be a critical focus of research in compiler technology for advanced functional languages [42]–[46]. Although optimizing compiler technology may someday support such an approach (so that, for instance, a complete Cartesian product of filtered sets will not

```
(*FAD Version*)
filter(fun(customer, order, item-ordered)
      if and?(eq?(customer.name, "John"),
         eq?(customer.code, order.ccode),
         eq?(order.ocode, item-ordered.ocode))
      then assign(item-ordered.qty, 5),
   db.customers, db.orders, db.items-ordered)

(*SQL Version)
update items-ordered
  set qty = 5
  where ocode in
  (  select ocode
     from customer c, order o
     where c.code = o.,ccode and c.name = "John")
```

Fig. 8. FAD filter versus SQL select and update.

be created even though the function implementing the filter is expressed this way), this is not currently possible. And, as we have pointed out, optimization of FAD's filter operation is absolutely essential for real applications.

The approach suggested by Machiavelli is similar to others that have been proposed [47]–[50], which finesse optimization issues raised by the use of extremely general operations and combining forms to support important database operations. In contrast, the FAD implementation of filter is specialized to its particular (though widely applicable) job—it does not necessarily create or even visit the Cartesian product of its argument sets, and uses proven technology developed for distributed relational systems to minimize the size of any intermediate sets that are required [29]. Given current compiler technology, we believe that FAD represents a realistic design tradeoff between the desire for powerful, declarative operations, and the critical need to automatically optimize programs expressed in terms of these operations.

Although FAD's filter operation is specialized to handle queries similar to those supported by SQL's select operation, it is more expressive. To illustrate this, Fig. 8 compares FAD with SQL by presenting FAD and SQL programs that update a single relation. Although both program segments do the same thing, the SQL version is handicapped by the fact that SQL's update operation can only process one relation at a time. This results in an SQL program segment that is difficult to understand (in comparison with the FAD version).

B. The Data Model

The primary comparison of FAD with other languages that we want to make here concerns the use of reference-objects in database programming languages such as Galileo [39] and Machiavelli [41]. These languages may be understood as having a single kind of updatable object (the reference-object), and an unstratified data model (i.e., there is no value/object dichotomy). Because pointer assignment is the only update operation in these languages, updates to data structures such as sets and tuples cannot be directly expressed. Although this approach may be acceptable for modeling purposes, it does not seem appropriate in a realistic database programming language.

Reference-objects originated with ML [36]. They hold an updatable pointer to another object. For example, an *int ref* is an object that holds an updatable reference to an integer; a *string ref* holds an updatable reference to a string. The reference object approach requires that pointers be data in the language; an expression whose result is assigned to a reference-object must be understood to return a pointer (because reference objects hold pointers, and because it is necessary to explicitly dereference such pointers). An update to a data structure pointed to by a reference-object is modeled by first constructing a new structure, often incrementally different from the original, and then updating the reference-object pointer so it points to the newly created structure.

We provide in Fig. 9 an example to illustrate the difference between FAD and reference-object languages. Two code segments are illustrated—the first uses updatable, sharable objects as provided by the FAD data model; the second uses FAD-like syntax and an unstratified data model containing updatable, sharable reference-objects. We begin by constructing a set containing a single tuple representing a person named Jack. Type declarations are omitted, but these would indicate that the name field is a key for this set. We then insert another tuple into the set, and then update Jack's age, so that any succeeding statements using x will see a set containing two tuples and Jack's changed age. The assumption within this example is that the set should be an updatable, sharable object (so that new tuples can be incrementally added to it, and others sharing the set will see added tuples), and the tuples representing people should be updatable, sharable objects (so that an age field can be changed, and other database structures sharing the tuple will see the changed age field).

In the Fig. 9 code segment that uses reference-objects, the operation semantics are as follows. Ref is somewhat analogous to new in FAD—it creates a new object (in this case, a reference-object pointing to data). The dereference operation ! is used to traverse the pointer stored in a reference-object. Assign is the only update operation—it replaces the pointer stored in a reference-object with the address denoted by its second argument. Insert creates a new set, incrementally different from the original set, and returns a pointer to the new set. Tupleassign performs in a similar fashion for tuples. The outermost ref used in the set creation is necessary because the data model is unstratified.

