The Implementation of FDL, a Functional Database Language

A. POULOVASSILIS
King's College London, Strand, London WC2R 2LS

We describe the implementation of the functional database language FDL. FDL improves upon previous languages with a functional data model by allowing any computable function to be defined and stored and by supporting arbitrarily nested data types which are all persistent. All functions are updated incrementally by the insertion and deletion of equations, and an integrity sub-system verifies updates against the declared semantic integrity constraints. We show how a binary relational storage structure is used to support all of FDL's persistent data. We also show how the technique of graph reduction from functional programming languages is modified for the evaluation of FDL queries. Finally, we compare our implementation with that of related languages.

Received November 1990, revised January 1991

1. INTRODUCTION

The functional data model represents the universe of discourse by means of entities and functions between them. Entities can be scalar, for example strings, integers and booleans, or abstract, for example students, teachers and enrolments. Sibley and Kershberg[29] demonstrated that the functional data model can provide a unifying formalism for the network and relational data models, and Hammer and McLeod[13] showed that intentionally defined functions can be used as derivation rules. Shipman integrated these ideas in the database language DAPLEX.[28] In parallel, Buneman *et al.* developed the functional query language FQL.[5] More recently, it has been shown that the functional data model is well suited to modelling complex objects.[2,8] It has also been shown that functions can provide an underlying formalism for object-oriented data models.[4,14]

Despite these advantages of the functional data model, previous implications of it have suffered from a number of drawbacks, as follows.

The functional query languages FQL[5] and GENESIS[3] rely on a relational back-end DBMS for data storage and update facilities, and only functions which are extensionally defined can be stored in this database.

In DAPLEX and languages which derive from it,[12,17,30] functions are defined either extensionally by a set of equations (base functions) or intensionally by a single equation (derived functions). It is not possible for a function to be partly extensionally defined but also to contain a default equation for the general case.

DAPLEX is not computationally complete. Thus, in the DAPLEX-related languages the functional data model is either embedded in a procedural programming language,[30] or extended with procedural features,[17,18] or embedded in Prolog.[12,23] Later functional systems[2,8] assume that intensional functions are coded in an external programming language and not stored in the central database. All these solutions give rise to different formalisms for modelling real-world data and for computation.

Finally, the persistent data types of the DAPLEX-related languages are scalar types, abstract types, and unions, products and sets thereof. No nesting of types is possible and function composition results in 'flattened' sets.

The development of FDL was motivated by the considerable potential of a functional data model on the one hand and the above limitations of previous implementations of it on the other. FDL improves on previous languages with a functional data model in a number of ways:

- It is based upon the λ calculus,[15] thus allowing any recursive function to be defined.
- Functions are not confined to base and derived ones but can be partly extensionally defined, partly intensionally defined.
- All functions, however defined, are stored in a single repository. Equations are stored in a pre-interpreted form ready for subsequent evaluation by a λ calculus evaluator.
- The data types of FDL include arbitrarily nested lists, sums and products and are all persistent.

In this paper we are concerned with the implementation of FDL and we refer the reader to Refs 25, 26 and 27 for a detailed description of the language itself. We begin the paper with an overview of FDL in Section 2, including a comparison between FDL and conventional functional programming languages. In Section 3 we consider the storage level underlying FDL and the operations it supports. In Section 4 we consider the implementation of FDL over this storage level, describing in particular the storage and update of functions and the evaluation of queries. In Section 5 we give some miscellaneous implementation details. Finally, in Section 6 we compare our implementation with that of related database languages and give our concluding remarks.

2. OVERVIEW OF FDL

2.1 Data types

FDL is polymorphic and statically typed.[7] Its primitive data types are string, integer and bool. Extensible data types corresponding to abstract entity types can also be declared – we term these types nonlexical ones, after Verheijen and Van Bekkum.[32] Also supported are arbitrarily nested lists, sums and products. For example, the following FDL statements declare a type synonym, date, for a product of three integers, a new sum type, marital_status, and a new non-lexical type, person:

```
date           == (integer ** integer ** integer)
marital_status :  sum
person         :  nonlex
```

The components of a sum type are tagged by upper-case identifiers, termed constructor functions,[24] which can be thought of as functions without reduction rules. For example, the following statements declare three constructor functions, SINGLE, MARRIED_TO and OTHER, which return objects of type marital_status when applied to arguments of the declared type:

```
SINGLE      :→ marital_status
MARRIED_TO  : person → marital_status
OTHER       : string → marital_status
```

In other words, marital_status is a sum of three types: the 'empty' type (identified by the tag SINGLE), the person type (identified by the tag MARRIED_TO) and the string type (identified by the tag OTHER). Unlike conventional functional languages, FDL supports the extension of sum types with new components (as illustrated above) and also the removal of components via the deletion of constructor functions.

