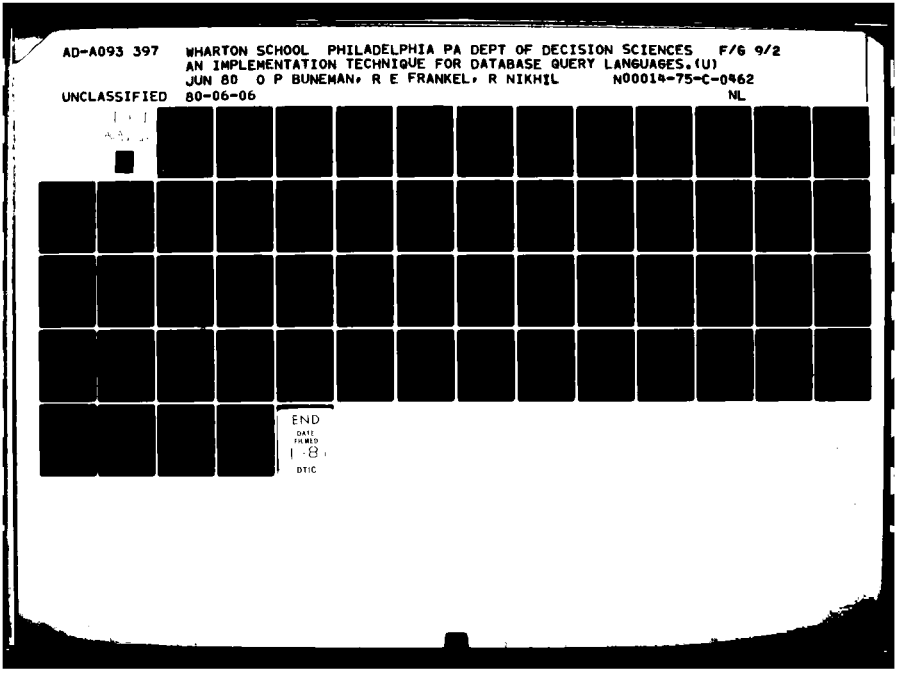
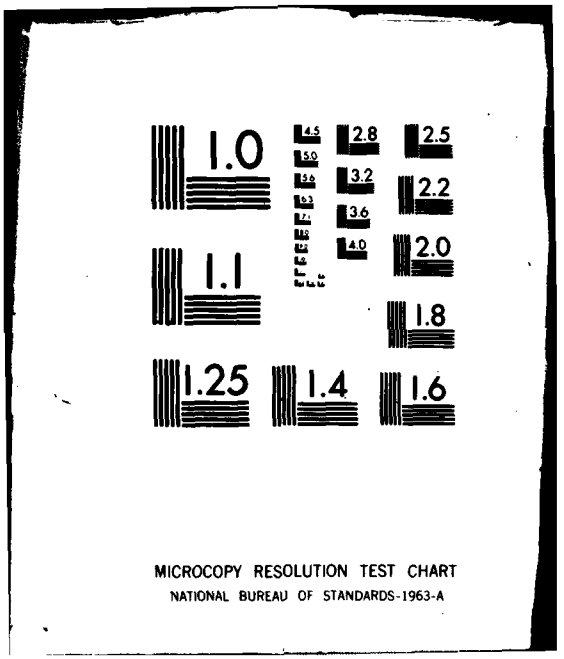

AD-A093 397 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
AN IMPLEMENTATION TECHNIQUE FOR DATABASE QUERY LANGUAGES.(U)
JUN 80 O P BUNEMAN, R E FRANKEL, R NIKHIL N00014-75-C-0462
UNCLASSIFIED 80-06-06 NL
END
DATE FILMED
1-81
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A093397

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 80-06-06	2. GOVT ACCESSION NO. AD A093397
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) AN IMPLEMENTATION TECHNIQUE FOR DATABASE QUERY LANGUAGES.	5. TYPE OF REPORT & PERIOD COVERED TECHNICAL rept. 4/80-3/81 Apr 80 - Mar 81
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) O. Peter Buneman, Robert E. Frankel, Rishiyur Nikhil	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Decision Sciences, The Wharton School, U. of PA, Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research	12. REPORT DATE June 80
	13. NUMBER OF PAGES 54
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DISTRIBUTION UNLIMITED

DTIC ELECTE JAN 5 1981 C

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

query languages, relational databases, database management systems. Codasyl DBMS; applicative programming, coroutines, lazy evaluation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Structured query languages, such as those available for relational databases, are becoming increasingly desirable for all database management systems. Such languages are applicative: there is no need for an assignment or update statement. A new technique is described that allows for the implementation of applicative query languages against most commonly used database systems. The technique involves "lazy" evaluation and has a number of advantages over existing methods:

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

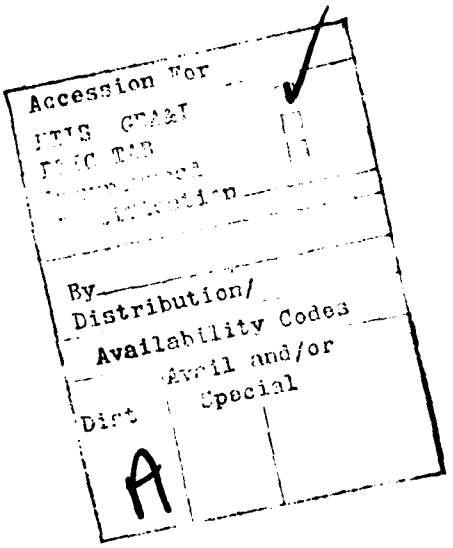
DDC FILE COPY

408757

80 12 30 008

20. (cont.d)

it allows queries and functions of arbitrary complexity to be constructed; it reduces the use of secondary storage; it provides a simple control structure through which interfaces to other programs may be constructed; and the implementation, including the database interface, is quite compact. Although the technique is presented for specific functional programming system, and for a Codasyl DBMS, the techniques are general and may be used for other query languages and database systems.



AN IMPLEMENTATION TECHNIQUE FOR DATABASE QUERY LANGUAGES

O. Peter Buneman
Robert E. Frankel
Rishiyur Nikhil

80-06-06

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

An Implementation Technique for Database Query Languages

O. Peter Buneman

Robert E. Frankel

Rishiyur Nikhil

ABSTRACT

Structured query languages, such as those available for relational databases, are becoming increasingly desirable for all database management systems. Such languages are applicative: there is no need for an assignment or update statement. A new technique is described that allows for the implementation of applicative query languages against most commonly used database systems. The technique involves "lazy" evaluation and has a number of advantages over existing methods: it allows queries and functions of arbitrary complexity to be constructed; it reduces the use of secondary storage; it provides a simple control structure through which interfaces to other programs may be constructed; and the implementation, including the database interface, is quite compact. Although the technique is presented for a specific functional programming system, and for a Codasyl DBMS, the techniques are general and may be used for other query languages and database systems.

Key words and phrases: query languages, database interfaces, applicative programming, coroutines, lazy evaluation.

CR categories: 4.33, 4.12, 4.13, 4.22

Authors' addresses: Peter Buneman and Rishiyur Nikhil, Department of Computer and Information Science, Moore School, University of Pennsylvania, Pa. 19104. Robert Frankel, General Research Corporation, PO Box 6770, Santa Barbara, CA 93111. This work was partly supported by the Office of Naval Research under Contract N00014-75-C-0462 and by a grant from the Digital Equipment Corporation.

1.0 INTRODUCTION

It is generally held that the more powerful programming languages are those that provide the user with functions for the manipulation of "bulk" data: the array operators of APL, and the functionals such as MAPCAR in LISP are well known examples [14, 17]. These operators allow the use of simple expressions for what would otherwise have to be implemented through more complicated code containing explicit control structures such as iteration. The same is true for database query languages. Consider a query that finds the names of employees who are under 30 and are paid more than the average salary for all employees.

```
retrieve NAME from EMPLOYEE where AGE < 30 and
        SALARY > average(retrieve SALARY from EMPLOYEE)
```

There are a number of problems involved in interpreting such a query and the purpose of this paper is to describe an evaluation technique that not only overcomes these problems, but also allows arbitrarily complex queries or functions to be evaluated. The form retrieve..from.. is an expression which, as far as the user is concerned, produces a sequence or set of objects. The use of such expressions greatly simplifies the process of constructing a database query, which would otherwise call for a relatively

complicated, iterative program. (The syntax of the example above is not intended as an example of a specific language; but it is not unlike SEQUEL [6] and some of the more concise query languages for hierarchical or network databases [13].)