Fig. 10 provides an illustration of the data structures that exist during execution of the two different program segments of Fig. 9. An implementation level combination of pointer/structure used for sharing and seen integrally by the programmer is illustrated with a dashed arrow (rather than being surrounded with dashed lines as in Fig. 2); reference objects and pointers visible to the programmer are shown using circles and solid arrows, respectively. Note that in the reference-object illustration, after the second update operation, there are two unreachable data items: the original set structure, which still holds a single (but not the original!) tuple, and the original tuple. These two structures are now garbage that must be collected.

It is unclear what advantage reference-objects provide, and the comparisons offered by Fig. 10 and the different program

```
(*using updatable objects as in FAD*)
let   people = { new([name: "jack", age: 8]) }
      insert(people, new([name: "jill", age:7]))
      tupleassign(people@"jack", age, 9)
in . . .

(*using reference-ojbects as in ML*)
let   people = ref({ref([name:"jack", age:8])})
      assign(people, insert(!people,ref([name:"jill", age:7]))
      assign((!people)@"jack", tupleassign(!((!people)@"jack"), age, 9)
in . . .
```

Fig. 9. Updatable objects versus reference objects.

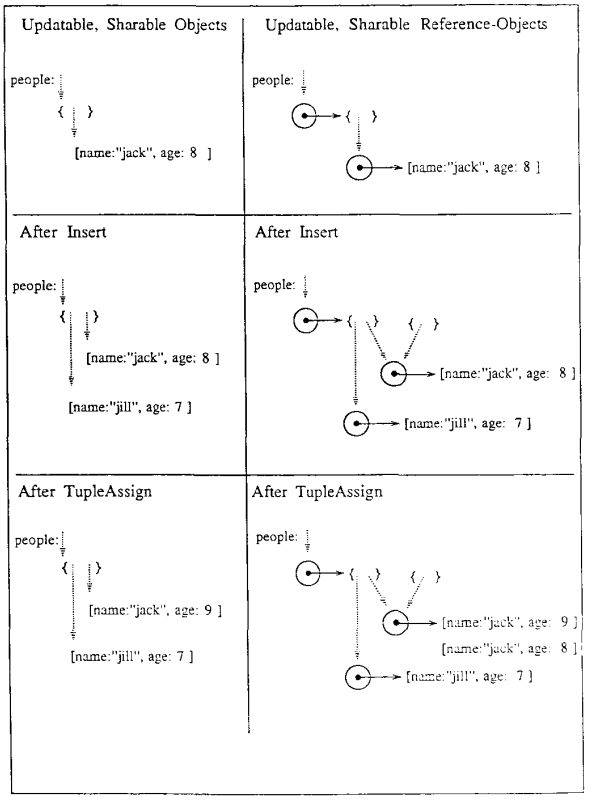


Fig. 10. Data structures for Fig. 9.

segments in Fig. 9 show that programming with updatable reference-objects as opposed to updatable structured objects adds a significant amount of programming complexity. Simply compare the two different versions in Fig. 9 of the final update that replaces the age field of Jack's tuple (each use of ! in Fig. 9 represents a traversal of a solid arrow in Fig. 10).

Our belief is that reference-objects were added to ML because they were a minimal extension to support side-effect based programming in what was otherwise a purely functional language. But ML was not intended as a database programming language (it was originally the meta language used for declaratively representing proof strategies in a theorem proving system [51]). In contrast, FAD started out with updatable structured objects because this was seen as being appropriate for database applications. Given objects as in FAD, there is no need for reference-objects. Perhaps because ML is an elegant and well-received functional-style language, other functional-style languages oriented towards database programming have borrowed ML's approach to updates. An essential difference between these languages and FAD is therefore that they support an unstratified reference-based data model with a single kind of updatable, sharable object containing an explicit pointer, whereas FAD supports a stratified identity-based data model with structured objects, none of which contain explicit pointers.

FAD's data model is unique in that it provides Values and Objects (as defined in Fig. 1) within a single language; other languages provide one or the other, or an unstratified mixture of the two (if both structured values and structured objects are supported). For example, purely functional languages provide Values. The original version of FAD provided Objects. Amber [37] and O2 [18] provide atomic values and structured objects. The unstratified EXTRA data model [52] supports both structured values and objects through use of the own, own ref, and ref type constructors.

