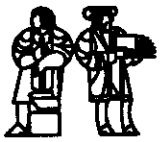
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Semantics of Update in a Functional Database Programming Language

Computation Structures Group Memo 276

December 7, 1987

Rishiyur S. Nikhil

To appear in: Proceedings of the 1987 ALTAIR/CRAI Workshop on Database Programming Languages; September 1987, Roscoff, France; Francois Bancilhon and Peter Buneman (*eds.*)

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

The Semantics of Update in a Functional Database Programming Language

R.S.Nikhil

MIT Laboratory for Computer Science

545 Technology Square,
Cambridge MA 02139, USA

Arpanet: nikhil@xx.lcs.mit.edu

Databases that can store complex, nested objects may suffer performance penalties for their generality. Parallelism may be a solution. However, we need database languages that can express parallelism, and implementations that can exploit it. Functional languages and and their dataflow implementations are one approach, at least for queries. However, it has not been easy to express database updates in functional languages. In this paper we present a model for databases and updates in a functional language, with an intended dataflow implementation. The update language is declarative, parallel, and determinate, and can be extended to model historical data.

1 Introduction

The dichotomy between databases and programming languages is one of expedience. Ideally, it should be possible for arbitrary objects created and manipulated by programs to be persistent. But today, we know how to implement persistence efficiently only by restricting the structure of persistent objects and the operations that can be done on them.

For example, in current relational database systems, persistent objects must be flat, rectangular tables containing scalar values, and they must be manipulated only by a given set of relational operations. It is generally not easy to change the structure of the tables or to write arbitrary programs to manipulate them. Because of these restrictions, the database implementor can pre-plan disk layouts for the tables, can create indexes that use knowledge of these layouts, and compile queries so that they exploit this information thoroughly.

The limitations on structure and operations in current database systems are a serious hindrance in applications that must model complex data (*e.g.,* in engineering design). In

[0]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 Office of Naval Research contract N00014-84-K-0099.

some applications, the limitations are so severe that implementors bypass database systems entirely— they store data in ordinary files, flattening objects for writing and reconstructing them on reading. Needless to say, the performance of persistent operations is not good. (Nor does this approach address issues of concurency control and resilience.)

Sometimes, implementors *encode* complex objects within, say, an existing relational system. For example, object containment in a programming language is normally implemented by pointers; in a relational system, pointers can be mimicked by foreign keys. But this layer of interpretation can again degrade performance, since the underlying relational system is unaware of the encoding. (Nor does this approach support data abstraction.)

It is clear that persistence of complex objects must be supported directly (see also [1, 7, 8]). But how to get adequate performance? One possibility is through parallelism. For this to become a reality, we need languages that do not obscure parallelism, and implementations that exploit the available parallelism. Functional languages have long been touted as suitable for this purpose.

Our approach is based on the Tagged-token Dataflow parallel execution model [5]. Here, we begin with Id, a language designed for easy compilation into dataflow graphs, a parallel machine language. The core of Id is a functional programming language, including higher-order functions and non-strictness in procedure calls and data structure constructors.

But in purely functional languages, incremental updates to large data structures can be difficult to express (see Section 5.4 for more on this topic), and are usually difficult to implement efficiently (see [6] for more on this topic). "I-structures" in Id are an attempt to address these issues. I-structures are array-like data structures that can be incrementally defined (like updatable arrays), yet retain the determinacy and parallelism of functional languages.

In this paper, we present some ideas on how to extend Id to make it into a database programming language. First, we generalize the I-structure idea to a richer class of indexed structures supporting incremental definition. Then, we introduce the notion of databases and update transactions on databases. Along the way, we will find that the model can be extended naturally to express the retention and manipulation of historical data.

2 Databases as Environments

Consider an interactive programming system for a functional language. At any given time, there is a current *environment* of bindings associating names to values. The user engages in two kinds of activities:

- When a user enters a *definition*, he[1] is specifying a new environment, usually in terms of the old. He is adding a new binding, or redefining an existing one. Viewing the environment as a database, this is an *update* transaction.