To our knowledge there are two methods of implementing such a query that are currently in use. One approach, which we shall call the "immediate" approach, is to instantiate physically the sequences or sets produced by the retrieve expression. A relational data management system would be quite likely to create an intermediate relation for the result of the retrieve SALARY.. expression, and this would have to be temporarily held in secondary storage. The second approach, the "translation" method, is to translate the program into an iterative program which is then run against the database. In this case, the query above would be translated into two iterative loops: one to traverse the EMPLOYEE and print out NAMEs, the other to traverse the EMPLOYEE file and compute the average SALARY. Both approaches have disadvantages. The immediate method requires substantial quantities of temporary working space, and since this is usually available only as secondary storage, results in rather poor execution times for queries. The translation method usually permits only a limited number of program forms, limiting the user to relatively simple queries: those that can be constructed out of iterations over the available data.

Another problem arises with optimization: the crudest implementation of the query above would call for the sub-expression

average(retrieve SALARY from EMPLOYEE)

to be computed for each retrieval of an EMPLOYEE in the surrounding retrieve NAME from .. expression. This is most inefficient, and any reasonable implementation would precompute the average. However, this is still not optimal: in the case that there are no EMPLOYEEs that satisfy the condition AGE < 30, there is no need ever to compute the average SALARY. The optimal iterative program would traverse the employee file looking for an instance that satisfies AGE < 30 and, on encountering the first such instance, compute the average and set a flag indicating that it had been computed. This solution is hardly "structured" code and would probably be overlooked by most applications programmers, however it is an example of the kind of optimization that can in practice lead to substantial savings in the amount of i/o required to evaluate a query.

We shall describe a method for the evaluation of database queries which overcomes the disadvantages of both the immediate and the translation approaches; and which automatically performs the kind of optimization just described. The method exploits "lazy" evaluation techniques. Although these have been widely discussed [4, 16] and implemented [10, 24] for a number of high-level languages, we believe that they will prove important in

the evaluation of database queries, where, because of space limitations, it is physically impossible to perform the in-core manipulation of lists or arrays in the fashion of LISP or APL. Some related suggestions have been made for databases; in particular it has been suggested [22] that relational operators could in certain circumstances be "pipelined", leading to substantial savings in the storage required for intermediate relations.

In order to describe how this method may be implemented, we shall first describe a specific query language: the Functional Query Language, FQL. The syntax of FQL is not intended as an ideal syntax for anyone except, perhaps, mathematicians. It is meant as a formalism for the underlying control structures of database queries. For example, it is an easy matter to represent retrieve..from..where.. in terms of the FQL operators. While it would be possible to interpret a conventional query language directly by using the techniques described in this paper, we shall use the FQL formalism because it appears to us to be more fundamental in that it allows the construction of any program; not a limited set of database access forms. The second section of this paper will therefore describe FQL.

The next section of the paper will specify the basic data structures and procedures for the implementation of FQL. While this will be done without reference to a specific programming language, it is hoped that anyone familiar with programming languages such as Pascal or ADA [1, 27], which have good methods

for defining data types, will have no difficulty in translating our formal expression of the data structures and procedures into working code.

Following the formal description of the implementation, we shall provide some observations on the implementation which may be of practical use to people who wish to build an interpreter along these lines. This includes some details of physical representation of data structures, garbage collection, and type checking. A final section describes some further improvements and optimization techniques that the authors have not yet implemented, but see as valuable extensions to the technique.

2.0 THE FUNCTIONAL QUERY LANGUAGE FQL

FQL is based upon a Functional Programming system as suggested by Backus [2]. Rather than using explicit control structures, a few operators or functional forms are used to construct new functions, or database queries out of existing functions. Consider a very simple FQL query:

```
!EMPLOYEE.*NAME;
```

Informally this reads: take the sequence of all EMPLOYEEs and create a sequence of their NAMEs. The result of typing in this query is that a sequence of names is printed out. Looking more closely at the query, it is built out of two functions, !EMPLOYEE and NAME, which are both functions that are defined in the database. !EMPLOYEE is a function that generates a sequence of EMPLOYEEs; NAME is a function that takes an EMPLOYEE as argument

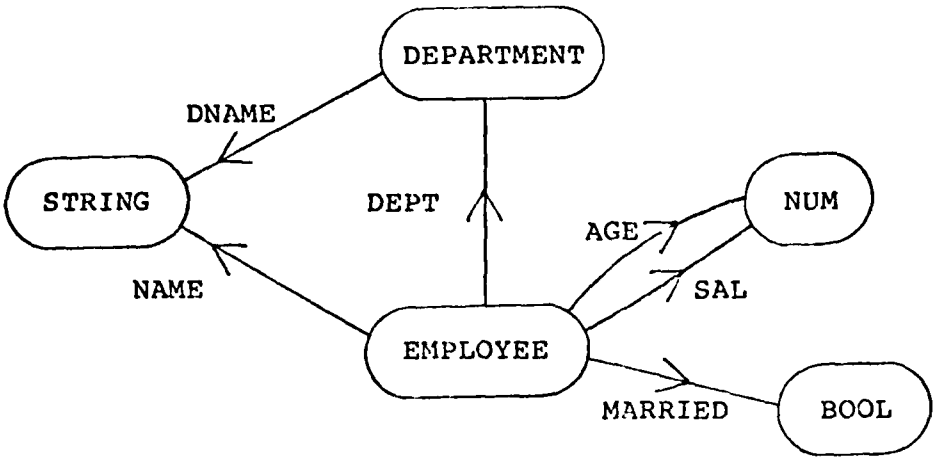
and produces a character string as result. * is an operator: in this query it operates on the function NAME, which works on single EMPLOYEEs, to create a function *NAME that operates on sequences of EMPLOYEEs to produce a sequence of character strings. The MAPCAR function of LISP is a direct analog of this operator. Finally, the symbol "." denotes the composition of the two functions !EMPLOYEE and *NAME into a function that creates a sequence of character strings. Note that the composition is in "reverse polish" notation. This is a deliberate choice resulting from the desire to have queries correspond to the database "access path".

Function	Type
NAME:	EMPLOYEE -> STRING
!EMPLOYEE:	-> *EMPLOYEE
*NAME:	*EMPLOYEE -> *STRING
!EMPLOYEE.*NAME:	-> *STRING

Figure 1. Some simple FQL functions and their types.

Figure 1 shows the functions that constitute this query together with their types. Note that the symbol "*" also has a meaning when applied to a type and denotes a sequence of elements of that type; thus *STRING denotes a sequence of character strings. The type of a function describes the types of the objects in its domain and range. EMPLOYEE denotes a data type, and !EMPLOYEE is a function that generates all instances of that data type; both !EMPLOYEE and the whole query are functions with

no arguments. We may use the notion of data types to describe the formal data model that is used by FQL. A database consists of a collection of types (often called classes) and functions (often called attributes) that have these types as ranges and domains. A somewhat richer version of this model has been proposed by Shipman, who also proposes a query language based upon a few simple iterative constructs. It is possible that the techniques described here could serve to implement Shipman's proposals. The three major data models, relational, hierarchical and network, can all be represented within the functional framework and details of this are discussed in [3, 21]. As an example of the functional model, figure 2 shows a simple database and some of the functions it defines.



Function	Type
!DEPARTMENT:	-> *DEPARTMENT
!EMPLOYEE:	-> *EMPLOYEE
DNAME:	DEPARTMENT -> STRING
NAME:	EMPLOYEE -> STRING
SAL:	EMPLOYEE -> NUM
AGE:	EMPLOYEE -> NUM
MARRIED:	EMPLOYEE -> BOOL
DEPT:	EMPLOYEE -> DEPARTMENT
↑DEPT:	DEPARTMENT -> *EMPLOYEE
↑NAME:	STRING -> *EMPLOYEE

Figure 2. A simple database and the functions it provides.

This figure describes a database containing five types of which two, EMPLOYEE and DEPARTMENT are specific to the database, and three, NUM(eric), STRING and BOOL(ean), are always defined. ↑DEPT and ↑NAME are inverse functions: database systems have sophisticated mechanisms for maintaining inverses. In a Codasyl system, for example, ↑DEPT would be implemented through a set, and ↑NAME through a hash function. In general we do not expect all database functions to have efficiently implemented inverses.