Non-lexical types are also populated dynamically, in this case via creation and deletion commands. Non-lexicals (i.e. elements of non-lexical types) are hidden from the user by displaying in lieu of any non-lexical a transient global variable of the form $x, where x is an alphanumeric identifier. For any non-lexical type, t, a zero-argument generator function All_t returns the current extent of t in the form of a list. Generator functions are non-deterministic with respect to the order in which non-lexicals are returned. For example, given the declaration of the non-lexical type person above and the commands

```
create person $p1, $p2, $p3
delete $p2
```

the query All_person returns either the list [$p1, $p3] or the list [$p3, $p1].

2.2 Functions

Functions are defined incrementally by the insertion and deletion of equations. The type of a function must be declared by the user before the function can appear in any equation. We list some FDL functions below: min_salary is a zero-argument function storing a minimum salary, salary records the salaries of persons, max determines the maximum of a list of integers, and the higher-order function map successively applies a function of type $t_1 \rightarrow t_2$ to a list of elements of type t_1 and returns a list of elements of type t_2, for any types t_1, t_2. The identifiers alpha, alpha1, alpha2 ... are type variables.[7] Lists of the form [ ], [x|y] and $[x_1, x_2, \ldots, x_n]$ are equivalent to the expressions NIL, CONS x y, and CONS x_1 (CONS x_2 (...(CONS x_n NIL)...)), respectively, where CONS and NIL are the constructor functions of the built-in polymorphic list type.

```
min_salary: → integer
min_salary  ⇐ 9000

salary: person → integer
salary $p1  ⇐ 20000
salary $p3  ⇐ 30000
salary v    ⇐ min_salary + 1000

max: (list integer) → integer
max [x]     ⇐ x
max [x|y]   ⇐ let z = = max y in if (x > z) x z

map: (alpha1 → alpha2) (list alpha1) → (list alpha2)
map f [ ]   ⇐ [ ]
map f [x|y] ⇐ [(f x) | map f y]
```

When a new equation is specified, if the RHS of the equation contains no variables and could therefore be evaluated before insertion into the database, it is not in fact evaluated. This is in contrast to DAPLEX and related languages, where the single equation which defines a derived function is stored as entered but where any expression on the RHS of an assignment to a base function is first evaluated and the resulting value stored. The advantage of FDL's approach is uniformity of treatment for all equations. However, one can simulate DAPLEX's update semantics in FDL if so desired since a global variable assignment statement is provided, $x = = expression, which evaluates the expression on its RHS and assigns the value obtained to the global variable on its LHS. For example, we can enter the rule that person $p1's salary is equal to person $p2's salary:

```
salary $p1 ⇐ salary $p2
```

or, alternatively, we can update $p1's salary with the current value of $p2's salary:

```
$s = = salary $p2
salary $p1 ⇐ $s
```

Equations with lists on their RHS are FDL's means of storing bulk data. Also, if an RHS consists of a list of distinct constants, $[c_1, c_2, \ldots, c_n]$, the user can request the storage of this RHS as a set:

f LHS ⇐ **set** $[c_1, c_2, \ldots, c_n]$

The c_i are then retrieved in an arbitrary order whenever the equation is invoked during query evaluation. An advantage of sets over lists on the RHS of equations is that they consume less storage, since no ordering information need be maintained (see Section 4.3.2). A further advantage is that sets can be updated: a new component, c_{n+1}, can be added to the above set by the request

f LHS ⇐ **include** c_{n+1}

and a component, c_i, can be removed from the set by the request

f LHS ⇐ **exclude** c_i

DAPLEX and related languages incorporate similar constructs for the update of base functions.

2.3 Semantic integrity constraints

A final category of information which can be expressed in FDL are semantic integrity constraints over functions of type $t_1 \ldots t_n \rightarrow s$, where $n \geqslant 1$, the t_i are non-lexical types and s is any type. These integrity constraints are zero-argument boolean-valued functions which must always evaluate to true. They are declared by a statement of the form

ic f

and defined by an equation of the form

f ⇐ [⟨$x_1, \ldots, x_n$⟩ || $x_1 \leftarrow$ All_t_1 & ... & $x_n \leftarrow$ All_t_n & list of boolean expressions] = [ ]

where the construct [... || ...] is a list abstraction.[24] For example, the following constraint states that there are no people with less than the minimum salary:

```
c1 ⇐ [x || x ← All_person & (salary x) < min_salary]
   = [ ]
```

and the following constraint states that there are no people who earn more than their manager:

```
c2 ⇐ [⟨x1, x2⟩ || x1 ← All_person & x2 ← All_person &
(manager_of x1) = x2 & (salary x1) > (salary x2)]
   = [ ]
```

2.4 Metalevel functions

As well as a functional model for data and for computation, FDL also has a functional model for its meta data. The meta data are stored in the same repository as the data, as we describe in Section 4. A number of metalevel functions are supported for querying this meta data and we list these functions below. Functions which are shown returning the type void print their result by side effect and cannot be composed with further functions.

```
Types: → (list string)
Functions: → (list string)
Confuns: → (list string)
Ics: → (list string)
Tempvars: → void
Typdec: string → (list string)
Fundec: string → (list string)
Confundec: string → (list string)
Fundef: string → void
```

Types, Functions, Confuns and Ics return the names of current types, functions, constructor functions and integrity constraints, respectively. Tempvars prints a list of the current global variables. Typdec, Fundec and Confundec take (the name of) a type, function or constructor function, respectively, and return its declaration. Finally, Fundef takes a function and prints its defining equations.

