LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Functional Databases, Functional Languages

Computation Structures Group Memo 259 (Revised)

November 14, 1985
Revised July 15, 1987

Rishiyur S. Nikhil

To appear in Proc. 1985 Workshop on Persistence and Data Types, Appin, Scotland.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Functional Databases, Functional Languages

Rishiyur S. Nikhil

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139, USA

Abstract

Database systems today have evolved a great deal from the first storage structures, towards greater data independence, expressive power in manipulation languages, and expressive power in data models. But their facilities are still poor substitutes for analogues in programming languages: data abstraction, structured control constructs and type systems respectively. Most programming languages, on the other hand, deal inadequately (if at all) with the question of long-lived structured data. The problem is compounded when dealing with both a database system and a programming language that are alien to each other (as is common today).

Functional programming languages with functional databases offer a clean solution to this problem. Functional programming languages are expressive, do not have destructive updates, and allow much parallelism. Functional databases are never destructively updated— rather, one views them as infinite sequences of versions.

After outlining the features of such a system, we discuss its many advantages pertaining to types, optimization, expressive power, and interactive environments.

1 What is Wrong with Databases Today?

Today's database systems evolved out of mass-storage device architectures. This evolution has moved away steadily from device-specific models to more and more abstract models— for example, relational databases. Still, they are so difficult to use that many applications do so only when efficient and reliable access to large amounts of data is of over-riding

This research was done at the MIT Laboratory for Computer Science. Funding for this project was provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-1025.

concern. The designers of hundreds of everyday applications (mail systems, software engineering aids, word processors, ...) prefer *not* to deal with the complexities of current database systems and use primitive file systems instead.

Let us examine some of the characteristics that make current database systems so difficult and inconvenient to use.

1.1 Inadequate Type Structure

To a first approximation, the term Data Model used in databases is analogous to the term Type System in programming languages, *i.e.*, it specifies a universe of definable types. A schema for a database is analogous to a particular type definition within that type system, and a database itself is analogous to a particular value (data structure) of that type.

The type systems of most database systems are extremely poor in comparison to those available in modern programming languages. For example, CODASYL databases [13] provide scalar types, tuples (*i.e.*, records) of scalars, sequences of records and record-sequence pairs (owner and members of a "set"). In addition, there are various means to share records amongst different sets. Relational databases provide scalar types, tuples of scalars, and sets of tuples (*i.e.*, relations).

Unfortunately, these structuring mechanisms are not suitable for many applications—the encoding of data from the problem domain into (say) relations can be extremely awkward, and is, thus, error-prone. For example, circuit diagrams, graphs, IC layouts, matrices, *etc.* while not impossible to represent relationally,[1] stretch the model so much as to lose the progammer's intuition. Forcing the use of *only* relations greatly increases the programmer's intellectual burden, and is not unlike trying to encode complicated structures such as graphs and trees into FORTRAN arrays.

Some of the newer so-called Semantic Data Models attempt to remedy this shortcoming [10]. But they are still poor substitutes for the type facilities of modern programming languages. Many of them permit the description only of passive collections of data—they do not integrate data-manipulation *procedures* as part of the schema for a particular database. Instead, there is typically only a single, pre-determined generic set of database operators. This separation leads to two problems. First, it does not allow data abstraction, about which we will say more in the next section. Second, it makes it difficult to mix intensionally- and extensionally-defined data smoothly.

A serious shortcoming of most proposed data models is that they are rarely accompanied by rigorous semantics. Even in the relational model (one of the cleanest) there are many unclear areas. For example, consider joining a PERSON-AGE-FATHER relation with itself to get a grandfather relation. To avoid producing a meaningless relation with two columns with the same heading AGE, one needs special constructs such as *renaming*, whose semantics are often unclear.

[1] See, for example, [15].

1.2 Lack of Data Abstraction

The lack of data abstraction facilities is another serious problem in most data models. In databases one of the goals has long been to achieve "data independence" [13], *i.e.*, the independence of application programs from the particular access methods and storage structures used to implement a conceptual schema. Current database systems achieve this to varying degrees.

This concept is well established in programming languages. It is in fact no more than the "representation independence" of abstract data types— it is exactly what makes these types abstract. It is well known how to achieve this— each abstract type is accompanied by a set of procedures to manipulate objects of that type, and there are linguistic mechanisms to ensure those procedures are the *only* way to manipulate such objects. If the representation of an abstract type is changed, one need only change the implementations of the associated procedures; all client applications will be unaware of and unaffected by the change.