In order to complete the description of FQL, we must introduce a few more operators and another method of extending types. The notation [NUM,STRING] denotes the type of tuples of objects of type NUM and STRING. The same notation applied to functions denotes a function that yields a tuple. For example, [NAME, SAL] is a function of type EMPLOYEE -> [STRING, NUM]. The function [NAME, SAL] applied to an EMPLOYEE produces a [STRING, NUM] tuple.

```
!EMPLOYEE.*[NAME, DEPT.DNAME];
```

is a query which produces the names and department names of all employees. The result of this query is of type *[STRING, STRING].

Another useful operator is restriction, denoted by "|". If p is a predicate (a boolean valued function) then |p restricts sequences by that predicate. For example, |MARRIED is of type *EMPLOYEE -> *EMPLOYEE, and will filter out, from any sequence of employees, those that are married.

```
!EMPLOYEE.|MARRIED.*[NAME, SAL];
```

is a query whose result is of type *[STRING, NUM] and which provides the name and salary of married employees.

The ability to define new functions is extremely useful. The average salary for a department may be defined as a function of type DEPARTMENT -> NUM as:

```
AVESAL = ↑DEPT.*SAL.AVERAGE;
```

and the average function may be in turn defined as a function of type *NUM -> NUM as:

```
AVERAGE = [/+, LEN].DIV;
```

where /+ adds up a sequence of numbers (perverting an APL notation), LEN computes its length and DIV divides two numbers.

This last definition illustrates a number of points. In the first place FQL functions all take one argument and return one result. The effect of multiple arguments is achieved through the use of tuples and selectors. The selectors are the functions #1, #2,... which respectively select the first, second.. members of a tuple. Second, it is possible to define arbitrary functions in FQL, and since the basic list processing functions are available, together with recursion, FQL provides the same computational power of pure LISP. Third, variables are not needed in function definition: this turns out to be a considerable advantage in evaluating FQL expressions as the overhead of preserving "environments", which is substantial in many lazy evaluation schemes, is greatly simplified. Fourth, it is possible to define functions and to evaluate FQL expressions without reference to any database. For example,

```
[1, 2].+;
```

is a valid FQL expression and will be evaluated. It is surprising how many database query languages fail to provide the power to evaluate simple arithmetic expressions, even though they can perform arithmetic when it is embedded in a database query. Note that in FQL, for uniformity, the numeric constants 1,2... and the boolean and string constants are treated, as constant functions. Like the function !EMPLOYEE, their result is independent of their argument. During the evaluation of a query, the result of !EMPLOYEE will not change; however, over a longer time scale, we

may expect the function !EMPLOYEE to be considerably less "constant" than the number 1.

Further examples of FQL function definitions, including some that exploit recursion, are to be found in [3, 9]. The appendices to this paper give a specification of the FQL syntax and a comprehensive list of built-in functions. A few examples are given here in order to demonstrate the use of the operators described above in some simple database queries.

1. The names and department names of all married employees.
   !EMPLOYEE.|MARRIED.*[NAME, DEPT.DNAME];
   The output of this query is a sequence of pairs of strings (type *[STRING, STRING]).

2. The name of each department together with the names of all married employees in that department.
   !DEPARTMENT.*[DNAME, ↑DEPT.|MARRIED.*NAME];
   Unlike the previous example, the output from this query is "hierarchical", of type *[STRING, *STRING]. Each department name will be followed by a sequence of employee names. Note that ↑DEPT generates a sequence of employees for each department.

---

1. The notation used in [3, 9] is slightly different. The inverse of a function, DEPT say, which is denoted here by ↑DEPT, was denoted by !DEPT. The "#" sign is used here for a selector, rather than a numeric constant. Also, in the version of FQL currently being implemented, there is no need to declare data types when defining a query or function.

3. The names of all married employees who earn less than the average salary for all employees.

```
!EMPLOYEE.|MARRIED.|([SAL,!EMPLOYEE.*SAL.AVERAGE].LT).*NAME;
```

Note that the double restriction could be replaced by a single restriction and a conjunction.

4. The names and salaries of all employees in the sales department.

```
!EMPLOYEE.|([DEPT.DNAME, 'SALES'].EQ).*[NAME,SAL];
```

The output of this is of type *[STRING, NUM].

5. The names and salaries of all employees in the sales department.

```
!DEPARTMENT.|([DNAME, 'SALES'].EQ).*↑DEPT./CONC.*[NAME,SAL];
```

A possibly more efficient version of the previous query. Notice that after *↑DEPT we have a sequence of sequences of employees. CONC is a function that concatenates two sequences; /CONC, the reduction of CONC over a sequence, "flattens" a sequence of sequences (note the analogy with /+). The possibility of automatically performing the transformation from example 3 to this example is briefly discussed later.

6. Does a department have more unmarried than married employees?

```
MOREMARRIED = ↑DEPT.[|MARRIED.LEN, |(MARRIED.NOT).LEN].GT;
```

This is a function of type DEPARTMENT -> BOOL that defines a predicate on departments. (Recall that LEN computes the length of a sequence.)

7. The names of such departments

```
!DEPARTMENT.|MOREMARRIED.*DNAME;
```

Finally, we should introduce one further operator, which although its use in database queries appears limited, is extremely important in the internal definition of the interface between FQL and the database access subroutines. The operator & takes a function f and creates a sequence-producing function which generates the sequence obtained by applying successive powers of f to an argument. For example,

```
0.&add1;
```

generates the sequence 0, 1, 2 ... and

```
[0, 1].&[#2 +].*#1
```

generates the Fibonacci sequence. The second example illustrates the use of the selectors #1, #2 to access components of a tuple. It is a property of the implementation that both sequences are produced indefinitely and will in practice only terminate on arithmetic overflow or when interrupted by the user.

To recapitulate, a database is viewed as a collection of functions over various data-types. Five operators, compose (.), tuple ([...]), extend (*), restrict (|) and generate (&) are available for combining the database and built-in functions into new functions and queries. Using the notation $*\alpha$ to denote the type of a sequence of some type α, and $[\alpha_1,\alpha_2,\ldots,\alpha_n]$ to denote the type of tuples of $\alpha_1,\alpha_2,\ldots,\alpha_n$, we may summarize the types of functions produced by these operators as follows:

1. Compose. If f and g are such that $f: \alpha \rightarrow \beta$ and $g: \beta \rightarrow \gamma$ then $f.g: \alpha \rightarrow \gamma$.

2. Extend. If $f: \alpha \rightarrow \beta$ then *f operates upon a sequence of these types; i.e., $*f: *\alpha \rightarrow *\beta$.

3. Restrict. If p is a predicate over α (i.e., $p: \alpha \rightarrow$bool) then $|p: *\alpha \rightarrow *\alpha$.

4. Tuple. If $f_1: \alpha \rightarrow \beta_1$, $f_2: \alpha \rightarrow \beta_2 \ldots$ $f_n: \alpha \rightarrow \beta_n$ then $[f_1, f_2 \ldots f_n]: \alpha \rightarrow [\beta_1, \beta_2 \ldots \beta_n]$.

5. Generate. If $f: \alpha \rightarrow \alpha$ then $\&f: \alpha \rightarrow *\alpha$.

3.0 IMPLEMENTATION STRUCTURES

It should be apparent from the previous section that the main problem in implementing FQL, or a language like it, will be with the representation of sequences. For reasons of space, sequences cannot, in general, be manifested as in-core lists or arrays; and their representation as files in secondary storage is needlessly time consuming.

The solution to this problem is to represent sequences as coroutines that provide their successive members as these are required by some calling program. The technique was originally suggested by Landin [16] and several programming systems [5, 10] have exploited the technique in various ways. We shall use the term stream for this representation of a sequence: Burge [4] gives an excellent description of how the iterative control structures of conventional programming languages can be represented as operators on streams.

We shall represent a stream as a two component data structure whose first component is the first member of the stream, and whose second component is a suspension. Informally, a suspension is a "promise" to create another stream. It consists of a function f and an argument a; the application of f to a creates another stream, the "tail" of the original stream. A procedure such as PRINT, which has to traverse a stream, must do so by repeatedly evaluating suspensions.

Suspensions have been introduced for the purpose of representing streams, but they can also be used in other places to cut down unnecessary evaluation. An example is the familiar "if p and q then.." in which the evaluation of q is only required if the evaluation of p yields TRUE. In this case, p and q may be represented by suspensions which are evaluated, if needed, by and. Since we are dealing with a side-effect free system, the usual arguments for (or against) requiring that q be evaluated do not apply. The technique of delaying the evaluation of expressions until their values are needed is known as "lazy" evaluation: it is particularly relevant to database queries because the alternative of "prompt" evaluation may call for unnecessary and time-consuming accesses of secondary storage. We shall therefore use it not only for the representation of streams but also for the basis of evaluation of any FQL expression.