2.5 Comparison of FDL and functional programming languages

The theoretical foundation of any functional language, including FDL, is the λ calculus. Functional programming languages extend this calculus to a high-level programming paradigm by including a number of primitive types and functions, a set of type-forming operators for the definition of further types, and facilities for the definition of further functions via equations. As we outlined above, FDL includes all of these features. However, there are a number of important differences between FDL and conventional functional programming languages.

Firstly, in functional programming languages functions are assumed to be defined completely at one time and are 'updated' by redefinition. Such an update is affected either by recompiling the entire programme with the new definition replacing the old (as in Miranda)[31] or by compiling a new version of the function while previous references to the function continue to use the old version (as in ML).[20] Neither approach is feasible in the database environment of FDL, where data is amassed incrementally and where small-grain (i.e. equation-based) update facilities are necessary for functions.

Secondly, in functional programming languages functions are assumed to be totally defined over their argument domain. Thus a failure to match a set of arguments against the equations defining a function is treated as an error. This is not suitable for database querying, where functions will frequently not be exhaustively defined over their argument domains and where a lack of information regarding a particular entity should not cause the abortion of a query over a number of entities. Hence, in FDL a null value @ (of polymorphic type) is assumed to be the default definition of every function.

Thirdly, in languages where functions are defined by multiple equations, a pattern-matching algorithm determines which equation defines the function for any given list of arguments. However, FDL functions are defined incrementally by the insertion and deletion of equations, so pattern-matching algorithms which rely upon some ordering of these equations (such as the top-to-bottom algorithm of Miranda) or upon an examination of all the equations to check that the function is defined unambiguously (such as the best-fit algorithm of HOPE+)[10] are not suitable. Instead, we use a left-to-right, best-fit pattern-matching algorithm (see Section 4.4.1) which is independent of the order in which equations are inserted and retrieved, and which guarantees that ambiguity is always avoided.

Unlike FDL, functional programming languages do not provide dynamically extensible data types. Neither do they include integrated meta data, so the powerful facilities available at the object level cannot be used to formulate metalevel queries. Finally, the long-term storage of functions as source or object code and their run-time storage in a main memory data structure are not adequate for large volumes of data. Thus in FDL functions persist on secondary storage and there is run-time management of this data.

Other functional languages also support the persistence of data on secondary storage, for example Galileo,[1] Amber[6] and Napier88.[21] However, these languages do not assume a functional data model and so inherit the drawbacks of functional programming languages regarding function updates and pattern-matching semantics.

3. THE STORAGE LEVEL UNDERLYING FDL

FDL supplies the Level 1 functionality of the 3-level TriStarp system,[16] which uses a binary relational storage structure (BRSS) at its Level 0. We refer the reader to Refs 9 and 11 for a detailed account of binary relational storage structures. Here we confine ourselves to their salient features.

Briefly, a BRSS supports the storage and retrieval of 3-field records, or triples. Each component of a triple is a token comprising a tag and a value (tokens need to be tagged so as to distinguish between different objects with the same alphanumeric representation, for example the function map and the string map). A simple associative form (saf) identifies by partial match a subset of the triples in a BRSS. A saf is itself a triple, $\langle s_1, s_2, s_3 \rangle$, where each s_i is either a token or the wildcard value, *.

For the purposes of FDL, we use tokens with tags Var, String, Int, Bool, Con, Fun, Builtin and Sys, corresponding to variables, strings, integers, booleans, constructor functions, user-defined functions, built-in functions and system-internal tokens (for example, non-lexicals). We also use two tokens, Let and Apply, to represent definition of local variables and application of expressions, respectively. The following operations are supported by the BRSS underlying FDL:

- **insert**(t) inserts the triple t into the BRSS, returning whether t is indeed a new triple.
- **delete**(s) deletes all the triples which match the saf s, returning the number of triples deleted.
- **retrieve**(s) returns a list of the triples which match the saf s, in some arbitrary order.
- **generate-new** returns a token with tag Sys and value some number, i, such that the token Sys i is not a component of any triple in the BRSS.

In the next section we describe how FDL utilises these operations for the storage and retrieval of its persistent data. We defer implementation details regarding the BRSS to Section 5.