It is often necessary to maintain a notion of consistency in a database beyond what it is expressible in the type system (for example, that the standard deviation of machine-part tolerances should not exceed a certain number). Updates that result in inconsistent states of the database should not be permitted. Definitions of consistency are usually application-specific and may be arbitrarily complex. Many data models rely on a separate language of integrity constraints to express consistency requirements. There are several problems with this approach. First, it is yet another language for the programmer. Second, because the constraint language is separate from the data manipulation language, it may be difficult to check constraints efficiently (*e.g.*, during the update). Third, the programmer is not left with much flexibility in error-handling— *e.g.*, to trap violations, identify reasons, and take corrective action. Thus, languages for integrity constraints are often deliberately simplistic, incapable of expressing complex consistency requirements.

In programming languages, one again uses abstract data types to maintain integrity constraints (also known as *invariants*). Abstract data types ensure that data of a particular type can be manipulated only by authorized procedures. These procedures are specified by the programmer who has the full power of the programming language available to express integrity checks as complicated as the application demands, with whatever efficiency is possible, and with appropriate error-handling. The programmer may choose to maintain the standard deviation of machine-part tolerances internal to the machine-part abstract type, recomputing it incrementally on each update. This is much more efficient than recomputing it after each update.

Alerters are another useful feature of databases. Like integrity constraints, an alerter monitors the database for some condition, except that these are not necessarily violations of consistency— *e.g.*, large monetary withdrawal, fuel-level low, *etc.* On such conditions, an alerter performs an associated action, such as sending a warning message to a responsible person. Again, it is much better to incorporate alerters into the procedure definitions in an abstract data type rather than use a separate alerter language and a separate imple-

mentation technique.

1.3 Lack of Integration with Programming Languages

Most databases can be queried and updated in two ways: using a stand-alone query/update language (*e.g.*, SQL, QUEL), or by embedding a database sublanguage in an existing programming language (*e.g.*, EQUEL in C, SQL in PL/1) [13].

Stand-alone database languages are generally very convenient, but are typically limited in expressive power, lacking general control structures, procedural abstraction, full arithmetic, *etc.*

There is usually a semantic mismatch between database sublanguages and the programming languages they are embedded in, making them notoriously difficult to use. Typically, the type structures and the control structures in the programming language are quite different from those of the database sublanguage. For example, the database may be relational, but there may be no types or iteration constructs in the programming language that can naturally model relations and relational operations.

The methods to circumvent this mismatch are usually cumbersome and unsafe. For example a procedure may send a message to be interpreted by the database system, which leaves a result in some invisible work area. To examine this result, it may be necessary to establish a "cursor" on it, which must then be moved in small sequential increments revealing a small piece of the result at a time. Further, this kind of interface may relinquish any type-checking normally provided in programming language. The imposition of unnatural control structures to traverse database objects can negate any data-independence that the data model originally provides. The programmer must now manage two dissimilar models of store— that of the programming language, and that of the database system.

2 What Database Features do Programming Languages Lack?

We have claimed that many database concepts have better analogues in programming languages. What then are the particular characteristics of databases that programming languages lack?

1. Databases are not *ephemeral*, *i.e.*, they are "permanent" data-structures. In contrast, most data-structures in programs have a lifetime no longer than that of the program itself.

2. Databases provide efficient access to data structures implemented in a memory heirarchy which encompasses devices with widely varying characteristics, such as latency, bandwidth, unit of data transfer, *etc.*

3. Databases are resilient. Data is remembered in spite of hardware failures.

4. Databases provide concurrency control, *i.e.,* controlled access from multiple, independent processes, such that database consistency is maintained.

5. Databases provide facilities for access control, security, *etc.*

Of course, not all databases today provide all these features! Database system designers have always attached great importance to these characteristics while slowly evolving databases away from device-specific architectures and incorporating more and more features normally found in programming languages.

3 Database Programming Languages

We believe that the bottom-up evolution of database systems outlined in the previous section can limit our view of what databases are and what they should be. We have often experienced the following situation: When we point out the poverty of today's data models in comparison with type systems in programming-language, database researchers often respond that such richness and generality is "not necessary in database work". Unfortunately, this is merely a self-fulfilling assessment— the requirements of database work tend to get defined in terms of what is currently available and feasible. We firmly believe that there is ample need for richer type systems in databases.

We are therefore exploring the opposite evolutionary path: begin with an expressive programming language, and gradually incorporate features normally found only in databases, putting to good use all the lessons learned in implementing today's database systems. Of course, this book is evidence that several researchers share this view. Let us now look at some the features we would like in an integrated system.

A primary requirement for a database programming language is a rich type system in which one can model structures in a wide variety of application domains naturally. Arbitrary objects should be storable in the database, *i.e.,* there should be no distinction between programming language types and database types. There should be user-defined types and abstract data types, with rigorous semantics. It is desirable to support subtyping and inheritance— experience with Simula and Smalltalk have shown this to be a powerful modeling tool.

Another major requirement is for the programming language itself to be a very high-level language. The higher the level, the easier it is to produce and to maintain programs. This need is especially acute in the database environment, where, almost by definition we are dealing with a long-lived, shared, continuously evolving system. Among necessary advanced features we certainly include the capability of having functions as first-class values.