In comparison with other data models, stratification represents a new approach to integrating structured values and objects. It avoids situations like that seen above in the reference-object example (in which the original set of people, not updated by the set insertion, ultimately ends up with a different tuple inside it), and it avoids referential integrity problems such as those of the EXTRA data model, in which own ref objects may be deleted even though they are referenced from other objects' ref attributes.

C. Strong Static Typing

1) Type Inferencing: Unification-based type inferencing was invented by Milner for use in ML [36], [53]. In this approach, unification of type expressions is used to incrementally solve the system of type equations corresponding to a given program. The free variables of such an equation system correspond to the unknown types within a program, and the type equations themselves arise from two sources: First, there are language-specific semantic axioms describing the types of data required and produced by the primitive operations of the language; and second, as specific actions as composed within a program, the output types of one action must be equated with the input types of another. The set of typing axioms for a language may be considered to define an inference system within which one

can prove that an expression in that language has a given type. Languages with different primitive operations have different sets of axioms.

The importance of unification is that it provides a mechanism for monotonically reflecting within type variables the most specific implications of type constraints. In particular, because unification of two type expressions produces a most general solution for the type variables in the expressions, constraints can be processed incrementally, as they are encountered, without fear of overconstraining the overall system and preventing a global solution if there is one. Thus, the essential capability required for this style of type inference is not unification per se, but rather the ability to monotonically refine the values of type variables (from an initial value of "unknown") as necessary to reflect the most specific implications of type constraints as they are encountered. Unification simply provides a mechanism for accomplishing this when type variables range over a domain of uninterpreted terms.

In ML, the terms used as a domain for type variables during inferencing are exactly those that are provided by the ML type expression language. For example, an expression such as f(3) in ML imposes the following constraint on f: it must be a function that maps an integer to some unknown type (we do not know what type this is, given only this context). The most general type for f can be represented using a type variable initialized to "unknown" (say, α) in an ML type expression: int-> α. Later in the program, f might be placed in a list whose elements are of type int-> bool-> β (where β is also an unknown type). At this point, unifying the type of f with the list element type (because lists are homogeneous in ML) would refine the value of α from unknown to bool-> β (another ML type expression).

In contrast, the terms used to represent FAD types are not sufficiently expressive as a domain for type variables during inferencing of FAD programs. This difference is a result of FAD's data domain, which includes a richer variety of types than found in ML (e.g., FAD includes keyed sets and tuples). An expression such as S@3 in FAD imposes the following constraints on S: it must be a keyed set containing elements of some type, and this type (whatever it is) must have ints as a component (because an integer is used as the key value). Now, it is possible to use the FAD type expression language to represent a variety of types that satisfy this constraint (e.g., *{[a:ints, b: ints]; keys is a}, {[a: ints, b: ints]; key is b}*, etc.), but all of these types make assumptions about S that are not specifically implied by the above constraints—it is simply not possible to express the most general type of S through use of FAD type expression containing type variables.

There are two approaches we could have taken to address this: 1) define a new term language for FAD types sufficiently expressive to represent the most general type satisfying any constraints that might be encountered during inferencing, or 2) implement an internal representation for types in which the specific constraints that might be placed on a type can be represented. We chose the second of these two approaches, since, in any case, it is an internal representation of type expressions for which unification is required during inferencing. Constraints that can be placed on FAD types during inferencing include (in addition to those appropriate in the above example) such requirements as the fact that a type may (or may not) be an object, that a tuple type must include a field with a particular label, that all fields in a tuple must be values, or that a particular tuple may (or does not) have additional fields not yet discovered. The "unification" procedure for FAD types takes all of these things into account.

The point of this comparison is to indicate that ML is a language with few combining forms and a simple data model, and because of this its type expression language is always able to express the most general type satisfying constraints that are encountered when processing arbitrary ML expressions. This is not something that should be expected in general. This point is reflected in recent papers that address extending the unification-based inferencing technique originally introduced with ML to languages with richer data models [41], [54]–[58]. Few database programming languages support type inferencing. Aside from FAD, we are only aware of Amber [37] and Machiavelli [41], neither of which include the range of data types found in FAD.