4. THE FDL IMPLEMENTATION

Our implementation has the modular architecture illustrated in Fig. 1. We describe the overall functionality

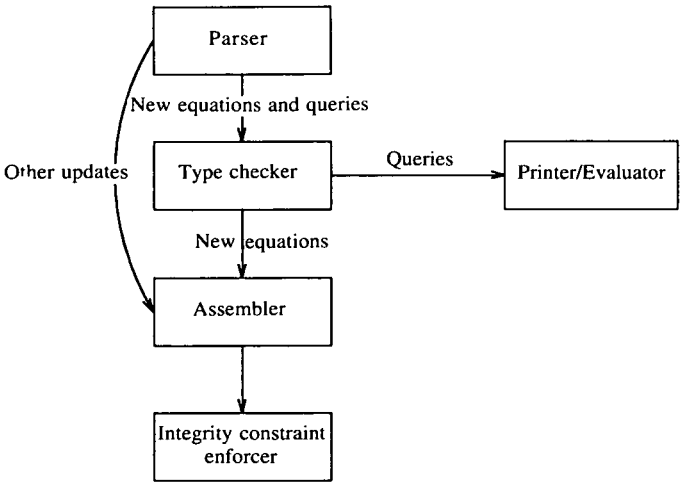

Figure 1. The FDL architecture.

of this architecture in Section 4.1. We then address individual components of the architecture in greater detail, namely the parser, the assembler and the printer/evaluator. FDL's type checker is based upon that described in Refs 7 and 10, with the simple extension that non-lexical types are treated just as built-in types. Due to space limitations, we do not discuss FDL's integrity constraint enforcer here and refer the reader to Ref. 26.

4.1 The Overall Architecture

FDL's parser recognises the following user requests:

- the insertion or deletion of an equation
- the creation or deletion of a non-lexical
- the declaration or deletion of a function
- the declaration or deletion of a type
- the declaration or deletion of a constructor function
- the declaration or deletion of an integrity constraint
- the evaluation of a query

These requests progress through the architecture of Fig. 1 as follows.

Insertion of an equation. The equation is passed to the type checker. This retrieves from the BRSS the type declared by the user for the function being defined and verifies that the equation conforms to this type. If it is correctly typed, the equation is assembled into a set of triples which are inserted into the BRSS. The integrity constraint enforcer then verifies the update with respect to the semantic integrity constraints and if any of these are violated reverses the update.

Deletion of an equation. The type checker is bypassed for such a request, since the type of a function is not altered by the deletion of one of its equations (we recall that in FDL function types are declared explicitly by the user rather than inferred from the defining equations). The assembler identifies a set of triples for deletion from the BRSS and the integrity constraint enforcer verifies the update with respect to the semantic integrity constraints.

Creation of a non-lexical. The extent of any non-lexical type t is stored as a set on the RHS of the single equation defining the generator function All_t:

All_t ⇐ **set** [e_1, ..., e_n]

Hence the creation of a new non-lexical of type t is treated as the following update to All_t, where e is a unique token obtained by a call to **generate-new**:

All_t ⇐ **include** e

Deletion of a non-lexical. The deletion of a non-lexical, e, of type t is treated as the following update to the generator function All_t:

All_t ⇐ **exclude** e

The deletion only proceeds if there are no equations containing e other than the above set valued equation for All_t - we give the implementation of this test in 4.3.3 where we consider deletions in detail.

Declaration of a function. The declaration of a function is treated as an insertion of a metalevel equation for the metalevel function Fundec. Thus, the declarations of the functions min_salary, salary, and max of Section 2 are translated by the parser into the following metalevel equations (we note that types and functions are represented by their names at the meta level).

```
Fundec "min_salary"
  ⇐ ["→", "integer"]
Fundec "salary"
  ⇐ ["person", "→", "integer"]
Fundec "max"
  ⇐ ["(list integer)", "→", "integer"]
```

Such equations are assembled for storage in the BRSS just as equations for user-defined functions.

Deletion of a function. The deletion of a function comprises the deletion of each of its defining equations, followed by the deletion of its declaration. The deletion

of the declaration only proceeds if the function does not appear on the RHS of an equation defining some other function – we consider the implementation of this test in Section 4.3.3.

Declaration of a type. The declaration of a type is treated as the insertion of a metalevel equation defining the metalevel function Typdec. For example, the declarations of the types date, person and marital_status of Section 2 are translated by the parser into the following metalevel equations:

```
Typdec "date"
  ⇐ ["synonym", "(integer**integer**integer)"]
Typdec "person"
  ⇐ ["nonlex"]
Typdec "marital_status"
  ⇐ ["sum"]
```

Deletion of a type. The deletion of a type proceeds only if the type does not appear in the declaration of any function or constructor function and, if the type is a non-lexical one, only if its generator function does not appear in any equation. We consider the implementation of these tests in Section 4.3.3.

Declaration or deletion of a constructor function. The declaration of a constructor function is also treated as a metalevel equation, in this case defining the metalevel function Confundec. For example, the declarations for SINGLE, MARRIED_TO and OTHER in Section 2 translate into the following metalevel equations:

```
Confundec "SINGLE"
  ⇐ ["→", "marital_status"]
Confundec "MARRIED_TO"
  ⇐ ["person", "→", "marital_status"]
Confundec "OTHER"
  ⇐ ["string", "→", "marital_status"]
```

The deletion of a constructor function comprises the deletion of its declaration and only proceeds if there are no equations with reference this constructor.