There have been some attempts to integrate programming languages with databases. For example, Pascal/R [18] is a system that integrates a relational database system with

Pascal. It extends the type system of Pascal to include a **relation** type built on Pascal records, and a **database** type built on relations, and it extends the control structures of Pascal to include constructs to traverse and manipulate relations. This is a very successful and elegant integration of a conventional database and programming language. This research has since evolved into the design of DBPL, a relational programming language based on Modula-2.

There are also many exciting efforts currently under way to integrate logic programming languages (such as Prolog) with relational database systems to create so-called knowledge bases [14]. The concept exploited here is that ground facts in a logic program have a natural interpretation as a relation in a relational database. This approach thus gives relational databases a reasoning capability, and allows smooth integration of intensionally- and extensionally-defined data.

However, a limitation of these approaches is that the languages used are not expressive enough in their own right (in particular, no functional values). In addition, not all types may be persistent, and those that can are too limited (just flat relations, in both cases).

PS-Algol [7] is a language in which objects of arbitrary type may persist. A database exists as a persistent heap external to the program, and there are mechanisms to make the database appear transparently as part of the standard program heap. Objects in the heap may be modified with the usual assignment statement, and any objects on the heap may persist beyond the lifetime of the program if they are made reachable from the top-level table of a database. Pointers to records are dynamically typed, giving a measure of polymorphism, and thus, type-checking is not completely static, though it is safe. PS-Algol has no abstract data type facility *per se*, although given the interesting combination of block structure, assignability of procedure values, the polymorphic procedure value **nullproc** and the fact that pointer types do not have to be declared, one can simulate abstract data types[8]. This work has since evolved into the design of Napier, a persistent language with richer types and stronger type-checking [9].

Galileo [1,2] is a database programming language with a rich polymorphic type system, based on ML [11]. In Galileo, the database is part of a global environment inside which each transaction runs. The Galileo type system includes mutable objects. A database program is an expression evaluated in this environment, and this evaluation may modify the updatable objects in the database.

In both PS-Algol and Galileo, mutable objects are an integral part of the language, and database updates are expressed using the standard assignment statement that destructively modifies an object in the database. Complex updates (*e.g.*, adding 5% interest to every account with more than $1000) are performed by embedding assignments in standard control structures.

In contrast, we are exploring a database programming language with an essential difference: the database is *immutable*, for which reason we call it a "*Functional Database*". An update transaction specifies a new version of the database in terms of the old. The same functional language is used both for queries and updates— a language in which there are no destructive assignments, and in which functions are first-class values.

We call this class of database programming languages FDBPLs, or Functional Database Programming Languages. We explicate and justify this approach in the next section, but first we summarize our view of the evolution of DBPLs below:

Conventional Programming Language + File Structures	—	COBOL, PL/1, *etc.*
⇓		
DB with limited types + alien PLs	—	Most current DBMSs
⇓		
Integrated DBPL with limited types	—	Pascal/R, DBPL, Prolog+RDB
⇓		
Integrated DBPL with rich types	—	PS-Algol, Napier, Galileo, FDBPL

4 A Functional Database Programming Language

In this section we outline a functional database programming language (FDBPL) and discuss its advantages.[2] The syntax used in these examples is still under development and is based on the dataflow language Id Nouveau [17,4].

4.1 What is a Database in FDBPL ?

In FDBPL a database is an *environment* of bindings— *i.e.,* a mapping from identifiers to types and values. The values named in an environment may be of any type, such as scalars, abstract types, data structures (*e.g.,* arrays and lists), and functions, but functions are central to the model.

Common database structures, especially indexing structures, are found as subtypes of the ordinary function type (written `t1 -> t2`). These subtypes are called *database* function types.[3]

[2]There seem to be some differences among various researchers in the use of the adjectives "functional" and "applicative" to describe a language. We use the terms interchangeably, with the meaning that it supports functions as values *and* is free from side-effects.

[3]Ordinary functions can be regarded as intensional, or programmed functions. Database functions can be regarded as extensional, or indexed functions.

To motivate this uniform functional view, let us draw an analogy with arrays. Abstractly, an array `A` of objects of type t is just a function from integers to t that is defined on a finite sub-domain of integers. To index the array is to apply the function to an integer argument. However, because it is a specialization of ordinary functions, an array admits additional operations, such as enumeration of the sub-domain on which it is defined.

In FDBPL we generalize this idea to encompass common database structures:

- The domain of a database function can be any type.
- Database functions often have associated inverses.
- The range of a database function is often a list.

To capture these variations, we introduce several database function type constructors which are specializations of the general "`->`" type constructor:

```
t1 --> t2          (1)
t1 -->* t2         (2)
t1 <--> t2         (3)
t1 <-->* t2        (4)
t1 *<-->* t2       (5)
```

A function `f` with any of these types has domain `t1` and range either `t2` (if the forward arrow is `>`), or `*t2`, *i.e.*, lists of `t2` (if the forward arrow is `>*`). In cases (3), (4) and (5), it also has an inverse named `^f` with domain `t2` and range `t1` or `*t2` depending on the reverse arrow.