3.1 The Internal Representation Of FQL Expressions.

It will be convenient to use a uniform physical structure both for the representation of an FQL expression and for the intermediate data structures created during the evaluation of such an expression. We use the notation $\{\underline{x}_1, \underline{x}_2, .. \underline{x}_n\}$ for a structure containing the finite sequence of objects $\underline{x}_1, \underline{x}_2, .. \underline{x}_n$. The structure may be implemented as a linked list or an array, depending on what is most expedient in the chosen programming environment. Operators must be available for selecting the ith component of such a structure and for constructing it from its components. It will also be convenient to describe the role of a structure in the evaluation process by an initial letter between the braces; for example a suspension consisting of the function $\underline{f}$ and the argument $\underline{a}$ will be denoted by $\{U\ \underline{f},\ \underline{a}\}$. There will in fact be four types of structure used in the evaluation process; their roles should become apparent as the evaluation technique is developed, but we shall briefly introduce them here:

$\{U\ \underline{f},\ a\}$ denotes a <u>suspension</u> (U for Unevaluated). The components of a suspension are a function $\underline{f}$, and an argument $\underline{a}$.

$\{F\ \underline{f}_1, \underline{f}_2, .. \underline{f}_n\}$ denotes a <u>functional form</u>. This is a compound structure representing a function. In general, the first component, $\underline{f}_1$, of a functional form will denote an operator and the remaining components will represent functions or functional forms.

{T $\underline{x}_1$, $\underline{x}_2$,.. $\underline{x}_n$} denotes a tuple. A tuple is a structure that cannot itself be further evaluated, and may serve as an argument in a suspension. The components of a tuple may themselves be suspensions, and are subject to further evaluation. Note that we have used the word "tuple" ambiguously to refer to a data type, an operator and to an internal data structure.

{S $\underline{x}$, $\underline{u}$} denotes a stream. This is a special two-component tuple whose second member is always a suspension. This is used for the representation of very long (or "infinite") sequences, typically those in the database.

Not all these distinctions are required; however a field or variant type statement describing the role of a structure will be found invaluable for debugging purposes.

The result of evaluating an expression must be a printable object. The printable objects are, by definition, the printable atoms, such as character strings and numbers; and streams and tuples of printable objects. For example, printing selected fields from all the records in a given class will call for the creation of a stream of tuples of printable atoms.

An FQL expression is internally represented as a function or functional form. If $\underline{F}$ is a (built-in or user-defined) FQL function symbol, we shall use $\underline{f}$ to denote its internal representation and, more generally, $\underline{e}$ will denote the internal

representation of an FQL expression E. The rules for the translation of FQL into an internal representation are quite simply:

F	-> f	for any user-defined or built-in function symbol F (this includes numbers and string constants.)
$E_1.E_2$	->	{F compose, e_1, e_2}
$[E_1,E_2,..E_n]$	->	{F tuple, e_1, e_2,..e_n}
#i	->	{F select, i}
*E	->	{F extend, e}
|E	->	{F restrict, e}
&E	->	{F generate, e}

where compose, tuple, select etc. are internal references to functions whose implementation we shall shortly describe. As an example of the translation,

[1,2].+

is represented as

{F compose, {F tuple, 1, 2}, plus}

What has been done here may simply be regarded as a syntactic transformation of the expression. It also represents a minimal form of "compilation" in which symbols are replaced by internal pointers, and expressions by data structures. As we shall indicate later, compilation involves somewhat more than this.

3.2 Evaluation Of An Expression

This structure produced by compilation still represents a function; and in order to create an object that may be evaluated, we first embed it in a suspension, giving it a dummy argument ø:

{U {F compose, {F tuple, 1, 2}, plus}, ø}

In this suspension, the value of the argument will be irrelevant because the function is constant, i.e it ignores its argument.

Now the process of evaluating a suspension may be performed through a very simple reduction rule, which we shall call evalstep

evalstep({U f,a}) =

f	if f is atomic
apply(f,a)	if f is built-in
{U def(f),a}	if f is user-defined
{U e_1,{T a,e_2,..e_n}}	if f = {F e_1, e_2,..e_n}

The effect of eval-step can be understood in terms of the data structure used to represent an FQL function f. The four lines of evalstep deal with four possible function types:

Atomic (line 1). Atomic functions include numbers, quoted strings and the boolean values true and false. Treating these as functions is merely a syntactic convenience. It would be quite possible to add an explicit apply operator in the source language (FQL) and treat them as objects rather than functions.

Built-in (line 2). This includes both the functions (plus, and etc.) and the operators (compose, restrict. etc.). A built-in function has some reference or index that is understood by apply.

User-defined (line 3). Internally, a user-defined function has a name, which may be used for debugging or for printing out definitions of other functions; and a def, which is the definition of this function. This indirect reference is essential in interpreted systems in order to allow for the dynamic definition and redefinition of functions by the user.

Functional form (line 4). Consider a simple suspension such as {U {F extend, f}, s}, where f is some function that may be applied to elements of the stream s. extend is, like MAPCAR in LISP, an operator, which takes as arguments a function and a stream and applies the function to each member of the stream. The form {F extend, f} represents a function in which extend has been given one of its arguments. It is a "partial application" [5] or "Curried" function [4]. When this form is in turn applied to a stream, we transform the suspension {U {F extend, f}, s} into the suspension {U extend, {T f, s}} in which the function f and the stream s are combined into an explicit argument for extend. The final line of the definition of evalstep

describes a generalization of this transformation that will also cope with forms such as {F compose, e_1, e_2}.

The function evalstep performs a single reduction step and may not completely evaluate a form; it could return another suspension. However, it is usually called in situations where it is required to return an explicit result. We therefore use evalstep to construct a function eval that will evaluate until something other than a suspension is returned.

eval(u) =

u	if u is not a suspension
eval(evalstep(u))	otherwise

In practice, eval would be implemented as an iterative program. We now turn to the the operators and some of the built-in functions. For each such function or operator f, we must define the result of apply(f,x):

apply(compose, {T a, f, g}) =
{U g, {U f, a}}

Composition is in reverse polish order.

apply(tuple, {T x, f_1,f_2,...}) =

{T {U f_1,x}, {U f_2,x}...}

Note that the members of a tuple are not evaluated by tuple.

apply(select, {T x, i}) =

iselect(eval(x), i)

where iselect is the internal function that finds the ith element of a tuple. The argument x must be evaluated in order to obtain a tuple, but the selected element itself is not evaluated.

Before looking at the other (stream processing) operators, let us examine the evaluation of our simple FQL query: [1,2].+ . As described above, this is translated into the suspension:

{U {F compose, {F tuple, 1, 2}, plus}, ø}

This is given to eval which, after one application of evalstep creates the suspension:

{U compose,{T ø, {F tuple, 1, 2}, plus}}

The next application of eval-step will apply the operator compose to give the suspension:

{U plus,{U {F tuple, 1, 2}, ø}}

At this point apply(plus,s) is called. Now the internal function for plus, call it iplus, is a two-argument function which will require both of its argumcnts evaluated, this can be done by apply:

```
apply(plus, u) =
     iplus(eval(iselect(1, x)),
          eval(iselect(2, x)))
     where x = eval(u)
```

In our example, u is

{U {F tuple, 1, 2}, ∅}

which evaluates to:

{U tuple, {T ∅, 1, 2}}

which in turn evaluates to:

{T {U 1, ∅}, {U 2, ∅}}

by the definition of tuple. The evaluation of the components of this tuple respectively yield 1 and 2; iplus now has both its arguments, and yields 3, which completes the example. At first sight this appears to be a somewhat laborious method of adding two numbers, but this is largely an illusion resulting from the syntactic representation of data structures. The physical reduction of the original expression has involved a only a small number of list or array operations that do not involve a great deal of processing. However, the main point is that the efficiency and power achieved in a database environment, where the importance of reducing i/o usually dominates other considerations, is greatly enhanced.