Declaration or deletion of an integrity constraint. The declaration of an integrity constraint, c, updates the metalevel functions Ics and Fundec:

```
Ics ⇐ include "c"
Fundec "c" ⇐ ["→", "bool"]
```

The insertion of the single equation defining an integrity constraint is treated as any other insertion. The deletion of an integrity constraint is identical to the deletion of any other function.

Evaluation of a query. Queries are expressions to be evaluated with respect to the current functions and non-lexical type extents. Queries are type-checked and, if type-correct, passed to the printer/evaluator for evaluation and output of results.

4.2 The Parser

The parser translates all update requests into equations, as described in Section 4.1. It then translates both equations and queries into directed acyclic graphs (dags) where

```
dag = token | Apply dag dag | λ dag dag |
      Let dag dag dag
```

Dags of the form Apply d_1 d_2, λ d_1 d_2 and Let d_1 d_2 d_3 represent application, λ abstraction and definition of local variables, respectively. As their name suggests, dags are implemented as directed acyclic graphs in main memory. The inner vertices of these graphs are labelled Apply, λ or Let and the leaf vertices are simple tokens. Vertices labelled Apply and λ have two branches emanating from them and vertices labelled Let have three.

A query or equation undergoes three transformations during its translation into a dag: zero-argument functions are given a dummy argument (Sys 0), global variables are replaced by their current values, and local variables are renamed as Var 0, Var 1, Var 2, ... according to their order of appearance. For example, the query

let s = = salary $p1 **in** s − min_salary

is represented by the following dag, where p1 stands for the current value of the global variable $p1:

```
Let  (Var 0)  (Apply (Fun "salary") (Sys p1))
     (Apply (Apply (Builtin "−") (Var 0)) (Apply
       (Fun "min_salary") (Sys 0)))
```

Similarly, the two equations for the function map in Section 2 are represented by the dags:

```
λ (Var 0) (λ (Con "NIL") (Con "NIL"))
and
λ (Var 0) (λ (Apply(Apply(Con "CONS") (Var 1))
  (Var 2))
    (Apply(Apply(Con "CONS")  (Apply(Var 0)
    (Var 1))) (Apply(Apply(Fun "map") (Var 0))
    (Var 2))))
```

The translation of list abstractions is beyond the scope of this paper and we refer the reader to Ref. 24 for a detailed exposition.

4.3 The assembler

In this section we describe the storage of equations (4.3.1) and the optimised storage of equation right-hand sides (4.3.2). We also show how FDL's deletion commands are implemented given this storage scheme (4.3.3).

4.3.1 Storage of equations

The equations defining each function (whether user-defined or metalevel) are stored in the form a labelled tree, which we term the function's match tree. The root of this tree is the function, and the remaining nodes are unique tokens of the form Sys i, which are generated by calls to **generate-new**. The arcs of the tree are labelled with the components of the LHSs of equations. Each arc $n \rightarrow^{label} m$ is stored as one triple $\langle n, label, m \rangle$. We illustrate the match trees for the functions salary, map and min_salary of 2.2 in Fig. 2. We observe that the arcs emanating from each node of a match tree are uniquely labelled so that the match tree represents a left factoring of the equations defining the function.

The insertion of a new equation thus comprises two parts:

(i) the assembly of the equation LHS into a set of triples to be inserted into the match tree; and

(ii) the assembly of the equation RHS into a set of triples uniquely identifiable by the leaf node of the LHS.

Step (i) is accomplished by 'flattening' the equation

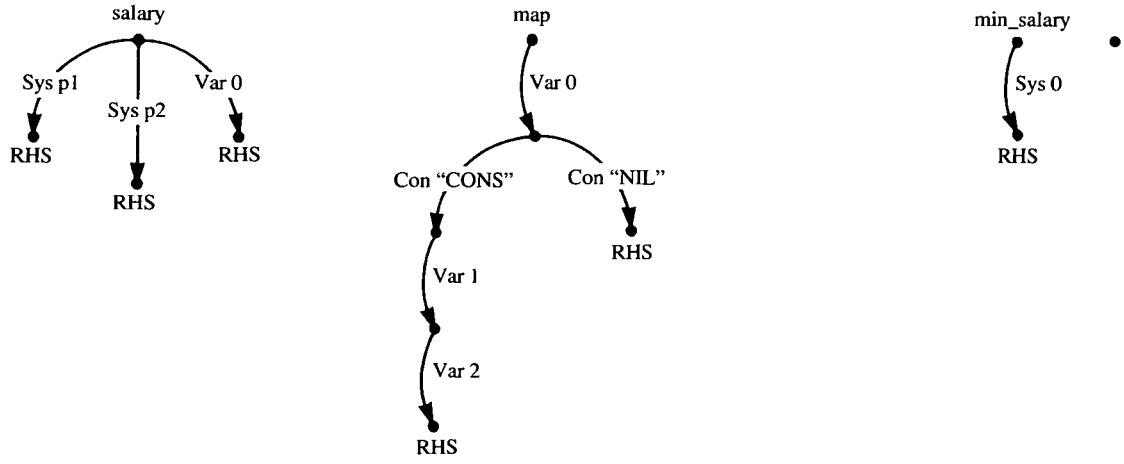

Figure 2. Examples of match trees.