For example, here is the type of an environment (*i.e.*, the schema) for a student-course database:

```
Student      : TYPE
mkStudent    : Void -> Student
Students     : *Student
SName        : Student  <-->  String
SStatus      : Student   -->  String
STotalUnits  : Student    ->  Number

Course       : TYPE
mkCourse     : Void -> Course
Courses      : *Course
CName        : Course  <-->  String
CUnits       : Course   -->  Number
CPrereq      : Course *<-->* Course
```

```
Enrollment    : TYPE
mkEnrollment  : Void -> Enrollment
Enrollments   : *Enrollment
EGrade        : Enrollment --> String

S-enroll      : Student <-->* Enrollment
C-enroll      : Course  <-->* Enrollment
```

Here Student, Course and Enrollment are database types. The name mkStudent is bound to a function of no arguments that creates a new Student object. The name Students is bound to a list of students (the "*" is a unary prefix type-constructor meaning "list of"). SName is a 1-1 function between Students and Strings; this means that SName applied to a Student returns a String and ^SName applied to a String returns the Student with that name. SStatus maps Students to Strings. STotalUnits is an ordinary function from Students to Numbers.

CPrereq is a many-to-many function between Courses and Courses. When applied to a Course, it returns a list (perhaps empty) of Courses that are its prerequisites. When ^CPrereq is applied to a Course, it returns a list of Courses for which it is a prerequisite.

S-enroll is a 1-to-many mapping between Students and Enrollments; when S-enroll is applied to a Student it returns a list of his/her Enrollments, and ^S-enroll applied to an Enrollment returns the Student corresponding to that Enrollment.

In conventional databases, Student, Course and Enrollment would be record types (tuple types), Students, Courses and Enrollments would be relations, SName and CName would be key fields, SStatus, CUnits would be ordinary fields, and functions like CPrereq and S-enroll and C-enroll would correspond to various joins between relations.

4.1.1 Queries

A *Query* is merely a functional expression evaluated in the database environment, producing its value as the answer. In functional languages, the treatment of functions as values is central, allowing the use of poweful, bulk operators to express computations concisely [20]. Examples of such operators are:

- (map f l) returns a list containing the results of applying the function f to each member of the list l.
- (filter p l) returns a list containing just those members of l that satisfy the predicate p.
- (fold op v l) returns an accumulated value over the list l, obtained by applying the binary function op pairwise to each element of l, with initial value v. For example, fold (+) 0 l sums up the list l.

Functions are curried, so that `(fold (+) 0)` is a function that takes a list of numbers as its argument and sums it up. The advantage of treating all database structures as functions is uniformity— one can then extend all the power of these high-level operators to database structures.

For example, here is a query to find the names of all special-status students taking 15-unit courses:

```
{  e_has_15 e          = (CUnits (^c-enroll e) == 15) ;
   es_with_15          = filter e_has_15 Enrollments ;
   ss_with_es_with_15  = map ^s-enroll es_with_15 ;
   special_s s         = (SStatus s) == "special"
 In
   map SName
       (filter special_s
               ss_with_es_with_15) }
```

We define a temporary predicate `e_has_15` that decides if the course related to an `Enrollment` object has 15 units. Using it, we filter all enrollments to compute `es_with_15`, the list of all 15-unit enrollments. Using `^s-enroll`, we map it into a list of the corresponding students. Finally, we filter it with the predicate `special_s` to get the students of interest, and map `SName` over that list to get their names. This operator-based view of functional query languages and methods to implement them are explored in [16].

The function `STotalUnits` shown in the database environment is an ordinary function. Here is a possible definition for it:

```
Def STotalUnits s = {  EnrollUnits e = CUnits (^c-enroll e)
                    In
                      fold (+) 0
                           (map EnrollUnits (^s-enroll s))} ;
```

i.e., when applied to a `Student`, it computes that student's total units using other database functions. This kind of function is sometimes called a "derived function" in the literature.

Here is a recursive query that checks if the course `"6.004"` is directly or indirectly a prerequisite for the course `"6.847"`:

```
{ q c1 c2 = if (c1 == c2) then true
            else fold (or) false (map (q c1) (CPrereqs c2)) ;

 In
  q (^CName "6.004") (^CName "6.847") }
```

Note that one mixes database and ordinary functions freely. Definitions for ordinary functions may use recursion, conditionals, *etc.* In short, the language is a full programming language.

4.1.2 Operations on Database Functions

For queries, one makes no distinction between ordinary functions and database functions, *i.e.*, one forgets that a database function is a subtype of ordinary functions, and the only interesting operation on functions that we use is function-application. In order to deal with updates, however, there are some additional useful operations on database functions. These operations are *incremental definition* operations, and are inspired by *I-structures* of dataflow research [6].