3.3 Stream Processing Operators

At the beginning of this section, we mentioned that a stream is a two-component structure denoted by {S x, u} whose second

component u is a suspension. The first component x, which may or may not be a suspension is termed the head of the stream. The result of evaluating u must be another stream, which is called its tail. The stream operators may now be defined:

apply(extend, {T s, f}) =
{S {U f,x}, {U extend, {T u, f}}}
where {S x, u} = eval(s)

In other words, the argument for extend must first be evaluated until it is a stream; a new stream is then created whose head is a suspension. There is a good reason for leaving the head unevaluated: the function f may call for a database access, and this may be unnecessary. For example, one might only want to retrieve the third record in a sequence, given a stream of indices for those records, in which case retrieving the first two records would be pointless.

apply(generate,{T x, f}) =
{S x, {U generate, {T {U f, x}, f}}}

Although the expansion of a stream produced by generate appears to give rise to deeply nested suspension, this does not, in practice, happen, for reasons that we shall give shortly.

apply(restrict,{T s, p}) =

{S x, {U restrict, {T u, p}}} if eval({U p, x}) is true

{U restrict, {T u, p}} if eval({U p, x}) is false

where {S x, u} = eval(s)

None of these stream processing operators has any provision for the termination of streams. There is a special atom end that is recognized by the operators extend and restrict. For example, extend will always map end into end, irrespective of the function f that it is extending. Under certain circumstances, the operator generate (or a variant of it) will produce end; this is useful in building the database interface. Note that the normal list terminator NIL is therefore equivalent to the FQL expression &end. In most database queries, stream termination is controlled by the underlying streams defined by the database. It is a simple matter to extend FQL with operators that correspond to the iteration control statements, such as while..do.. or repeat..until.. of conventional programming languages. See Burge [4] for details.

In order to complete our discussion of evaluation, we should explain how the result of a query, or any expression, is printed out. It is in fact the function print that drives the whole evaluation. print is initially given a suspension and must call for its complete evaluation.

print(s) =

iprint(x) if x is atomic

```
for i = 1 to n do print(x_i)   if x = [T x_1, x_2 .. x_n]
sprint(x)      if x is a stream
where x = eval(s)
```

We have assumed an internal function iprint that can deal with the atomic types: STRING, NUM and BOOL. sprint is another iterative procedure that traverses a stream (using tail) and recursively calls print on the successive heads until the end-of-stream marker end is encountered.

3.4 The Database Interface

In this section, we shall describe some of the details of an interface to a Codasyl system. Although this does not automatically generalize to all database management systems, most of the systems that the authors have examined (including the relational systems) have, at some internal level, subroutines or control blocks that support record-at-a-time access to the physical database. In our initial implementation of the language we were particularly fortunate to have at our disposal a Codasyl system, SEED [11], that provides, together with the DBTG [6] standard data manipulation routines, a set of subroutines that answer questions about the database schema. This makes the process of checking a query extremely simple: in other systems one might have to resort to reading a (parsed or unparsed) version of the Data Definition Language.

To our set of primitive types, NUM, STRING and BOOL, we shall add two new internal types, DBID (Database Identifier) and CP (Currency Pointer). A DBID is any name associated with the schema: for a Codasyl system, a DBID may be the name of a record, item or set. A CP is a physical record address. According to the DBTG specifications, there is a group of FIND subroutines that operate on a global set of currency pointers. Our first task is to confer some "functionality" on these subroutines. Consider the problem of representing a set traversal. One would normally establish a currency pointer to the owner record and then do a "find-first-in-set" to obtain the first member record. A "find-next-in-set" procedure is then repeatedly invoked to obtain currency pointers to successive member records of the set. An error flag is set when the set is exhausted. These two FINDs may be represented by two functions. findfs (find-first-in-set) is of type [CP, DBID]->CP. Given a CP for an owner record and a DBID for the set, it produces a currency pointer for the first member record of the set or end if the set is empty. findns (find-next-in-set) is also of type [CP, DBID]->CP. It is given a CP to a member record and the DBID for the set, and returns the CP for the next member of the set, or end if the set is empty. With these definitions, it is a simple matter write a function genset

---

The FIND commands may all be variants of one subroutine. Date [7] and Ullman [25] present concise descriptions of these.

in FQL, of type [CP, DBID]->*CP, that generates the stream of all currency pointers to records owned by the given record in the given set. genset is:

[findfs, #2].&[findns, #2].*#1

The selector #2 in the second position of the second tuple preserves the DBID, a set identifier, for successive applications of findns.

To be more precise about how FQL code is interpreted, we first add to our translation table of Section 3.1, the rules:

↑D	->	{F inverts, D}	if D is a set DBID
D	->	{F seto, D}	if D is a set DBID
D	->	{F item, D}	if D is an item DBID
↑D	->	{F invertk, D}	if D is a CALC key DBID
!D	->	{F instances, D}	if D is a record DBID

Type checking should be done to ascertain that the operators have been used correctly. We may now exploit the function genset, defined above, in the application of inverts:

apply(inverts, {T c, d}) =
    {U gensets, {T c, d}}

where c is a CP and d is a DBID. The code for gensets can also be represented, somewhat more efficiently, as an internal function.

apply(inverts, {T c, d}) =

```
{U generate, {T {U findfs, {T c, d}},
                {F findns, d}}}
```

However, it should be noted that once the data access routines have been expressed applicatively, the database interface can, to a great extent, be written in FQL.

Traversing a record class (instances) is done by

```
apply(instances,{T x, d}) =
    {U generate, {T {U findfc, d},
                    {F findnc, d}}}
```

Where findfc, of type DBID->CP, finds CP for the first record in a class and findnc, of type [CP, DBID]->CP, finds CPs for successive members of that class. Since {F instances, D} is a constant function, the argument, x, is ignored. Doing a CALC key access to a sequence of records with the same key may be done in a very similar fashion. The remaining functions, seto (for set ownership) and item (for access to items in a record) correspond directly to the database subroutines.

At this point it may be illuminating to trace through some key points in the evaluation of the query of Section 1.0,

```
retrieve NAME from EMPLOYEE where AGE < 30 and
        SALARY > average(retrieve SALARY from EMPLOYEE)
```

This is expressed in FQL by the expression

!EMPLOYEE.|([[AGE, 30].LT, [SAL, AVESAL].GT].AND).*NAME;

i.e. !EMPLOYEE generates the stream of all employees, |(..) restricts this to the stream of just those employees that satisfy the predicate (call it P) within (..), and *NAME converts this to the stream of names of these employees, which is to be printed. As in Section 2, AGE and SAL are functions that, when applied to an employee, return that employee's age and salary respectively, and AVESAL is another FQL expression that finds the average salary of all employees.

The top-level "PRINT" drives the evaluation of this expression by embedding its internal form in a suspension with a dummy argument, and calling eval, expecting a number,string, boolean, tuple or stream:

{U {F compose, {F compose, e_{EMP}, $e_{|P}$ },
{F extend, name}, ∅}

where e_{EMP} is {F instances, EMPLOYEE}, the internal form of !EMPLOYEE, and $e_{|P}$ is the internal form of |P. After a few evalsteps, this is transformed into

{U extend, {T e_1, name}}

where e_1 is {U {F compose, e_{EMP}, $e_{|P}$ }, ∅}. evalstep then applys extend, which evaluates e_1, expecting a stream. After a few evalsteps, e_1 becomes

{U restrict, {T e_2, e_P}}

where e_2 is {U e_{EMP}, ∅}, which is {U {F instances, EMPLOYEE}, ∅}, and e_P is the internal form of the predicate P. restrict, via apply, evaluates e_2 to get a

stream {S $\underline{x}_1$, $\underline{t}_{EMP}$}, the head of which is the first employee $\underline{x}_1$, and the tail of which is a suspension $\underline{t}_{EMP}$ that represents the stream of remaining employees. restrict now embeds $\underline{x}_1$ in a suspension with $\underline{e}_P$, the predicate P, and evaluates it, expecting a boolean value indicating whether the employee satisfies P; i.e. it evaluates

{U {F compose, {F tuple, $\underline{e}_{AGE}$, $\underline{e}_{SAL}$}, and}, $\underline{x}_1$}

where $\underline{e}_{AGE}$ and $\underline{e}_{SAL}$ are the internal forms of [AGE,30].LT and [SAL, AVESAL].GT respectively. After further evalsteps, this becomes

{U and, {U {F tuple, $\underline{e}_{AGE}$, $\underline{e}_{SAL}$}, $\underline{x}_1$}}

and evaluates its argument to first get a tuple

{T {U $\underline{e}_{AGE}$, $\underline{x}_1$}, {U $\underline{e}_{SAL}$, $\underline{x}_1$}}

and then evaluates the first component of the tuple, {U $\underline{e}_{AGE}$, $\underline{x}_1$} to get a boolean result. Suppose this is true (i.e. the age of employee $\underline{x}_1$ is less than 30). and then evaluates the second component {U $\underline{e}_{SAL}$, $\underline{x}_1$}. Suppose this too returns true (note that this is the first time that the average salary is evaluated) and thus returns true to restrict.

restrict now has the first element of the stream; it constructs the stream {S $\underline{x}_1$, $\underline{t}_1$}, and returns it to extend, where $\underline{t}_1$ is {U restrict, {T $\underline{t}_{EMP}$, $\underline{e}_P$}}, in which $\underline{t}_{EMP}$ is the tail of the stream of employees and $\underline{e}_P$ is the predicate P.