LHS to remove λ abstractions and applications and then traversing the match tree from the root to the leaves, inserting any arcs which are missing (if there are no arcs missing, there already exists an equation with the same LHS and the new RHS simply replaces the old). Step (ii) is accomplished by storing the sub-expression representing the RHS as a set of triples which contain the leaf node of the LHS in their first field and a unique number in their second field. For example, consider the insertion of the equation

map f [x|y] ⇐ [(f x) | map f y]

into an empty match tree. The 'flattened' LHS is [Var 0, Con "CONS", Var 1, Var 2], hence the following four triples are inserted, for some unique tokens Sys i_1 – Sys i_4:

⟨Fun "map", Var 0, Sys i_1⟩ ⟨Sys i_1, Con "CONS", Sys i_2⟩ ⟨Sys i_2, Var 1, Sys i_3⟩ ⟨Sys i_3, Var 2, Sys i_4⟩

The dag representing the RHS of the above equation is

(Apply(Apply(Con "CONS") (Var 1))) (Apply(Apply(Fun "map") (Var 0)) (Var 2)))

which is assembled by a pre-order traversal into the following set of triples:

⟨Sys i_4, Int 0, Apply⟩⟨Sys i_4, Int 1, Apply⟩
⟨Sys i_4, Int 2, Con "CONS"⟩⟨Sys i_4, Int 3, Apply⟩
⟨Sys i_4, Int 4, Var 0⟩⟨Sys i_4, Int 5, Var 1⟩
⟨Sys i_4, Int 6, Apply⟩⟨Sys i_4, Int 7, Apply⟩
⟨Sys i_4, Int 8, Fun "map"⟩⟨Sys i_4, Int 9, Var 0⟩
⟨Sys i_4, Int 10, Var 2⟩

Thus the token Sys i_4 links the RHS of the equation to the LHS and also enables the retrieval of the RHS by only one call to the BRSS, namely **retrieve**(Sys i_4, *, *).

4.3.2 *Optimised storage of right-hand sides*

Two simple optimisations are applied to the storage of equations. Firstly, if the RHS of the equation is a constant, it is stored as the third component of the last triple representing the LHS. For example, the equation salary $p1 ⇐ 20000 of Section 2 is stored as one triple ⟨Fun "salary", Sys p1, Int 20000⟩.

A second optimisation concerns the storage of lists. We recall from Section 2 that if an equation has an enumerated list on its RHS, [c_1, c_2, ..., c_n], the user can request the storage of this RHS as a set. This results in the storage of n arcs ⟨Sys i, last-component, c_1⟩, ⟨Sys i, last-component, c_2⟩, ... ⟨Sys i, last-component, c_n⟩, for some Sys i, at the bottom of f's match tree as opposed to just one arc. The number of triples stored is thus reduced by a factor of four since no Apply or CONS tokens are stored. The inclusion of a further constant c_{n+1} in the above set is achieved by inserting the triple ⟨Sys i, last-component, c_{n+1}⟩ and the exclusion of a constant c_i by deleting the triple ⟨Sys i, last-component, c_i⟩.

4.3.3 *Deletions*

Given the above storage scheme for both user-defined and metalevel equations, we can use the retrieval facilities provided by the BRSS to implement FDL's deletion commands without having to store any further 'dependency' information.

Deletion of an equation. The appropriate path from the root of the match tree to a leaf is identified, the RHS is deleted, and any arcs of the match tree which are now redundant are also deleted.

Deletion of a function. We recall from Section 4.1 that this consists of deleting both the function's definition and its declaration. We delete the definition by traversing and deleting the entire match tree. We only delete the declaration if the function does not appear in the RHS of any other function definition. Given our storage of RHSs, this test simply consists of verifying that **retrieve** ⟨*, *, Fun function-name⟩ = [ ].

Deletion of a constructor function, c. We only delete c if it does not appear in the LHS or RHS of any equation. We observe from Sections 4.3.1 and 4.3.2 above that c appears in a triple of the form ⟨token, Con c, token⟩ if it is on the LHS of an equation and in a triple of the form ⟨token, token, Con c⟩ if it is on the RHS. Hence, it is sufficient to check that **retrieve** ⟨*, Con c, *⟩ = **retrieve** ⟨*, *, Con c⟩ = [ ].

Deletion of a type t. This requires two checks. Firstly, t must not appear in any declaration, that is **retrieve** ⟨*,

*, String t⟩ = [ ]. Secondly, if t is a non-lexical type the generator function All_t must not appear on the RHS of any equation, that is **retrieve** ⟨*, *, Fun All_t⟩ = [ ].

Deletion of a non-lexical e of type t. We only delete e if it does not appear in the LHS or RHS of any equation. We recall that the extent of t is stored on the set-valued RHS of the generator function All_t, in other words as a set of triples ⟨All_t, Sys 0, e_1⟩, ⟨All_t, Sys 0, e_2⟩, ..., ⟨All_t, Sys 0, e_n⟩. Hence, we need to check that **retrieve** ⟨*, e, *⟩ = [ ] and **retrieve** ⟨*, *, e⟩ = [⟨All_t, Sys 0, e⟩].