[1] We use *he* instead of the more awkward *he or she*.

- When the user enters an *expression*, it is evaluated in the current environment, and an answer is printed. Viewing the environment as a database, this is a database *query*.

We model a database on exactly this idea. A database is an environment of bindings; update transactions specify new environments in terms of old; and queries are simply expressions evaluated in the latest environment. Thus, the operation of a single-user database system can be specified as a function from a list of transactions to a list of responses:[2]

```
Def dbsystem db (cons xact xacts) =
        {  resp, new_db = eval db xact
         In
           cons resp (dbsystem new_db xacts) } ;

dbsystem empty_db xacts
```

The phrase `(cons xact xacts)` is a pattern that matches the input list, binding the name `xact` to the first transaction and `xacts` to the rest of the list of transactions. `Xact` is evaluated in the database environment to produce a response and a new database environment (of course, for query transactions, the new database will be the same as the old). Finally, we construct and return the list of responses, beginning with this response and followed by the remaining list of responses obtained by running the remaining transactions against the new database.

A database shared among multiple users needs a little more packaging: we need a *manager* that non-deterministically receives transactions from individual users and merges them into a single list of transactions as input to `dbsystem`. The responses from `dbsystem` must then be despatched to the appropriate users. The details are outside the scope of this paper; the interested reader is referred to [9] or [3] for suggested solutions.

Each binding in the database associates a name to a *database type* or to a *value* of arbitrary type. The type structure is:

- Primitive types: `V` (void), `N` (numbers), `B` (booleans), `S` (strings), `SYM` (symbols) ...
- Database types: `Student`, `Course`, `Department`, ...

  These can be viewed as abstract types and are introduced by the user explicitly. For each database type `t`, the system provides:

[2] We use Id notation here. As in most modern functinal languages, application of a function `f` to argument `a` is written by juxtaposition: `f a`. Blocks (analogous to `let` or `where` expressions) are written

```
{    statement ;
     ...
     statement
In
     expression }
```

The left-hand sides of statements can be *patterns* that match the structure of the values returned by the right-hand-sides.

- a constructor function "`make t`" which, when applied to `():V`, returns a new object of type `t`.
- a set "`all t`" containing all objects of type `t`.

- Sets of objects of arbitrary, but uniform type: `*t`
- Finite tuples of objects of arbitrary type: `(t1,...,tn)`
- Function types: `t1 -> t2`

  We will refer to such functions as *ordinary* functions, in contrast with indexed functions below.

- Indexed types: For every primitive and database type `t1` and `t2`, there are indexed types:[3]

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

  The similarity to ordinary functions is deliberate— in queries, they are regarded as functions on finite domains. Like functions, they can be applied to arguments to produce results. For example, the object `SStatus: Student => S` can be applied to a `Student` object to produce a string. The indexed types represent one-to-one, one-to-many and many-to-many relationships, with or without inverses. We will see examples below.

  We will refer to objects with indexed types as *indexed* functions.

For example, here is (the type of) a segment of a typical database:

```
Student      : TYPE
SName        : Student <=> S
SStatus      : Student  => S
STotalUnits  : Student -> N
```

`Student` is a database type. `SName` is an invertible indexed function that maps `Students` to strings. There is a generic inverting funtion `^`, such that "`^ SName`" maps strings to `Students`. `SStatus` is a (non-invertible) indexed function that maps `Students` to strings, and `STotalUnits` is an ordinary function that maps `Students` to numbers.

Another segment of the database:

[3] In [10], we used the notation `-->`, `-->*`, *etc.*, but `-->` was easily confused with `->` in hand-written text.

```
Course      : TYPE
CName       : Course <=> S
CUnits      : Course => N
CPrereq     : Course *<=>* Course
```

`Course` is another database type. `CPrereq` maps a `Course` into a set of `Courses` that are its prerequisites. "`^ CPrereq`" maps a `Course` into a set of `Courses` for which it is a prerequisite.