2) Polymorphism: Another useful point of comparison in this context is the fact that ML is a polymorphic language, whereas FAD is not. What this means in the context of unification-based type inferencing is that in ML, unbound type variables are allowed to remain after all constraints have been reflected in type variables. The understanding is that type variables are universally quantified at an outermost scope. For example, a function with the type (α list)->int is understood to return an integer when it is passed a list, no matter what type of elements are in the list. An example of such a function is length, a function that returns the number of elements in a list.

In contrast, the FAD analyzer considers type variables that are unbound (after all constraints have been satisfied) to reflect an ambiguity in the program. Thus, although unification-based type checking directly supports inference of polymorphic types for functions, we chose not to make use of this capability. We took this approach because it was unclear to us how to generate efficient code for FAD that was actually polymorphic (i.e., the same code would really work on different types of arguments), and efficiency was our ultimate objective.

The reason why ML code that deals with lists may be polymorphic is that the compiler can generate the same code to get the next element of a list no matter what kind of a list is involved. We did not want to assume that the same code is used to access a tuple field with a given label, no matter what type of a tuple is involved. In fact, our objective was exactly the opposite—we wanted to generate code that directly accessed structure elements based on the type of the structure, and this code is different for different tuple types even if these types may have fields in common (the fields are generally at different offsets). Supporting polymorphism for tuple selections can be done by interpreting the tuple select operation based on run-time examination of tuple fields (as was done in the original untyped version of FAD, which was, of course, polymorphic, but also unsafe), or by using an object-oriented approach at the implementation level. Both of these approaches involve overhead we wanted to avoid. Given a willingness to accept the implications of polymorphic tuple selections, however, strong

static type inferencing for such a language is straightforward, as demonstrated by Machiavelli [41].

VII. Conclusions

Based on the comparisons given above, and our use of FAD for a number of realistic database applications, we have come to the following conclusions. The basic functional-style used by FAD is similar to a number of other database programming languages that have been proposed. Some of the operations FAD provides within this context are unique to FAD (e.g., the filter action constructor, and element selections from keyed sets), and these have been very useful to us in our application experiments. Null data have been integrated into FAD through the combined interaction of many of its facilities (e.g., structure insertions and selections, and practically all of the action constructors), as well as being made uniformly available for use as application data. As far as we know, FAD is the first statically typed database language to do this to such an extent. Because special cases are handled uniformly, our approach to null greatly simplified the control of distributed process threads executing (for example) multiple copies of the same filter operation on the different portions of a persistent set distributed over multiple processing nodes [32].

We believe that FAD has achieved its objectives with respect to balancing user-level simplicity with generality, and feel that type inferencing is an important factor in this balance. It contributes to the ease of use of the language, while detecting many conceptual errors before program execution. Although query languages such as SQL, or those based on an extended relational calculus (e.g., EXCESS [52]) can offer greater simplicity for some queries, they are not general purpose programming languages.

Given our objective of high performance, we believe we have achieved a realistic balance between the desire for powerful set-oriented database operations and the restrictions imposed by existing compiler technology. FAD represents a step upwards from SQL, but, with further progress in compiler technology, it may be possible to support fewer special purpose operations (e.g., define or otherwise support FAD's n-ary filter in terms of lower level primitives), and rely on increasingly sophisticated optimization techniques to achieve execution efficiency.

We believe that FAD successfully integrates objects and values while supporting our objectives, and believe the resulting data model suggests an appropriate design space for choices here. We see no benefits in the reference-object approach, and view the absence of explicit pointers in FAD as an asset. Many of our applications have used structured values to good effect, although the persistent space has generally required only atomic values and structured objects. Transient structured values have been important to providing efficient execution of FAD on Bubba, in which values can be efficiently sent between nodes without loss of information, and we suspect it makes good sense to directly reflect updatability within types on any system whose architecture is expected to be truly parallel. Atomic objects have not been required by our applications, but they still make good sense from a data modeling standpoint. FAD does not support user-defined abstract data types (this planned feature was not implemented due to time pressure), but ADT instances should be treated as atomic data in a data model, and would be considered atomic objects if they can change over time.