4.4 The printer/evaluator

Query evaluation in FDL is based upon the technique of graph reduction[24] employed in functional programming language implementations, with the important difference that when evaluating a function application, f $a_1 \ldots a_n$, the function f is not replaced by a λ abstraction representing the 'whole' of f but by the RHS of the single equation which defines f for the arguments $a_1 \ldots a_n$.

Query evaluation requires two main operations, PRINT and EVAL, listed in Fig. 3. The evaluation is driven by PRINT. PRINT passes the dag representing a query to EVAL which simplifies the dag into weak head-normal form (WHNF),[24] that is, to a dag of the form Apply(...(Apply d_0 d_1)...) d_n where d_0 is a constant, constructor function, or function of arity $m > n$. Upon return from EVAL, PRINT displays d_0 and recursively calls itself to evaluate and display d_1 to d_n in turn.

In EVAL, the operation MATCH (which we consider further in Section 4.4.1) matches the arguments of a user-defined function against the arcs of its match tree, creating bindings for any variable arcs on the way and terminating at a unique RHS. MATCH returns this RHS with its free variables instantiated to the appropriate bindings. The operation REDUCE-BUILTIN executes the code defining a built-in function for the given arguments and replaces the function application by the result. REDUCE-BUILTIN also undertakes the evaluation of inverse functions, which we consider further in Section 4.4.2.

4.4.1 *The pattern matcher*

The pattern matcher, MATCH, takes a user-defined function and its arguments and returns the definition of the function appropriate for these arguments. Conceptually, FDL's pattern-matching algorithm scans the LHSs of the function's equations from left to right and at each parameter selects only those equations which contain the most specific match for the current argument. The detailed semantics of this algorithm are given in Ref. 26. The algorithm has two desirable properties:

- the semantics of a function, f, are independent of the order in which its equations are inserted or retrieved;
- no update can give rise to an ambiguous definition for f (since no two equations can have the same LHS and so the process of selecting only the most specific matching equations at each parameter must terminate with at most one equation).

Given our storage of equations in a match tree, MATCH implements the above semantics by traversing the match tree from the root to a leaf, selecting the most specific arc at each node and not backtracking. If at any step of the traversal there is no arc matching the current argument, MATCH returns Con "@" (we recall that @ is FDL's 'undefined' null value). We list MATCH in Fig. 4. The parameter "v" records the number of the next variable expected in the traversal and is initially 0.

From the clause marked † in Fig. 4 we observe that FDL evaluates function applications lazily, i.e. it only evaluates them if they need to be matched against a constant. We also observe that this clause is the only place where MATCH retrieves arcs matching a saf of the form ⟨node, *, *⟩ as opposed to the more specific saf ⟨node, label, *⟩. Since the arcs emanating from any node in a match tree are labelled uniquely, there will be at most one arc matching a saf of the form ⟨node, label, *⟩ but there may be any number of arcs matching a saf of the form ⟨node, *, *⟩. To avoid the complete retrieval of these latter arcs when in fact we are only interested to see whether there is any constant-labelled arc, a 'triple-at-a-time' retrieval operation would be useful. In fact, the BRSS underlying FDL does provide such an

```
PRINT(q)
case EVAL(q) of
    a function application:
        display ('partially applied function')
    an application, Apply(...(Apply(Con c) d1)...) dn, where n ≥ 0:
        display(Con c)
        for i := 1 to n PRINT(di)

EVAL(q)
case q of
    a local definition, Let d1 d2 d3:
        substitute d1 by d2 in d3
        return(EVAL(d3))
    a user-defined function application, Apply(...(Apply (Fun f) d1)...) dn, where n ≥ Arity(f):
        d0 := MATCH([d1, ..., dArity(f)], [ ], Fun f)
        return(EVAL(Apply(...(Apply d0 dArity(f)+1)...) dn))
    a built-in function application, Apply(...(Apply(Builtin f) d1)...) dn, where n ≥ Arity(f):
        return(REDUCE-BUILTIN(q))
    other:
        return(q)
```

Figure 3. The printer/evaluator.

operation and we refer the reader to Ref. 9 for details of it.

The operation DISASSEMBLE in Fig. 4 reconstructs into a dag the set of triples representing an RHS by reversing the pre-order traversal of Section 4.3.1. INSTANTIATE instantiates the variables in this dag according to the bindings collected by MATCH. Once an RHS has been disassembled the resulting dag is in fact maintained in a main-memory cache in order to avoid its repeated retrieval and disassembly (for example during the evaluation of a recursive function). A main-memory cache for the LHS of equations, that is for the arcs of match trees, also achieves a significant improvement in query evaluation times.