The (type of the) rest of the database:

```
Enrollment  : TYPE
EGrade      : Enrollment => S

S-Enroll    : Student <=>* Enrollment
C-Enroll    : Course  <=>* Enrollment
```

`S-Enroll` maps a `Student` into the set of his `Enrollments`, while "`^ S-Enroll`" maps an `Enrollment` into the corresponding `Student`.

The database type `Enrollment` (with associated functions) was introduced to model the event of a student enrolling in a course, which allows associating various data with that event, such as grade, date of enrollment, name of supervisor who approved it, *etc.* An alternative strategy would be to define the following functions directly on `Students` and `Courses`:

```
Takes-Courses: Student *<=>* Course
Grade        : (Student,Course) -> S
```

In conventional database terminology, our database types correspond to distinct record types. "`=>`" corresponds to an ordinary record field, whereas "`<=>`" corresponds to a record field that is also a key. The other indexed types correspond to one-to-many and many-to-many relationships, usually obtained by set owner/member links in CODASYL databases, and by joins in relational databases.

2.1 Queries

Queries are arbitrary applicative expressions evaluated in the database environment. A very powerful notation for expressions on collections is the "set comprehension" notation invented by Turner [14, 13]. This notation can be regarded as a significant generalization of relational calculus languages like SQL.

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

```
{ SName s | s <- all Student    ; SStatus s == "Special" ;
            c <- all Course     ; CUnits c  == 15 ;
            e <- all Enrollment ; ^ S-Enroll e == s ;
                                  ^ C-Enroll e == c  }
```

In words: For all students s such that his status is special, for all courses c such that its units are 15, for all enrollments corresponding to s and c, return the name of s.

Of course, the above expression may not be the most efficient way to compute the result. Here is a more efficient solution:

```
{ SName (^ S-Enroll e) | e <- all Enrollment ;
                         CUnits (^ C-Enroll e) == 15 ;
                         SStatus (^ S-Enroll e) == "special" }
```

In words: For all enrollments corresponding to 15 unit courses and special-status students, return the name of the corresponding student.

One of the strengths of a functional language is that its clean semantics makes programs amenable to very powerful transforms, resulting in significant optimizations. It is possible that such a transformer could automatically produce the second query from the first.

The user can use a more algebraic notation that exploits a significant feature of functional languages: functions (both ordinary and indexed functions) are first-class values, *i.e.*, they can be arguments to and results from other functions, and components of data structures. This allows the use of high-level bulk operators to express common computations concisely. Examples of such operators are:

- "`map f l`" returns a set containing the results of applying the function `f` to each member of the set `l`.
- "`filter p l`" returns a set containing just those members of `l` that satisfy the predicate function `p`.
- "`compose f1 f2`" is a function that takes an argument `x` and returns `(f1 (f2 x))`.
- "`fold op v l`" returns an accumulated value over the set `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 set `l`.

We use the standard curried notation of functional languages, so that "`fold (+) 0`" is a function that takes a set of numbers as its argument and sums it up. The advantage of treating indexed functions like ordinary functions is uniformity— one can then extend all the power of these high-level operators to database structures.

Here is a more algebraic notation for the efficient version of the above query:

```
{  e_15_special e  =     (CUnits  (^ C-Enroll e) == 15)
                     and (SStatus (^ S-Enroll e) == "special")
 In
   map (compose SName (^ S-Enroll))
       (filter e_15_special
               (all Enrollment)) }
```

`e_15_special` is a predicate that decides if the course related to enrollment `e` has 15 units and the student related to `e` has special status. Using it, we filter all enrollments, and map the composition of `SName` and `^ S-Enroll` over the remaining enrollments to produce the desired set. This operator-based view of functional query languages and methods to implement them are explored at length in [12].

Because of our parallel model of computation, the enumeration of enrollments, the filtering and the final mapping are all overlapped in a pipelined manner (see [11]).