If `f` is a database function (a subtype of `t1 -> t2`), the expression:

```
domain f
```

has type `*t1`, and returns a list of objects of type `t1` on which `f` is defined. This is analogous to an operation on an array that returns its index bounds.

If `f` is a database function (a subtype of `t1 -> t2`), the expression:

```
new f
```

returns a new database function of the same type that is everywhere undefined. Let this new function be called `g`. If `x:t1`, `y:t2` and `g` is undefined at `x`, then the statement:

```
g [x] = y
```

is an *incremental definition* of `g`, *i.e.*, it refines, or constrains `g` so that it now maps domain value `x` into range value `y`. By executing many such statements, the meaning of `g` is gradually "filled in". It is a runtime error to specify two different mappings for `g` at any domain value `x`. This distinguishes it from conventional assignment statements, and is sometimes also called the "single-assignment" criterion.

For example, suppose we wish to compute a function that is identical to `SStatus` except that the status of student Smith is to be special. We could write this (in laborious detail) as:

```
      {
(1)      g = new SStatus ;
(2)      smith = ^Sname "Smith" ;
(3)      g [smith] = "special" ;
(4a)     map (fun s. g[s] = SStatus s)
(4b)         (list_without (domain SStatus) smith) ;
      In
(5)      g }
```

(The line numbers are not part of the program.) In line 1, g is bound to a new undefined function with the same domain and range types as `SStatus`. In line 2, `smith` is bound to the student object with name "Smith". In line 3, g is refined to map Smith to "special". In lines 4a and 4b, g is refined so that `(g s)` is the same as `(SStatus s)` for all other students.[4] In line 5, g is returned as the value of the whole block.

Note that the incremental definition statement is *not* a destructive assignment. The single-assignment criterion ensures that g never has a value different from its previous one— it always has a value consistent with its previous value. Each incremental definition of g *monotonically* increases its information content.

A dramatic implication of these semantics is that it retains the parallelism and determinate semantics of a conventional functional language. Unlike imperative languages with destructive assignments, there is no required sequential ordering between the various expressions of a program. In our example above, lines 3 and 4a-4b can be executed in parallel. The function g has a unique meaning, independent of the order of execution.

4.1.3 Update Transactions

An update transaction in FDBPL is a specification of a new database environment e' in terms of the existing database environment e. Recall that an environment is a set of bindings of names to types and values. We follow the convention that a name of the form `f'` refers to the value of `f` in e', while a name of the form `f` refers to the value of `f` in e.

Semantically, an update transaction in our example database that merely changed student Smith's status to "special" would specify trivial new bindings for all names except `SStatus`:

```
Student' = Student ;
mkStudent' = mkStudent ;
Students' = Students ;
SName' = SName ;
SStatus' = {g = new SStatus; ... as in example above ... } ;
STotalUnits' = STotalUnits' ;
...
```

Syntactically, of course, this would be too verbose, because in most database updates, the bulk of the database is untouched, and only a few objects are likely to have interesting redefinitions. So, while retaining the above semantics, we provide the following more convenient syntactic conventions for an update transaction:

- Unless explicitly dropped using "`drop f`", every name `f` is implicitly carried over to the new environment as if it had been defined by "`f' = new f`".

[4] We use the notation `(fun x. e)` for lambda-expressions.

- Only new name bindings and interesting incremental definitions are specified in the update transaction.
- At the end of an update transaction, if `d'` is the domain on which `f'` is defined (*via* incremental definitions in the transaction), the following expression is executed for each `f`:

```
map (fun x. f' [x] = f x)
    (list_difference (domain f) d')
```

i.e., for all domain values of `f` for which `f'` was not defined, `f'` is now defined to have the same mapping as `f`.

Thus, syntactically our example update transaction to change Smith's status to "special" need only say:

```
SStatus' [^Sname "Smith"] = "special"
```

The names from both the new and the old environments are available in the update expression. For example, to increase the weight of the course "`Intro to Spelunking`" by 3 units, we would write:

```
CWeight' [^CName "Intro to Spelunking"] = (CWeight c) + 3
```

To increase the weight of *all* courses by 3 units, we would write:

```
map (fun c. CWeight' [c] = (CWeight c) + 3)
    Courses
```

4.1.4 A Functional Database System

The above discussion was concerned only with the question of how *one* update transaction specifies a new environment with respect to a given old environment. A Functional Database *system* still has to decide how to connect together multiple updates and queries. There are many possibilities, differing in the kind of serialization of transactions that is necessary.

The simplest possibility is for the system to maintain only one, current database environment. Queries are evaluated in this environment. After computing a new environment in response to an update transaction, the transaction is finally committed by designating this as the current environment and discarding the old one. For consistency, update transactions must be serialized, and multiple queries can be run concurrently, but only *between* transaction-commits. Even though such a system appears similar to a conventional

database system, we will argue in the next section that the use of a functional language itself has advantages.