A. Current Implementation Status

The FAD compiler [32] is currently operational, and comprises four phases: semantic analysis (static type checking and inference [28]); optimization for distributed data access based on an architectural performance model (in support of n-ary filter operations [29]); parallelization (creation of separate code bodies for distributed execution based on an overall dataflow model of process communication [30]), and low-level code generation [32].

The ultimate result of FAD compilation is a load module appropriate for execution on a prototype database system implemented on a Flex-32 with 40 processor nodes [21]. Each node has a dedicated (unshared) memory and a CDC Wren disk drive. Persistent sets are declustered (partitioned and distributed) over multiple nodes to increase performance [31]. Application performance measurements from the working prototype, and simulation results (necessary for performance projection of larger configurations) are reported in [21].

The Bubba project has been completed, but the investment in people represented by the project, and the lessons learned during the course of the project, will hopefully be reflected in cost-effective, scalable, and easy to use commercial systems of the future.

Acknowledgment

Primary acknowledgment for FAD is due to F. Bancilhon, who helped shape the original FAD semantics, and to the original developers of FAD, including S. Khoshafian, B. Hart, and T. Briggs. The project as a whole benefited from the management and overall direction of H. Boral. Bubba was developed by the MCC Systems Technology laboratory, headed by G. Lowenthal, and supported by MCC corporate shareholders.

References

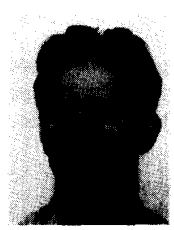
[1] E. Codd, "A relational model of data for large shared data banks," *Commun. ACM,* vol. 13, no. 6, June 1970.
[2] G. Jaeschke and H. Schek, "Remarks on the algebra of non first normal form relations," in *Proc. 1st ACM Int. Symp. Principles Database Syst.,* 1982.
[3] C. Zaniolo, "The representation and deductive retrieval of complex objects," in *Proc. 11th Int. Conf. Very Large Databases,* 1985.
[4] H. Schek and M. Scholl, "The relational model with relational valued attributes," *Inform. Syst.,* vol. 11, no. 2, 1986.
[5] M. Ozsoyoglu and L. Yuan, "A normal form for nested relations," in *Proc. 4th ACM Int. Symp. Principles Database Syst.,* 1985.
[6] S. Abiteboul and N. Bidoit, "An algebra for non normalized relations," in *Proc. 3rd ACM Int. Symp. Principles Database Syst.,* 1984.
[7] R. Hull and C. Yap, "The format model: A theory of database orgainization," *J. ACM,* vol. 31, no. 3, July 1984.
[8] A. Furtado and L. Kerschberg, "An algebra of quotient relations," in *Proc. ACM Int. SIGMOD Conf.,* 1977.
[9] M. Roth, H. Korth, and A. Silberschatz, "Theory of non-first-normal-form relational databases," Dep. Comput. Sci., TR-84-36, Univ. of Texas at Austin, 1984.
[10] S. Thomas, "A non-first-normal-form relational database model," Ph.D. dissertation, Vanderbilt Univ., 1982.