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

```
Def STotalUnits s =
          fold (+) 0
                   { CUnits (^ C-Enroll e) | e <- S-Enroll s }
```

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

Here is a recursive query that checks if the course `"6.001"` is directly or indirectly a prerequisite for the course `"6.004"`:

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

 In
  q (^ CName "6.001") (^ CName "6.004") }
```

Note that one mixes indexed and ordinary functions freely. Definitions for ordinary functions may use recursion, conditionals, *etc.* In short, the query language is a complete, high-level programming language.

3 Operations on Indexed Functions

Indexed functions differ from ordinary functions in that they are defined incrementally with many statements, rather than in a single statement. An indexed function is first created using the "`empty`" construct, at which point it has an empty domain (undefined everywhere). It has zero information content, and is said to be "open". As the transaction progresses, incremental definitions *monotonically* (*i.e.*, consistently) add more information to the object, defining it over a larger and larger domain. When the transaction completes, *i.e.*, the program terminates, the value of the indexed function is frozen, and it is said to be "closed". Incremental definitions can only be given for open indexed functions, *i.e.*, new indexed functions introduced in the current transaction.

The operations described below are inspired by I-structure operations in the dataflow language Id [6].

3.1 Single-Valued Index Functions: => and <=>

The expression:

```
empty (t1 => t2)
```

returns a new, empty, indexed function of type `t1 => t2`, *i.e.*, it initially maps all arguments to $\perp$. For convenience, one can also say "`empty e`" where `e` is any expression of type `t1 => t2` (the value of `e` is irrelevant— the system only uses its type).

Given an indexed function `f: t1 => t2`, and expressions `e1:t1` and `e2:t2` that evaluate to `v` and `w`, respectively, the statement:

```
f [e1] = e2
```

extends the definition of `f` so that it maps `v` to `w`. By executing many incremental definition statements, an initially empty `f` is gradually "filled in", defining it over a larger and larger domain.

Contrast this with statements for ordinary function definitions. For example,

```
g x = e
```

defines `g` at once for *all* arguments `x`. Each incremental definition statement, on the other hand, defines `f` only for one *specific* argument.

In our parallel model of computation, the order in which the incremental definitions are executed is unpredictable. Consequently, we allow a definition for `f` at the each argument `v` *at most once*, *i.e.*, the value of `(f v)` can make at most one transition, from $\perp$ to some `w`. Any attempt to redefine `f` at `v` so that it has some other value `w'` is treated as an *inconsistent* specification of `f`, and causes a runtime error. This rule is sometimes called the "single-assignment" rule in dataflow literature.

If `f` is applied to an argument `v` within the same program, that part of the computation simply waits (if necessary) until `f` is defined at `v` by some other, concurrent part of the computation. The requirement that `f` can be given only a single definition at `v` ensures that each function application returns a unique, determinate result.

As we shall see later, in update transactions a new `f: t1 => t2` automatically inherits mappings from an old version unless specified otherwise. To inhibit this, for an expression `e1:t1` that evaluates to `v`, the statement:

```
f [e1] = undef
```

specifies that `(f v)` is always undefined. Any other attempt to define `f` at `v` is an error.

The treatment of `<=>` is similar. For an indexed function `f: t1 <=> t2` and expressions `e1:t1` and `e2:t2` that evaluate to `v` and `w` respectively, the statement:

```
f [e1] = e2
```

defines (f v) to be w and (^ f w) to be v. It will succeed only if f was previously undefined at v *and* if (^ f) was previously undefined at w.

For an indexed function f: t1 <=> t2 and an expression e1:t1 that evaluate to v, the statement:

```
f [e1] = undef
```

specifies that (f v) is always undefined. Any other attempt to define f at v is an error.

3.2 Multiple-Valued Index Functions: =>*, <=>* and *<=>*

Multiple-valued indexed functions initially map all arguments to $\perp_{set}$, the undefined set. As incremental definitions at some argument v are executed, the mapping improves to (insert w1 $\perp_{set}$), (insert w1 (insert w2 $\perp_{set}$)), and so on. If, at the end of the transaction, (f v) is

```
(insert w1 (...  (insert wn ⊥set)))
```

then it becomes closed with those values, *i.e.*, (f v) is