Having got a stream, extend returns a new stream to PRINT, {S {U name, $\underline{x}_1$}, {U extend, {T $\underline{t}_1$, name}}}.

PRINT now has a stream. It evaluates the head {U name, x_1} and prints it out- the name of the first employee satisfying the predicate. It then turns its attention to the tail, a stream which is still a "promise" to extend name to a stream, which is still a "promise" to restrict, with predicate P, another stream, which in turn is still a "promise" to get the stream of remaining employees. The evaluation of this tail applys extend, which evaluates t_1, which applys restrict, which evaluates t_{EMP}, which returns a stream with the second employee x_2 as its head,...and so on.

Again, we should point out that although the trace of evaluation of this query appears quite complicated, this results from our syntactic representation of data structures with shared sub-structures: the physical processing mostly involves relatively simple list processing operations. In our current implementation, less CPU time is spent is spent on the compilation and interpretation of a query such as this, than is spent in the database access routines. In any case, the perceived delays almost always are a result of i/o waits.

4.0 IMPLEMENTATION DETAILS

In the previous section we gave an abstract description of the code for a FQL interpreter. If this were to be taken literally and transcribed into some programming language, the resulting code would be extremely inefficient and space consuming. For example, a programmer will discover that there are a number of places where, for the sake of simplicity, we have needlessly constructed suspensions. In this section we shall describe a number of important implementation details that lead to a dramatic improvement in efficiency of interpretation without compromising the benefits of lazy evaluation.

4.1 Avoiding Repeated Evaluation

In our abstract description of the interpreter we used a purely applicative formalism, in which the entire control structure was defined through function calls. Taken literally, i.e. if the "result" returned by eval is always a physically "new" structure, this would require the repeated evaluation of the same expression. Consider a simple query:

[3,4].[+,#2]

If the evaluation of this is traced, there are points at which evaluation of the two suspensions:

{U plus, {U {F tuple, 3, 4}, ø}}

and

{U select, {T {U {F tuple, 3, 4}, ø}, 2}}

is called for. The (physically) common sub-expression

{U {F tuple, 3, 4}, ∅} would be evaluated twice. The situation is very much worse in the case of queries such as the Fibonacci sequence (Section 2), where producing the nth element will require order n calls to evalstep.

However, if we realize that eval performs a reduction, from one expression to another, "simpler" expression that is always mathematically equivalent to the first, we can make eval replace the suspension it is evaluating with its value. The code for eval therefore has the "benign side-effect" [10] of modifying its argument; when this expression is accessed from other surrounding expressions, it will be found to be evaluated already. To illustrate this technique, consider a PASCAL implementation of the FQL interpreter. It is convenient to describe all the internal data types of the interpreter as variants of one major type, OBJECT:

```
type OBJECT = ↑OBCELL;

     OBCELL = record
                case OBTYPE = (INDIRECTION,
                               ATOM,TUPLE,STREAM
                               SUSPENSION  ... ) of
                  INDIRECTION: (CONTENTS: OBJECT)
                  ATOM       : ...
                  TUPLE      : ...
                  STREAM     : ...
                  SUSPENSION : ...
                   :
                   :
              end;
```

The definition of eval in section 3 suggests a function whose header is:

```
function EVAL(O:  object):  object;
```

Instead, we use the code:

```
procedure EVAL(var O: object);
  var RES: object;
  begin
    while O↑.OBTYPE = SUSPENSION do
      begin
        RES := EVALSTEP(O)
        O↑.OBTYPE := INDIRECTION;
        O↑.CONTENTS := RES;
        O := RES
      end
  end;
```

The "var" in the parameter description of EVAL indicates a call by reference. The pointer O is thus physically reset to point to the result, and the original object that O pointed to is changed to an indirection object. Subsequent calls (if any) to EVAL with the same original object as argument will immediately find a pointer to the evaluated result. The effect of this form of EVAL is also to replace any component of a tuple that contains a suspension with the value of that suspension. As a result, the interpretation of expressions involving & (such as the FQL expression for the Fibonacci sequence) do not create increasingly nested suspensions, but involve a constant number of evaluation steps for each application of generate. Turner [24], in describing a graph reduction technique for the interpretation of applicative languages through the use of combinators, reports a similar need for an indirect reference.

There is an important exception to this rule of replacing suspensions with their values. If a stream is evaluated in this fashion, then each time the second component u of a stream {S x, u} is evaluated, the suspension u will be replaced by a

pointer to another stream element. Thus, as the stream is traversed, it is "solidified" into a linked list. Although this is acceptable in some programming environments, it cannot happen in a database application because the representation of a database sequence as an in-core list will be physically impossible. The solution to this problem is to ensure that all functions and operators use the special function tail as the only method of traversing a stream. tail copies the object corresponding to the second component of the stream before calling eval, thus shielding the original object from the "benign" side-effect, leaving it intact.

4.2 Garbage Collection

The implementation we have described clearly calls for the dynamic creation of new structures such as streams, tuples and suspensions; and while programming languages such as PASCAL and PL/1 provide automatic allocation methods, they do not automatically detect that a structure is no longer referenced, and that the storage it occupies may be de-allocated. It is up to the programmer to decide at what point in the program a structure is no longer referenced and to issue an explicit instruction (FREE or DISPOSE) to return the storage occupied by the structure to the run-time system. It is theoretically impossible in this evaluation scheme to predict when a structure is no longer referenced. Some form of dynamic garbage collection mechanism is required. Fortunately all structures created by this evaluation

pointer to another stream element. Thus, as the stream is traversed, it is "solidified" into a linked list. Although this is acceptable in some programming environments, it cannot happen in a database application because the representation of a database sequence as an in-core list will be physically impossible. The solution to this problem is to ensure that all functions and operators use the special function tail as the only method of traversing a stream. tail copies the object corresponding to the second component of the stream before calling eval, thus shielding the original object from the "benign" side-effect, leaving it intact.

4.2 Garbage Collection

The implementation we have described clearly calls for the dynamic creation of new structures such as streams, tuples and suspensions; and while programming languages such as PASCAL and PL/1 provide automatic allocation methods, they do not automatically detect that a structure is no longer referenced, and that the storage it occupies may be de-allocated. It is up to the programmer to decide at what point in the program a structure is no longer referenced and to issue an explicit instruction (FREE or DISPOSE) to return the storage occupied by the structure to the run-time system. It is theoretically impossible in this evaluation scheme to predict when a structure is no longer referenced. Some form of dynamic garbage collection mechanism is required. Fortunately all structures created by this evaluation

technique are acyclic: they do not contain circular chains of pointers. A simple reference-count garbage collection scheme [9] may therefore be employed. Each structure contains a reference-count field that counts the number of other structures that directly refer to it. Should this count fall to 0, the space occupied by the structure may be returned to free storage, and the reference counts of all structures referenced from this structure may be recursively reduced. The advantage of this scheme is that it works synchronously: garbage collection does not, in general, cause large-scale disruptions in program execution.

As a matter of practical importance to PASCAL programmers: the DISPOSE command is often left unimplemented. It is therefore necessary to keep one or more free lists of unused structures. It considerably simplifies the code, and the efficiency of storage allocation, if all such structures are instances of just one data type. Variants of this type are used to implement the different kinds of structure required by this evaluation scheme.

4.3 Compilation

The only form of compilation described so far is the simple translation of an FQL expression into an internal list structure. There are several reasons for adding some further translation checks at compile time. In the first place, some database

---

3. A recursive function definition does contain a circular chain, but this is broken by an indirect reference through the symbol table.

management systems do not allow character strings to be used as DBIDs. The character strings known to the user must be translated into internal references before a query may be run. The translation tables may not be kept in the same physical file as the database, and may not be structured in a way that allows a character string DBID to be repeatedly translated into its internal form with any degree of efficiency. It is therefore advisable to scan all the functions involved in a query at compile time and substitute the "internal" DBIDs for the external database names.