[11] F. Bancilhon and S. Khoshafian, "A calculus for complex objects," in *Proc. ACM Int. Symp. Principles Database Syst.,* 1986.
[12] G. Copeland and D. Maier, "Making Smalltalk a database system," in *Proc. ACM Int. SIGMOD Conf.,* 1984.
[13] J. Schmidt, "Some highlevel language constructs for data of type relation," *ACM Tran. Database Syst.,* vol. 2, no. 3, Sept. 1977.
[14] M. Atkinson and P. Buneman, "Types and persistence in database programming languages," *ACM Comput. Surveys,* vol. 19, no. 2, June 1987.
[15] S. Khoshafian and G. Copeland, "Object identity," in *Proc. 1st Int. Workshop Object Oriented Programming Syst., Languages, and Appl.,* Portland, 1986.
[16] G. Kuper and M. Vardi, "On the expressive power of the logic data model," in *Proc. ACM Int. SIGMOD Conf.,* 1985.
[17] S. Tsur and C. Zaniolo, "An implementation of GEM—Supporting a semantic model on a relational back end," in *Proc. ACM Int. SIGMOD Conf.,* 1984.
[18] C. Lecluse, P. Richard, and F. Velez, "O2, An object-oriented data model," in *Proc. ACM Int. SIGMOD Conf.,* 1988.
[19] P. Chen, "The Entity-Relationship model—Toward a unified view of data," *ACM Trans. Database Syst.,* vol. 1, no. 1, Jan. 1976.
[20] M. Hammer and D. McLeod, "Database description with SDM: A semantic database model," *ACM Trans. Database Syst.,* vol. 6, no. 3, Mar. 1981.
[21] Bubba Team, "Protyping Bubba, A highly parallel database system," *IEEE Trans. Knowledge Data Eng.,* vol. 2, Mar. 1990.
[22] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, A Powerful and simple database language," in *Proc. Int. Conf. Very Large Databases,* Brighton, 1987.
[23] H. Boral, "Parallelism in Bubba," in *Proc. Int. Symp. Databases in Parallel and Distributed Syst.,* Austin, Dec. 1988.
[24] M. Stonebraker, "The case for shared-nothing," *Database Eng.,* vol. 9, no. 6, June 1986.
[25] D. DeWitt *et al.,* "GAMMA—A high performance dataflow database machine," in *Proc. Int. Conf. Very Large Databases,* Tokyo, 1986.
[26] S. Tsur and C. Zaniolo, "LDL: A logic-based data language," in *Proc. 12th Int. Conf. Very Large Databases,* 1986.
[27] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM,* vol. 21, no. 8, Aug. 1978.
[28] S. Danforth, "Typechecking FAD, A database programming language," MCC Tech. Rep. ACA-ST-194-88(Q), June 1988.
[29] P. Valduriez and S. Danforth, "Query optimization for database programming languages," in *Proc. 1st Int. Conf. Deductive and Object Oriented Database Syst.,* Kyoto, Dec. 1989.
[30] B. Hart, S. Danforth, and P. Valduriez, "Parallelizing FAD, A database programming language," in *Proc. Int. Symp. Databases in Parallel and Distributed Syst.,* Austin, Dec. 1988.
[31] G. Copeland, B. Alexander, and E. Boughter, "Data placement in Bubba," in *Proc. ACM Int. SIGMOD Conf.,* Chicago, IL, 1988.
[32] P. Valduriez, S. Danforth, B. Hart, T. Briggs, and M. Cochinwala, "Compiling FAD, A database programming language," in *Proc. 2nd Int. Workshop Database Programming languages,* Salishan, June 1989.
[33] G. Steele, *The COMMON LISP Reference Manual.* Bedford, MA: Digital, 1984.
[34] P. Valduriez and S. Danforth, "Functional SQL (FSQL), An SQL upward compatible database programming language," MCC Rep. ACA-ST-045-89, 1989, to be published in *Inform. Sci.—An International Journal.*
[35] G. Sussmann and G. Steele, "An interpreter for extended lambda calculus," MIT AI Memo 349, Dec. 1975.
[36] R. Milner, "A proposal for standard ML," Internal Rep. CSR-157-83, Dep. Comput. Sci., Univ. of Edinburgh, 1984.
[37] L. Cardelli, "Amber," AT&T Bell Labs Tech. Memo 11271-840924-10TM, 1984.
[38] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," DEC SRC Rep. 47, 1989.
[39] A. Albano, L. Cardelli, and R. Orini, "Galileo: A strongly-typed, interactive conceptual language," *ACM Trans. Database Syst.,* vol. 10, no. 2, June 1985.
[40] M. Atkinson, K. Chisholm, and W. Cockshott, "PS-Algol: An Algol with a persistent heap," *ACM SIGPLAN Notices,* vol. 17, no. 7, July 1981.
[41] A. Ohori, P. Buneman, and V. Breazu-Tannen, "Database programming in Machiavelli," in *Proc. ACM Int. SIGMOD Conf.,* Portland, 1989.
[42] G. Steele, "Rabbit: A compiler for scheme," MIT Rep. AI-TR-474, 1978.
[43] F. Bellegarde, "Rewriting systems on FP expressions that reduce the number of sequences they yield," in *Proc. ACM Symp Lisp and Functional Programming,* Austin, 1984.
[44] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "ORBIT: An optimizing compiler for scheme," in *Proc. ACM Symp. Compiler Construction, SIGPLAN Notices,* vol. 21, no. 7, July 1986.
[45] A. Ferguson and P. Wadler, "When will deforestation stop?" in *Proc. Glasglow Workshop Functional Programming,* 1988.
[46] K. Gopinath and J. Hennessy, "Copy elimination in functional languages," in *Proc. ACM Symp. Principles Programming Languages,* 1989.
[47] P. Buneman, R. Nikhil, and R. Frankel, "A practical functional programming system for databases," in *Proc. ACM Conf. Functional Programming Languages and Comput. Architecture,* 1981.
[48] R. Nikhil, "Semantics of update in FDBPL," in *Proc. Int. Workshop Database Programming Languages,* Roscoff, 1987.
[49] P. Trindler and P. Wadler, "List comprehensions and the relational calculus," in *Proc. 1988 Glasglow Workshop Functional Programming,* 1988.
[50] A. Poulovassilis, "FDL: An integration of the functional data model and the functional computational model," in *Proc. 6th British Nat. Conf. Databases (BNCOD 6),* 1988.
[51] M. Gordon, R. Milner, and C. Wadsworth, *Edingurgh LCF (A Logic of Computable Functions), LNCS 78.* New York: Springer-Verlag, 1979.
[52] M. Carey, D. DeWitt, and S. Vandenberg, "A data model and query language for EXODUS," Rep. 734, Dep. Comput. Sci., Univ. of Wisconsin, 1987.
[53] R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.,* vol. 17, 1978.
[54] M. Wand, "Complete type inference for simple objects," in *Proc. Second Annu. Symp. Logic in Comput. Sci.,* 1987.
[55] Y.-C. Fuh and P. Mishra, "Type inference and subtypes," in *Proc. ESOP, '88,* LNCS 300. New York: Springer-Verlag, 1988.
[56] L. Jategaonkar and J. Mitchell, "ML with extended pattern matching and subtypes," in *Proc. ACM Conf. Lisp and Functional Languages,* Utah, 1988.
[57] M. Remy, "Typechecking records and variants in a natural extension of ML," in *Proc. ACM Symp. Principles Programming Languages,* 1989.
[58] J. Gaver, Type-Checking and Type-Inference for Object-Oriented Programming Languages, Ph.D. dissertation, Rep. UIUCDCS-R-89-1539, Univ. of Illinois at Urbana-Champaign, 1989.