A second possibility is for the system to maintain a *history* of database environments. Each query or update transaction is evaluated with respect to the environment that is latest at the moment the transaction is received. Each update transaction, when it commits, extends the history with the new environment that it specifies. To maintain a linear history, update transactions must be serialized. Multiple queries may be evaluated concurrently; they never have to be delayed because of update transactions.

In this scenario, one could also extend the language so that expressions can be evaluated in any environment in the history, instead of just the current one. First, we need a sub-language of "environment expressions" to identify specific environments, for example, by an absolute or relative index in the history, or by a real-time stamp. Using this, we can qualify an expression by the environment in which it should be evaluated. The notation used above— `f'` and `f` refer to new and old versions of `f`, respectively— can be seen as a special case of environment-relative naming.

In Section 4.3 we argue that this linear history model is an attractive and useful one.

A third possibility is to maintain a *tree* of database environments rather than a linear history. Here, an update transaction can grow a fresh branch by specifying any point in the tree as the environment relative to which it is to be evaluated. The only transactions that need to be serialized are those that grow the same branch of the tree. Of course, in this tree model, it is necessary to have a way to refer to different points in the tree.

While this third possibility is very general, it is not clear how useful it is— a database being a shared resource, the linear history seems to makes more sense, the latest point in the history being the common, shared view of the database. However, the tree model can be useful, especially in engineering databases, where it is often desirable to try several experimental alternatives to a design, later choosing one of them as the common, shared view.

4.2 Why Functional Languages?

4.2.1 Expressiveness

It is widely accepted that functional languages have great elegance and expressive power [20]. Various high-level features of functional languages give them this expressive power.

In functional languages, functions are first-class values. This is a powerful abstraction mechanism that permits the programmer to design appropriate control structures for existing and new types of objects, thus leading to compact, transparent notation. For example, in defining a new table type in a database, one can simultaneously define general-purpose generators, iterators and reducers to operate on such tables, so that subsequent uses of such tables are concise and clear.

Non-strictness of functions is also a useful tool, making it feasible to define and manipulate infinite and/or large objects perspicuously [20]. However, non-strictness requires that there be no destructive update and its attendant sequentiality, because the order in which expressions are evaluated is in general unpredictable.

Functional languages are expression-based— the main composition rule is function application and is used uniformly from small expressions to large programs. Thus the same language can be used not only for quick, one-off queries in interactive environments but also for large compiled programs.

Even though we believe that functional languages are easier to use because of their clean semantics and high-level features, they are still formal languages. Under certain circumstances, such as casual use of a database by untrained users, one may wish to use other front-ends such as natural-language or graphical interfaces. In such situations, the regularity of functional languages makes them good target languages into which queries are first compiled.

4.2.2 Query Optimizations via Transformation

The semantic cleanliness of functional languages makes it feasible to perform many meaning-preserving program transformations automatically. Because functional languages are referentially transparent, they admit a rich algebra of programs, a feature essential for effective query optimization. The success of current relational database implementations relies in no small measure on the ability to optimize queries in the relational algebra, which is a (restricted) functional language.

4.2.3 Optimizations Due to Parallelism and Non-strictness

Side-effects in a language introduce many read-before-write and write-before-read constraints. Most of these constraints are artificial and make it very difficult, if not impossible, to move away from a purely sequential scheduling of evaluation activities. In functional languages, only the logically necessary data-dependencies remain; this reveals much fine-grained parallelism that permits great latitude in scheduling evaluation activities. This degree of freedom is exploited heavily in dataflow architectures to overcome memory latencies [3,5]. We believe that the same techniques could be used to overcome disk latencies too, a major factor in database system (in)efficiency.

Non-strict functions also increase the parallelism available in a program, because they relax data dependencies even further [5]. In addition, non-strict functions have certain automatic database optimization capabilities in that unnecessary computations (which may involve disk accesses) never get done [16].

We are currently investigating the possibility of high performance *via* parallelism by studying support for persistence in the MIT Tagged-Token Dataflow architecture.

4.2.4 Cleaner Type Systems

There is much research nowadays into polymorphic type systems [19]. Polymorphism allows an economic style of programming *a la* Lisp but with complete type safety and with the rich data abstraction facilities necessary for database work. These data abstraction facilities include user-defined abstract types, and type-heirarchies with inheritance.

Much of this research is centered on functional programming languages. Extensions to languages with updates is not often straightforward. For example, in ML there is a well-known problem with polymorphic mutable cells. Consider the following program:

```
def f:(t -> (list t)) =
  {  a = ref nil
     g x = {prog
              a := cons x (deref a) ;
              deref a }
   In
     g } ;
```

Here, `f` is function with a local variable `a`, bound to a polymorphic mutable cell, initially containing the polymorphic list `nil`. In each call, the list is updated by consing the argument onto this list. The function has been declared to have the polymorphic type `(t -> (list t))` and its definition is consistent with that declaration. The calls `(f true)` and `(f 0)` are therefore individually well-typed, but together they produce a list with mixed types, violating the type discipline. We must point out that there are solutions to this problem, but they are either unintuitive, or they involve *ad hoc* language restrictions (ML originally did not allow polymorphic mutable cells for this reason).