A second and important improvement to efficiency may be achieved by the compile-time detection of "constant-valued" sub-expressions. This formalism treats the form {U f, a} uniformly as a function f applied to an argument a. When evalstep finds f to be atomic (BOOL, STRING,NUM), it is treated as a "function that ignores its argument a", and f is directly returned. Ideally we would like the same thing to happen even when f is a complicated expression that is constant-valued; we would like a to be ignored and the value of f to be returned directly. However, the final reduction rule in evalstep, i.e.

evalstep({U f, a}) =

:

:

{U e_1, {T a, e_2,..,e_n}} if f = {F e_1,e_2,..,e_n}

always splits f into a new structure surrounding a even if f is constant-valued and ultimately ignores a. When evaluation reaches some other part of the query that points to this same f, this

process will be repeated.

Consider the following part of a query that finds all employees with salary less than the average salary:

!EMPLOYEE.|([SAL,AVESAL].LT)

where AVESAL is an expensive FQL function that returns a constant value over the duration of query evaluation. As part of the overall evaluation, the form {U def(AVESAL), x} is to be evaluated over each employee x generated by !EMPLOYEE. Because of the restructuring of this form by the reduction rule in evalstep, the form is re-evaluated, at great expense, each time it is encountered.

If, however, we knew that the expression f in {U f, a}, though complicated, was constant-valued, we could avoid the restructuring, evaluate f by itself, and replace f by its value. All forms {U f, b}, {U f, c}, etc. in other parts of the query that refer to the same f thus simultaneously benefit from this evaluation, and, in fact, f will be evaluated only once.

It is possible to detect and "tag" constant-valued sub-expressions at compile-time with very little overhead, by modifying the translation rules. For example,

boolean, string, numeric constants F are translated into

{K f}

{F compose, e_1, {K e_2}} is translated into

{K e_2}

{F compose, {K e_1}, e_2} is translated into

{K {U e_2, {K e_1}}}

where {K e} indicates that the sub-expression e is constant-valued. In this way, the "K" tags "bubble" up to the top of maximal constant sub-expressions; in fact, a complete query is always a constant expression. eval and evalstep can now be modified slightly to take advantage of these "K" tags. In particular, eval({K e }) just replaces e by eval(e), and evalstep({U {K e}, a}) returns {K e} irrespective of the form of e. Thus constant-valued subexpressions are never evaluated repeatedly, and we obtain the desired improvement in efficiency.

Detecting constant sub-expressions is a special case of type checking. In the first implementation of FQL, the user was required to specify the range and domain types for each of his defined functions. The compiler would then check that each definition was well-typed before executing the query. There are certain advantages to allowing some type checking to be delayed until run-time. For example, it is possible, in some database management systems, to traverse a collection of mixed record types. In the current development of FQL, we hope to achieve the benefits of both compile- and run-time type checks. This is

briefly discussed in the final section.

5.0 EXTENSIONS

There are several directions in which the techniques we have described could be extended. The developments of immediate practical importance for database interfaces include (1) A more "user oriented" query language: it should be possible to construct a translator for a language that syntactically resembles SEQUEL into the control structures we have described. (2) Interfaces to other database management systems. For the most part, DBMSs at some level provide the kinds of sequence traversing control that is used in Codasyl; but there are important exceptions. An ADABAS query may return a bit vector that describes the locations of records in a sequential file that have a given property. To include such structures in FQL presents no fundamental problem, since all the FQL operators have a meaning when used with vectors. The only question is whether to make vectors visible to the user as a separate data type, or to make the choice of physical representation an internal decision for the interpreter. (3) Updates. While FQL was designed to overcome problems with database query languages, it would be an advantage if it could also handle updates within the same general framework.

Of these three developments, we see only the update problem as requiring new control structures. In this section we shall briefly discuss the update problem and some further optimization techniques.

5.1 Updates

From a purely applicative viewpoint, the purpose of an update is to create a new database without destroying the original. While some systems have been proposed, especially in the area of office automation, that never "forget" data [15], this is an unrealistic proposition for most practical database applications. A simple-minded approach to updating in the functional model is to regard an update as a redefinition of some extensionally defined database function in terms of some (intentional) function definition. Thus to update the SAL(ary) function, one must provide a new definition of it: for example, SAL := [SAL,1.05].x would give everyone a 5% raise. To add new members to a class such as EMPLOYEE, one must provide the values of all database functions on those employees (a stream of tuples), and invoke a suitable updating primitive for !EMPLOYEE.

Such updates are similar in form to defining new functions, and should only be allowed as top level commands in any system. Thus, an update may not occur as a sub-expression of some FQL expression. Moreover, there should be no need for this. The update primitives suggested are all bulk updates. The intentional component of an update may be an arbitrarily complex expression, and may involve external files or databases.

While such update primitives are adequate for a pre-defined database, such as an existing Codasyl system, they will be inadequate for the user who wishes to extend the schema of the database dynamically. We see it as being highly desirable for the

user to add incrementally new types and new database functions. This appears to be essential for "personal" databases and for the "federated" DBMS proposals discussed by McLeod [18]. However, while we believe that a very simple functional model, such as the one we have described, is adequate for defining database queries (especially when these involve heterogeneous systems), a semantically richer model is desirable for update specification. We are currently developing [26] a simple DBMS that may be used in conjunction with an FQL processor, and which will support some recently proposed [12,23] semantic extensions.

5.2 Interfaces For Other Programming Languages

A common problem arises from the need to embed a database access mechanism in some programming language. Although this should become less important as query languages become more powerful, in a practical operating environment, query languages fail to provide all the tools that are provided by a general-purpose programming language. There are in fact two problems, one is how to embed the query language in the programming language itself; the other is how to return data to the calling program.

The simple solution to the first problem is to embed database queries as character strings, and to call some subroutine that interprets the character string as a query. A less trivial approach is to merge the two languages either by means of a preprocessor, or by modifying the compiler for the programming

language. Although an applicative language such as FQL and varieties of the relational calculus can be expressed as a set of functions in a "von Neumann" language such as Pascal, PL/1 or ADA, a programmer in such a system might be somewhat confused at the two styles of programming. Embedding FQL in a language such as LISP, which is based upon an applicative formalism, is much more natural. It may have been noted that our symbolic notation for the internal representation of an FQL expression (section 3.1) corresponds directly to the syntax of a LISP S-expression, and that this is an alternative surface syntax for FQL.

The second problem, that of returning data to the calling program, is cleanly solved by these implementation structures. The class of structures that can be returned to the calling program is precisely the class of printable structures. That is, the atomic types such as STRING, NUM and BOOL; and, recursively, any combination of stream or tuple of these. In order to traverse such a structure, the calling program requires only the functions head, tail, select and the predicate null to check whether a stream is empty. The result of applying these predicates is to produce either an atom or something of type tuple or stream. Through these functions, the calling program may perform multiple traversals of the output structure.

5.3 Query Optimization

FQL, although it is concise, is still a low-level language in the sense that the evaluation of a function or query is directly determined by its definition. In a higher level programming system a query may be transformed into a more efficient representation. For example, the query that prints the names of those employees who earn less than the average salary for their department is simply expressed in FQL as:

```
!EMPLOYEE.|([SAL,DEPT.AVESAL].LT).*NAME;
```

where AVESAL is of type DEPARTMENT->NUM and has been defined to compute the average salary of the employees in a given department. Unfortunately, none of the techniques so far described will prevent AVESAL from being recomputed for each employee. There are two solutions to this problem. One is to store values of AVESAL as they are computed, thereby turning AVESAL into an extensionally defined function (this is sometimes called "memoizing"). The other is to observe that the query can be made much more efficient if it is based upon a traversal of the DEPARTMENT class, rather than the EMPLOYEE class. In order to do this, the query must be subjected to a set of syntactic transformations. For example,

```
!EMPLOYEE --> !DEPARTMENT.*↑DEPT./CONC
```

where /CONC concatenates a stream of streams, is an allowable transformation. Combined with other transformations, it can be used to generate the efficient version of the query for, say, a Codasyl database. We have not yet attempted to augment our existing implementations with such optimizing techniques, nor are we sure that the optimization should be done by the FQL processor

itself as opposed to some higher-level system that generates FQL-like queries. The point to be made here is that a functional expression of a database query reduces the problem of optimization to one of syntactic manipulation of expressions, much as has been suggested for the relational calculus [25].

5.4 The Use Of Data Types

In our initial implementation of FQL, each query and function has a type, and these were provided by the user for each user-defined function. A "compiler", as well as generating the internal form of an FQL expression, also checks that a query and all functions required by that query, are well typed. It also converts database identifiers (the DBIDs mentioned in Section 4) into their internal form. This is too restrictive, especially since a number of apparently well-formed expressions cannot be typed in advance. For example, the function that takes the second element of a stream, TL.HD, cannot be completely typed because we do not know the underlying type of the stream. Thus a separate function must be defined for each type of stream, *NUM, *STRING *EMPLOYEE, etc. This problem has been remedied by the introduction of "wild-card" types: the type of TL.HD is given as *?A -> ?A, where ?A is a wild-card type, but there remain other problems.