Scott Danforth received the Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 1983, where he assisted in the design of a cellular architecture for parallel execution of functional languages.

He is currently employed by IBM in Austin, TX, where he is concerned with the theory and developing technology of object-oriented systems. From 1984 to 1990, he was a senior research scientist at MCC, where he developed parallel execution models for integrated functional and logical programming languages, and was responsible for defining and implementing FAD, a strongly typed functional-style language for programming Bubba, a parallel database system developed at MCC. He has published over 20 papers and technical reports, and is on the editorial board of the *International Journal of Parallel Programming.* His interest areas include parallel processing, programming languages, and compiler technology.

Dr. Danforth is a member of the Association for Computing Machinery.



Patrick Valduriez received the Ph.D. degree in computer science from the University of Paris in 1981.

He is currently a Director of Research at INRIA, the national research center for computer science in France. There he heads a project on advanced database technology including distributed, parallel, deductive, and object-oriented database systems. From 1985 to 1989, he was a senior scientist at MCC, Austin, TX. There he participated in the design and implementation of the Bubba parallel database system, managing the design and development of the FAD database programming language and its compiler/optimizer. He is the author or coauthor of over 60 technical papers and several books on various aspects of database systems, among which are *Relational Databases and Knowledgebases* and *Analysis and Comparison of Relational Database Systems* (Reading, MA: Addison-Wesley, 1990), and *Principles of Distributed Data Systems* (Englewood Cliffs, NJ: Prentice-Hall, 1991).

Dr. Valduriez is a member of the Association for Computing Machinery.