The presence of mutable objects also complicates abstract data type facilities. When building an object of abstract type, one is very careful to ensure that its internal representation cannot be accessed by procedures outside the abstract type definition. Such sharing can make it impossible to guarantee the invariants supposedly maintained by the data abstraction, because it is then possible to update the representation of an abstract object without using one of the allowed procedures, and thus circumvent any integrity checks. This sharing occurs because an abstract object constructor often receives parameters from the outside which it may embed in the new abstract object that it creates. In functional languages, this sharing is safe and so the embedding may be achieved very cheaply (by reference); in languages with side-effects, it may imply expensive and/or excessive *copying* of objects in order to ensure that representations of abstract objects are truly private. In addition to this execution overhead, there is the intellectual overhead for the programmer in that he has to be aware of this pitfall and must explicitly take precautionary measures.

4.3 Why Functional Databases?

Assuming the linear-history model introduced in 4.1.4, functional databases "never forget". A database is never modified; instead, it is "updated" by creating a new version that is

appended to the history of versions. The versions form a (conceptually) infinite sequence of database environments. Older versions can be named and accessed just like the latest version.

There are many situations where this approach has already been taken, though perhaps not in such a formal sense. For example, many modern file systems (*e.g.*, Vax/VMS) maintain file versions, and this is widely regarded as far superior to no-backup or one-level-backup file systems. Another example is the "Undo" capability now appearing in some programming environments and text editors. Challis also poses arguments in favour of multi-version databases in [12], where it is claimed that such databases simplify recovery and concurrency algorithms.

Many existing database systems *do* in fact retain all the information contained in previous versions of a database, but only for crash recovery and audit purposes. This is not part of the data model, and the data is not directly accessible to an applications program.

A functional database can offer a superior environment for concurrency control. Queries (read-only transactions) are never delayed, because there is always a latest, committed version that is never going to be changed subsequently. Update transactions do not interfere with queries, because they always build new versions and never interfere with any version currently being examined by queries. Multiple updates, however, still must be serialized (though if two updates work on different parts of the database, they may actually be able to proceed in parallel).

Assuming parallel execution, non-strictness can also improve update performance, because it allows logically serialized operations to be physically overlapped, thus "pipelining" a sequence of updates, instead of doing them strictly one at a time.

Functional databases permit uniform access to historical data. In current databases, if access to historical data is required, the database designer must explicitly encode histories as lists in the schema, and write applications to maintain those lists. An example is the history of withdrawals and deposits against a bank account. Thus, the way these histories are encoded and accessed can vary from database to database, and even within a single database. Further, historical data is available only if anticipated by the database designer. In a functional database system, since the history of versions is visible *via* expressions that are qualified by environment expressions, the programmer has uniform access to the history of any part of the database.

Because the model explicitly supports the notion of a new version of the database, and because it is so easy to revert to old versions, functional databases permit one to write "what-if" programs that experiment with possible futures. This capability would be extremely useful in databases for engineering design, and in so-called intelligent databases or knowledge-bases. For example, when designing a part, there may be several alternative designs that must be pursued simultaneously because it is not yet clear which one would be the most suitable.

Functional databases may ameliorate the difficult problem of manually or automatically

merging two separate databases into one. This problem arises, for example, when the operations of two or more companies are consolidated. Typically, the data in the two source databases will disagree in various ways. Having access to the histories of the two databases may alleviate the inconsistency problem because one may be able to use pattern-matching algorithms to establish consistent correspondences between the data at different points in the history of the two databases.

In many information systems, to the history of the database is important. It may be a legal requirement to maintain records of all transactions, for example in a bank or personnel database (in fact banks already do this, but without any direct help from the data model or the database management system). The ability to examine time trends, for example in a stock-market or econometric modelling database, can be an invaluable capability.

5 Implementation Issues

We are in the process of designing a prototype of a Functional Database System. Here are some of our early design ideas.

There are many implementation techniques currently in use for functional languages. The one that we are currently pursuing is the dataflow approach [5], because of its support for non-strictness and because it is an approach to parallelism for higher performance.

To implement an FDBPL, the memory is organized into a sequence of heaps. Each version of the database is associated with its own heap, though this does not mean that the entire database version resides on that heap. Each update transaction initiates a new heap, and new objects constructed as part of that transaction are allocated in the new heap, though they may refer back to objects in older heaps. Thus an updated version of the database shares much of the previous versions; only new objects allocated as part of the new version initially reside in the new heap for that version.