We are now experimenting with a system in which much of the type checking is delayed until run-time. There are a number of reasons for doing this. First there are databases in which

streams may contain mixed types (this is actually allowed by the Codasyl standard). Second, in the semantically richer DBMSs mentioned earlier, it is useful to have predicates which determine a type, and functions which coerce one type into another. Third, since FQL is already in use as an interface for a number of Natural Language systems, it may be appropriate to embed it in LISP or some other "untyped" language that is suitable for such applications. Fourth, it may be useful to have at an internal level, alternative representations of types: for example a stream could be represented as we have suggested in this paper; it could also be represented as an array. In an untyped system, it would be possible to write the bulk of the routines for manipulating and coercing such types in FQL itself.

In the long term, however, we believe that the correct solution is to have a richer calculus for data types, and to employ a mixture of compile-time and run-time checking. Using techniques similar to those proposed in [19], it is possible to assign a type automatically to many expressions. Moreover, to have type information available at run-time may simplify parts of the interpreter. For example, in the implementation we have described, each built-in function is responsible for deciding whether to evaluate its arguments. However, this job could be given to eval (as it is in conventional languages), if it could be determined, at run-time, that say plus requires a tuple of numbers, while and requires a tuple of objects that are either booleans or suspensions that will generate booleans. In general, there are known [20] problems involving efficiency and debugging

for lazy evaluators. While these have not caused a problem in the relatively simple functions and queries that are usually built up in a database environment, it may be that a more sophisticated treatment of data types will lead to a more efficient mixture of prompt and lazy evaluation in general programming environments.

6.0 ACKNOWLEDGMENTS

The authors are indebted to Tom Myers for his help in the formalization of FQL and to Ira Winston for building an extremely powerful compiler and user interface. They would also like to acknowledge their debt to the designers of the POP [5] programming language, which not only served as the basis for many of the ideas in this paper but has also provided us with an excellent medium in which to experiment; also to the implementors of the SEED [11] for providing us with a clean and flexible Codasyl system.

7.0 REFERENCES

1. ADA Preliminary Reference Manual, SIGPLAN Notices, Vol 14, 6, June 1979.

2. Backus, J. "Can Programming be Liberated from the von Neumann Style?" Comm. ACM, Vol. 21,1 Jan 1978.

3. Buneman, O.P and R.E. Frankel, "FQL -- A Functional Query Language", Proc. ACM SIGMOD, May 1979.

4. Burge, W.H., Recursive Programming Techniques, Addison Wesley, 1975.

5. Burstall, R.M., J.S. Collins and R.J. Popplestone, Programming in POP-2, Edinburgh, 1971.

6. Chamberlin, D.D. and R.F. Boyce, "SEQUEL: a Structured English Query Language." Proc. ACM SIGMOD, 1974.

7. Date, C.J., An Introduction to Database Systems, Addison Wesley, 1975.

8. Data Base Task Group April 1971 Report, ACM New York, 1971.

9. Frankel, R. MS Thesis, Moore School, University of Pennsylvania, May 1979.

10. Friedman, D.P. and D.S. Wise, "CONS should not evaluate its arguments", in Automata, Languages, and Programming, Edinburgh, 1976.

11. Gerritsen, R. Seed Reference Manual, International Database Systems, Philadelphia, 1978.

12. Hammer, M. and D.J. McLeod, "The Semantic Data Model: a Modelling Mechanism for Database Applications", Proc. ACM SIGMOD, 1978.

13. Harvest User Manual, International Database Systems, Philadelphia, 1979.

14. Iverson, K.E., "Operators", ACM TOPLAS Vol 1,2 1979

15. Kimball, K. "The DATA System", Decision Sciences Working Paper, University of Pennsylvania, March 1978.

16. Landin, P.J., "A Correspondence between ALGOL 60 and Church's Lambda Notation", Comm. ACM, Vol 8, 89-101 and 158-165, 1965.

17. McCarthy, J. et al., LISP 1.5 Programmer's Manual MIT Press, Cambridge, Mass., 1962.

18. McLeod, D. and D. Heimbigner, " A Federated Architecture for Database Systems", Proc. NCC (AFIPS) 1980, to appear.

19. Milner, R. "A Theory of Type Polymorphism in Programming", J. Computer and System Sciences, Vol 17, 348-375, 1978.

20. Morris, J.H., E. Schmidt and P. Wadler, "Experience with an Applicative String Processing Language." ACM POPL, 1980.

21. Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX", Supplement to Proc. ACM SIGMOD, May 1979.

22. Smith, J.M. and P.Y. Chang, "Optimizing the Performance of a Relational Algebra Database Interface". Comm. ACM, vol 18, 10, 1975.

23. Smith, J.M. and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization". ACM TODS, Vol 2,2, 1977.

24. Turner, D.A. " A New Implementation technique for Applicative Languages", Software-Practice and Experience, Vol 9, 1979.

25. Ullman, J.D. Principles of Database Systems, Computer Science Press, 1980.

26. Winston, I. "An Extensible Data Management System based on the Functional Model", Moore School Report, University of Pennsylvania, 1980.

27. Wirth, N., "The Programming Language Pascal", Acta Informatica, Vol 1, 1, 1971.

Appendix A

FQL Syntax

The following BNF gives the syntax for a function definition (<def>), a data-type (<type>), a functional expression (<fexpr>), and a function (<function>) itself. Optional components are denoted by "{"..."} " while "{"..."}* " signifies a set of elements may occur an arbitrary number of times.

```
<def> ::= <name>{:{<type>}->{<type>}=<fexpr>;

<type> ::= NUM
       ::= STRING
       ::= BOOL
       ::= *<type>
       ::= [<type>{,<type>}*]

<fexpr> ::= <function>{.<function>}*

<function> ::= <name>
           ::= *<function>
           ::= |<function>
           ::= &<function>
           ::= [<fexpr>{,<fexpr>}*]
           ::= (<fexpr>)
           ::= !<name>
           ::= ^<name>
```

Appendix B

Standard Functions

The standard functions supported by FQL are grouped here by their type.

Arithmetic functions

The functions +, -, x, /, and MOD all map from [NUM,NUM] into NUM. The functions /+ and /x perform addition- and times-reduction on streams of NUMs; i.e., they map *NUM into NUM. Given an empty stream these functions return their respective identities, 0 and 1.

Relational and Boolean Functions

The operators EQ, NE, GT, LT, GE, and LE map from either [NUM,NUM] or from [STRING,STRING] into BOOL. The functions AND and OR map [BOOL,BOOL] into BOOL; the complement NOT maps BOOL into BOOL. The two reduction operators, /OR and /AND, represent mappings from *BOOL into BOOL and, given empty streams, return the values "true" and "false" respectively.

Constant Functions

Any numeric constant represents a mapping ->NUM whose value is this constant. Any character string similarly denotes the mapping ->STRING. !<name>, where <name> identifies a database class, is a function that generates all members of that class. The function NIL is a constant signifying the empty stream of any type; i.e., ->*α .

Basic Stream-manipulating Functions

Given a non-empty stream, the operation HD returns its first element (*α ->α) while the operation TL returns a stream of the remaining elements (*α->*α). The function CONS takes an element of some type and a (possibly empty) stream whose elements are of that same type and returns a new stream in which the individual element is its "head" while the original stream becomes its "tail"; i.e., CONS : [α,*α]->*α.

Other Stream-manipulating Functions

The function LEN computes the length of a given stream and is thus a mapping from *α into NUM. CONC maps a pair of streams [*α,*α] (whose elements are of the same type) into a single stream *α; /CONC produces a single stream *α by "flattening" an arbitrary stream of streams **α. The operator DISTRIB takes a tuple of the form [*α,β] and returns a stream of tuples *[α,β] with the value of type β "distributed" over the stream of α's.

Miscellaneous Functions

The function #i (i=1,2...n) selects a component from a tuple; i.e., $[\alpha_1, \alpha_2, \ldots, \alpha_n] \rightarrow \alpha_i$. ID is the identity mapping $\alpha \rightarrow \alpha$.

Database Functions

If D is a database identifier, then if D names a function, then D is that function in an FQL expression ↑D is its inverse. If D names a class then !D returns the stream of members of that class (see constant functions).