```
(insert w1 (...  (insert wn EmptySet)))
```

for subsequent transactions.

For an indexed function f: t1 =>* t2 and expressions e1:t1 and e2:t2 that evaluate to v and w respectively, the statement:

```
f [e1] += e2
```

extends the definition of f so that (f v) includes w.

Again, as we shall see later, in update transactions a new f: t1 =>* t2 automatically inherits mappings from an old version unless specified otherwise. To inhibit this, for expressions e1:t1 and e2:t2 that evaluate to v and w respectively, the statement:

```
f [e1] -= e2
```

specifies that (f v) always excludes w. Any other attempt to include w in (f v) is an error.

For an indexed function f: t1 <=>* t2 and expressions e1:t1 and e2:*t2 that evaluate to v and w respectively, the statement:

```
f [e1] += e2
```

extends the definition of `f` so that `(f v)` includes `w`, and `(^ f w)` returns `v`. It will succeed only if `(^ f)` was previously undefined at `w`.

For an indexed function `f: t1 <=>* t2` and expressions `e1:t1` and `e2:*t2` that evaluate to `v` and `w` respectively, the statement:

```
f [e1] -= e2
```

specifies that `(f v)` always excludes `w`, and `(^ f w)` never returns `v`. Any other attempt to define `f` to include this mapping is an error.

For an indexed function `f: t1 *<=>* t2` and expressions `e1:t1` and `e2:t2` that evaluate to `v` and `w` respectively, the statement:

```
f [e1] += e2
```

extends the definition of `f` so that `(f v)` includes `w`, and `(^ f w)` includes `v`.

For an indexed function `f: t1 *<=>* t2` and expressions `e1:t1` and `e2:t2` that evaluate to `v` and `w` respectively, the statement:

```
f [e1] -= e2
```

specifes that `(f v)` always excludes `w` and that `(^ f w)` always excludes `v`. Any other attempt to include this mapping in `f` is an error.

4 Update Transactions

Executing an update transaction against an environment (database) produces a new environment. The transaction itself is a specification of the new environment, based on the old. In principle, this involves specifying *all* the names in the new environment, together with their new bindings. In practice, however, most update transactions express only small changes to the environment. Thus, for convenience, we would like a notation by which we need to specify only the *difference* between the new and old environments.

The entire database can be regarded as a graph. Each internal node in the graph is either a set, a tuple, an ordinary function or an indexed function. The leaves of the graphs are objects of the primitive types and database types. The root of the graph is the database environment itself, which, for uniformity, can be regarded as an indexed function of type `SYM => object`. The special symbol "`db`" in the database environment evaluates to the database environment object itself.

An update transaction is a program that specifies the new graph in terms of the old. At the beginning of the transaction, every node in the graph has a new "shadow" version. Nodes corresponding to indexed functions are open and empty, *i.e.*, with no outgoing edges in the graph. If `e` is an expression that refers to an object in the old graph, then "`new e`" refers to

its new version (thus "**new db**" refers to the new database environment itself). The update transaction contains incremental definitions for the new versions of objects. At the end of the transaction, *i.e.,* when the program has terminated, the new version of each object inherits any old contents that were not incrementally redefined, after which it becomes closed.

The new extension of a type, *e.g.,* **(new (all Student))** is $\perp_{set}$ until the end of the transaction, when it becomes closed, containing all objects of that type that are present in the new version of the database.

4.1 Examples

An update to increase the number of units for the course 6.006 by 3 units:

```
(new CUnits) [ (^ CName "6.006") ] = (CUnits c) + 3
```

The update consists of a single statement that specifies an incremental definition of the new version of the indexed function bound to **CUnits**. The new version differs from the old in that the course referred to by "**^ CName "6.006"**" is now mapped to a number 3 units greater than before.

An update to change the name of student John Xiao to John Zhao:

```
(new SName) [^ SName "John Xiao"] = "John Zhao"
```

An update to increase the units of all courses by 3:

```
{ f c = { (new CUnits) [c] = (CUnits c) + 3 } ;
  mapdo f (all Course) }
```

The first statement defines a temporary function **f** that increases the units of a course by 3. The second statement applies this to all courses (**mapdo** is like **map** in that it applies **f** to each course, but is different in that there are no results to be returned). In our parallel model of computation, all the applications of **f** can be performed in parallel.

An update to remove a grade erroneously recorded for John Zhao in the course 6.001:

```
{ s = ^ SName "John Zhao" ;
  e = hd { e | e <- S-Enroll s;
                    (^ C-Enroll e) == (^ CName "6.001") } ;
  (new EGrade) [ e ] = undef } ;
```

The first two statements locate the enrollment for John Zhao in 6.001. The last statement specifes that the new **EGrade** function will be undefined at that enrollment.

To drop an existing name (say **SNationality: Student => S**) from the top-level database environment:

```
(new db) ['SNationality] = undef
```

"New db" refers to the new database object. The binding indexed by the name SNationality is made undefined.

To introduce a new name into the top-level database environment, or to rebind an existing name to a completely new value, one simply supplies its definition. Examples:

```
{ f s = { c | c <- ^ C-Enroll (S-Enroll s) ;
                  CUnits c > 12 } ;

  (new db) ['SHeavyCourses] = f ;

  (new db) ['CDescription] = empty (Course => S) }
```

This transaction defines two new database functions. SHeavyCourses is an ordinary function on Students that returns the set of courses he takes with greater than 12 units. CDescription is an indexed function on Course objects that returns a string description. CDescription is still undefined on every course— other parts of this update transaction, or later update transactions can fill it in incrementally.

Here is an update that introduces a generally useful function called theEnrollmentFor. Given a student name and a course name, it returns the enrollment corresponding to that student and course:

```
{ theEnrollmentFor sn cn =
      hd { e | e <- S-Enroll (^ SName sn);
                    (CName (^C-Enroll e)) == cn } ;

  (new db) ['theEnrollmentFor] = theEnrollmentFor }
```

Update programs can themselves be stored in the database. For example, here is an update that introduces a function record_grades that can be used in subsequent transactions. It takes a course name and a set of student names and grades, and records the grades for that course.

```
{ record_grades Cn SnGs =
      {  f (Sn,G) = { e = theEnrollmentFor Sn Cn;
                      (new EGrade) [e] = G } ;
         mapdo f SnGs } ;

  (new db) ['record_grades] = record_grades } ;
```

The first statement defines the function value itself, and the second statement records it in the new database.

Another update introducing a function that can be used in subsequent transactions: given a student name and a course name, it adds that enrollment:

```
{  add sn cn =  { s = ^ SName sn ;
                  c = ^ CName cn ;
                  e = make Enrollment () ;
                  (new S-Enroll) [s] += e;
                  (new C-Enroll) [c] += e } ;

   (new db) ['add] = add }
```

Again, the first statement defines the function value itself, and the second statement records it in the new database.

An update introducing a function that, given a student name and a course name, deletes that enrollment:

```
{  drop sn cn =  { s = ^ SName sn ;
                   c = ^ CName cn ;
                   e = theEnrollmentFor sn cn ;
                   (new S-Enroll) [s] -= e ;
                   (new C-Enroll) [c] -= e ;
                   (new EGrade) [e] = undef } ;

   (new db) ['drop] = drop }
```

Note that the way to remove an object from the database is to ensure that there is no function defined on it. The object then disappears from the database.

5 Discussion

5.1 Parallelism

The major issues in designing a database programming language with parallelism are:

- how to specify what can be done in parallel, and
- determinacy, *i.e.,* guaranteeing that the result of the transaction does not depend on the runtime schedule for the parallel parts chosen by the system.

Inspired by the dataflow approach, especially I-structures in Id, our model addresses both issues. Decomposition into parallel parts is implicit— programs can be compiled into dataflow graphs, which constitute a parallel machine language (for an outline of how this is done in Id, see [5] and [11]). The semantics of incremental definition, in which the information content of a function increases monotonically during an update transaction, ensures that the transactions are determinate.

5.2 Historical Data

The model of update just described can be extended to deal with historical data. In Section 2 we modelled the database system as a function from a database and list of transactions to a list of responses. Each transaction was evaluated in the current database environment to produce a new database environment. The old database was discarded.

Instead, we could retain the old database environments, modelling the database system as a function from a *list* of database environments and a list of transactions to the list of responses:

```
Def dbsystem dbs (cons xact xacts) =
        {  resp, db' = eval dbs xact
         In
           cons rest (dbsystem (cons db' dbs) xacts) }

dbsystem (cons empty_db nil) xacts
```

Each transaction thus has the entire history of databases available to it. To make use of this, we need additional notation to specify that an expression (part of the transaction) must be evaluated in an arbitrary previous environment.

First, we need "environment expressions" that specify an environment by indexing into `dbs`, the history of database environments. There are various possibilities for specifying this indexing:

- by absolute position, where the initial, empty database supplied to `dbsystem` has position 0.
- by relative position, with the most recent database environment having position 0.
- by time, assuming that `dbsystem` records the creation time of each database environment.
- by name of creator, assuming that `dbsystem` records the name of the creator of each database environment.
- by arbitrary property, *i.e.,* the most recent database environment in which a given boolean expression evaluates true.
- ...

Once we can specify particular environments, the phrase:

```
with environment-expression
    expression
```

can be used to evaluate an expression within that environment. Thus, we can write queries and updates that depend on any or all previous states of the database.

5.3 Concurrency Between Transactions

The parallelism that we have focused on so far is all within a single transaction. Referring to the database system model of Section 2, the parallelism is within the phrase: `(eval db xact)`. Within `dbsystem`, the result database from one transaction is used as the environment in which to evaluate the next transaction.

This is not to imply that there cannot be any parallelism *between* transactions. First, since a closed database environment is never subsequently modified, a read-only transaction (query) can continue using an old database as long as necessary, without holding up subsequent update transactions. Second, even update transactions can be overlapped: the lenient semantics of our language allows `(eval db xact)` to return a value (the response and the new database) immediately, before the transaction has completed (this behavior is also exhibited by languages with lazy evaluation). This permits `dbsystem` to begin evaluating the next transaction immediately.

A problem arises due to aborted transactions, which can cascade through all subsequent transactions that have already begun executing. To avoid this, one will have to employ the usual solutions: either prevent multiple transactions from overlapping (pessimistic), or allow them to overlap, keeping track of which parts of the database they actually see, so that an abort does not cascade through non-interfering transactions (optimisitic).

5.4 Comparison With Other Approaches

The top-level definition of the database system (`dbsystem`) that we presented in Section 2 is almost identical to other "functional" views ([9], [2]). The differences arise in the meaning of the phrase `(eval db xact)`— what is a database, what is a transaction, and what is the `eval` function?

The approach in [9] can be regarded as a "transaction-as-command" approach. For instance, the database is a bank balance— just a number. Transactions are commands of the form `deposit x` and `withdraw x`. The `eval` function interprets a transaction by examining whether it is a deposit of withdraw command, and constructs the response and new database appropriately:

```
Def eval db xact =
        {  command, val = xact ;
         In
           if command = 'withdraw then
             ("Here is", val), (db - val)
           else if command = 'deposit then
             ("Deposited", val), (db + val)
           else error } ;
```

The problem with this approach is that even though the database system is implemented as a functional program, the transaction language (`deposit` and `withdraw` commands) is a

purely imperative language. Though we have not seen any more complicated examples in the literature, the natural generalization of such a language would be a conventional imperative language, with all the attendant sequentiality and difficult semantics.

The approach in [2] may be regarded as a "transaction-as-function" approach. The database is a data structure, and a transaction is directly a function from a database to a response and new database. Thus, the `eval` function simply applies the transaction to database.