The programmer's model of memory is that the entire sequence of databases (*i.e.*, the sequence of heaps) is implemented in stable storage and is resilient across crashes. As a transparent optimization, pages of various heaps will have temporary working copies in high-speed, volatile memory. When a transaction is committed, the copy in stable storage is made to agree with any extant temporary copy.

Even though the data model presents the database as a potentially infinite history of versions, because of finite storage capacity it will in general be necessary to prune the sequence of databases at some version, and move all prior versions off-line. To do this, all data shared with prior versions are copied forward into a fresh heap at the oldest retained version. Decisions to prune the version history, and identification of the oldest version to be retained are taken dynamically depending on current storage utilization, for example by a database administrator.

Our semantics ensures that one cannot see different data as a result of this pruning activity. When the version history is pruned, one will no longer be able to execute a query

that interrogates a pruned version— such attempts will invoke a run-time error. Despite this, if the total storage capacity of the database system is subsequently increased (*e.g.*, by adding more disks), it is possible to re-integrate earlier pruned verions easily and smoothly.

We emphasize though we still have to live with finite databases, we believe our model is still a step forward. We have relieved the database programmer of concerns about finiteness in much the same way that the Algol programmer is relieved of certain finite storage allocation issues that bedevil the Fortran programmer, even though in a deep theoretical sense they are certainly equivalent.

It appears that our model seems naturally suited to high-capacity write-once storage technology such as optical disks, but we have yet to study algorithms that make effective use of such devices in this functional setting.

6 Conclusion

We have argued that it is essential that databases be treated as an integral part of programming languages rather than separately, as is common today. Further, we have claimed that the full range of type-structures in a programming language should be available for database objects. We are in substantial agreement with other researchers on these positions.

We then put forward the view that functional languages offer many advantages having to do with expressive power, rich type systems, interactive environments, parallelism and optimization. Databases should be viewed as functionally, as environments in which queries (*i.e.*, expressions) are evaluated. They should be "updated" not by destructive assignment, but by specifying a new environment, thus creating a conceptually infinite sequence of database versions. We call such a system a Functional Database Programming Language. We argue that FDPLs are superior vehicles for concurrency control, and that existence of a model of the history of a database is useful not only for recovery mechanisms but also in general for application programs.

We have begun developing a prototype functional database programming language to demonstrate the feasibility of this model.

References

[1] Antonio Albano, Luca Cardelli, and Renso Orsini. *Galileo: a Strongly Typed Interactive Conceptual Language.* Technical Report 83-11271-2, Bell Laboratories, 1983.

[2] Antonio Albano, F. Giannotti, Renso Orsini, and D. Pedreschi. The type system of galileo. In *Proc. 1985 Persistence and Data Types Workshop, Appin, Scotland,* August 1985.

[3] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. *Id Nouveau Reference Manual; Part II: Operational Semantics.* Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, April 1987.

[5] Arvind and Rishiyur Sivaswami Nikhil. Executing a program on the mit tagged-token dataflow architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*, Springer-Verlag, June 15-19 1987.

[6] Arvind, Rishiyur Sivaswami Nikhil, and Keshav Kumar Pingali. *I-Structures: Data Structures for Parallel Computing.* Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1987. (Also to appear in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[7] Malcolm P. Atkinson, K.J. Chisholm, and W.P. Cockshott. Ps-algol: an algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1981.

[8] Malcolm P. Atkinson and Ron Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[9] Malcolm P. Atkinson and Ron Morrison. Types, bindings and parameters in a persistent environment. In *Proc. 1985 Persistence and Data Types Workshop, Appin, Scotland*, August 1985.

[10] Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages.* Springer Verlag, Berlin, 1984.

[11] Luca Cardelli. *ML Under Unix.* Technical Report, AT&T Bell Laboratories, 1983.

[12] M.P. Challis. Version management - or how to implement transactions without a recovery log. *Database: Infotech State of the Art Report*, 9(8):435–458, January 1982.

[13] Christopher J. Date. *An Introduction to Database Systems, Volume I (Fourth Edition).* Addison Wesley, Reading, MA, USA, 1986.

[14] Herve Gallaire, Jack Minker, and J-M Nicolas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[15] T.H. Merrett. Persistence and aldat. In *Proc. 1985 Persistence and Data Types Workshop, Appin, Scotland*, August 1985.

[16] Rishiyur S. Nikhil. *An Incremental, Strongly-Typed Database Query Language.* PhD thesis, Moore School, University of Pennsylvania, Philadelphia, PA, August 1984. Available as Technical Report MS-CIS-85-02.

[17] Rishiyur Sivaswami Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[18] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1977.

[19] *International Symposium on the Semantics of Data Types, Sophia-Antipolis, France (Springer-Verlag LNCS 173)*, June 1984.

[20] David A. Turner. The semantic elegance of applicative languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92, ACM, October 1981.