```
Def eval db xact = (xact db) ;
```

The user has the full generality of a functional language in which to specify queries and updates. Being functional, of course there is plenty of parallelism available.

While this approach is very elegant if the database is a tree-like data structure, it becomes very awkward when the database is a general graph (with shared sub-structures). Consider a database that is a list of ten `Course` objects, five of which contain the same `Classroom` object, which in turn contains a number, its seating capacity. Now suppose the seating capacity is to be changed. The update transaction must of course rebuild the `Classroom` object with the new number. But unfortunately, we must also rebuild five `Course` objects to contain the new `Classroom` object, and then we must rebuild the top-level list to contain the five new `Classroom` objects in place of the old. In general, the transaction programmer must explicitly identify and rebuild *every path* from the root of the database down to the "updated" object.

In contrast, our approach can be viewed as an attempt to retain the expressive power, parallelism and declarative nature of the transaction-as-function approach, while achieving the economy of expression of a more imperative approach.

6 Future Directions

The work described here is a preliminary attempt to design a declarative update language within the framework of a functional database system. There are many details to be completed, many issues still to be investigated. As a vehicle for this research, we are constructing a prototype of the system. This is initially implemented in Lisp to take advantage of Lisp's rich programming environment; later we expect to incorporate it into Id and to run it on our dataflow multiprocessor (emulated for now, a real one later). Until we have more experience with writing applications in our prototype, we cannot make a convincing judgment as to whether it is easy or difficult to express updates in this model.

Despite the title of this paper, what we have presented is by no means a formal semantics, and until that event, we cannot possibly be precise in our claims about parallelism, determinacy, *etc.* Once the language has reached a reasonably stable point, we expect to extend the formal semantics of Id, expressed as rewrite rules [11] to cover this database model.

There is a disturbing lack of type-orthogonality in the indexed types— currently, the domain and range of an index type can only be database or primitive types. We are taking

this position currently for pragmatic reasons— it is not clear what it means to index on tuples, sets, nested structures, *etc.*

In our model, currently an object is deleted automatically from the database when it no longer participates in any mappings (no query can be asked of it). The reason for this choice, rather than a command to delete an object directly, was that it is not clear what happens to the mappings in which the object participates. However, removing it from all mappings can be quite tedious to specify. This issue requires more investigation. A more difficult question: when can a *type* be deleted from the database, *i.e.,* what happens to existing objects of that type, mappings on those objects, *etc.*?

The transaction language, like Id with I-structures, is not a purely functional language any more, though it does retain the parallelism and determinacy (and, we claim, declarative nature) of functional languages. The loss of referential transparency is not without cost: it can inhibit certain optimizations that are possible in functional languages. In Id, we have developed a programming methodology whereby we use I-structures only to define new, efficient functional array abstractions, after which the bulk of the program is written functionally [4]. Can such a methodology be extended to deal with our database extensions?

In a related project, we are looking at architectural and low-level programming issues in implementing arbitrary object persistence in the Tagged-Token Dataflow architecture, assuming explicit commands to store and retrieve objects. The gap between that implementation and the database model presented here is yet to be bridged.

References

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

[2] G. Argo, J. Fairbairn, J. Hughes, J. Launchbury, and P. Trinder. Implementing functional databases. In *Proc. ALTAIR-CRAI Workshop on Database Programming Languages, Roscoff, France, September 1987*, 1987.

[3] Arvind and J. D. Brock. Resource managers in functional programming. *Journal of Parallel and Distributed Computing*, 1(1), June 1984.

[4] Arvind and K. Ekanadham. Future scientific programming on parallel machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987.

[5] Arvind and R. S. 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, R. S. Nikhil, and K. K. 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] M. P. Atkinson, K. Chisholm, and W. Cockshott. Ps-algol: an algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1981.

[8] G. Copeland and D. Maier. Making smalltalk a database system. In *Proc. ACM SIGMOD*, page 325, 1984.

[9] P. Henderson. *Purely Functional Operating Systems*, pages 177–192. Cambridge University Press, Cambridge, England, 1982.

[10] R. S. Nikhil. Functional languages, functional databases. In *Proc. Workshop on Persistence and Data Types, Appin, Scotland, August 1985*, April 1987.

[11] R. S. Nikhil. *Id Nouveau Reference Manual: Syntax and Semantics.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[12] R. 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.

[13] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[14] D. 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.