4.4.2 Inverse functions

For any first-order function f: → s such that t is a non-lexical type and s is not a list type, FDL provides an inverse function inv_f: s → (list t) defined by the equation

$$\text{inv\_f}\ x \Leftarrow [y \parallel y \leftarrow \text{All\_t}\ \&\ (f\ y) = x]$$

If f is defined extensionally by a set of equations

$$f\ e_1 \Leftarrow c_1$$
$$f\ e_2 \Leftarrow c_2 \ldots$$
$$f\ e_n \Leftarrow c_n$$

we optimise the evaluation of expressions of the form (inv_f E) by evaluating the argument E to a constant, c say, and by retrieving (inv_f c) directly from the BRSS rather than using the equation for inv_f: we call **retrieve** ⟨Fun f, *, c⟩ and return a list containing the second field of each triple retrieved.

Similarly, for any first-order function f: t → (list s), where t is a non-lexical type and s is any type, FDL provides a 'converse' function inv_f: s → (list t) defined by the equation

$$\text{inv\_f}\ x \Leftarrow [y \parallel y \leftarrow \text{All\_t}\ \&\ \text{member}\ x\ (f\ y)]$$

Again, if f is defined extensionally by a set of equations

$$f\ e_1 \Leftarrow \textbf{set}\ [c_{11}, \ldots, c_{1k_1}]$$
$$f\ e_2 \Leftarrow \textbf{set}\ [c_{21}, \ldots, c_{2k_2}] \ \ldots$$
$$f\ e_n \Leftarrow \textbf{set}\ [c_{n1}, \ldots, c_{nk_n}]$$

we optimise the evaluation of expressions (inv_f E) by evaluating E to a constant, c say, and returning a list containing the second field of each triple in **retrieve** ⟨Fun f, *, c⟩.

```
MATCH (arg_list, bindings, node, v)
if arg_list = [ ]
    RHS_triples := retrieve ⟨node, *, *⟩
    if RHS_triples = [ ]
        return (node)
    else a := DISASSEMBLE(RHS_triples)
        return (INSTANTIATE (a, bindings))
else [arg | rest] := arg_list
    case arg of
†     an expression not in WHNF:
        if retrieve ⟨node, *, *⟩ contains a triple
          ⟨node, Con c, n⟩ for some c and n
            arg := EVAL(arg)
        /* drop through to... */
      an application, (Apply(...(Apply d0 d1)...) dn):
        if retrieve ⟨node, d0, *⟩ = [⟨node, d0, n⟩]
          for some n
            return(MATCH(rest, bindings, n, v))
        else if retrieve ⟨node, Var v, *⟩ = [⟨node,
          Var v, n⟩] for some n
                return(MATCH(rest, [⟨Var v, arg⟩|
                  bindings], n, v+1))
        else if retrieve ⟨node, d0, *⟩ = [⟨node,
                  d0, n1⟩, ... ⟨node, d0, nm⟩] or
                retrieve ⟨node, Var v, *⟩ = [⟨node,
                  Var v, n1⟩, ... ⟨node, Var v, nm⟩]
                return(Apply(Apply(Con "CONS")
                  n1)    (...    Apply(Apply(Con
                    "CONS")    (nm))    (Con
                    "NIL")...))
        else return(Con "@")
```

Figure 4. The pattern-matcher.

5. FURTHER IMPLEMENTATION DETAILS

The BRSS currently underlying FDL consists of a triple store implemented by Derakhshan[9] and a lexical token converter (LTC) based upon the work described by Lavington.[19] The triple store provides operations for the insertion, deletion, and retrieval of triples of fixed-length identifiers. The LTC provides a one-to-one two-way mapping between tokens and fixed-length identifiers to be used in the triple store in the place of tokens. The LTC also supplies the **generate-new** operation.

In the interests of simple memory management and efficient manipulation of strings, we represent tokens by their corresponding LTC identifiers in FDL's dags and we only convert them back for printing. Thus dag vertices are of fixed rather than variable length. These vertices are allocated from a statically declared heap. Vertices no longer required during the evaluation of a query are added to a free list and recycled as necessary (see Ref. 24 for a detailed discussion of memory management in functional languages).

The triple store is a 3-dimensional grid file[22] extended by Derakhshan with a facility for optimising query response time given the expected probability distribution for queries.[9] A key feature of an *n*-dimensional grid file is that tuples corresponding to neighbouring points in the *n*-dimensional key space are clustered into the same data pages. Furthermore, Derakhshan's optimisation facility has allowed the 3-dimensional grid file used by FDL to be set up so that the clustering of triples is biased in favour of the first key field and within that in favour of the second key field. We justify this tuning by observing that the safs used by FDL during query evaluation always have their first field specified and frequently have their second field specified too (refer to MATCH in Fig. 4 and the evaluation of inverse functions in Section 4.4.2). In fact, it is only during the verification of deletions (in Section 4.3.3) that the first field of our safs is indeterminate.

The main requirements for the implementation of any database language derive from the assumption that the language will be accessing data structures which do not reside in main memory. Thus the minimisation of database I/O during query processing is desirable. In particular:

- virtual storage should be utilised for buffering database I/O;
- main memory should be utilised for caching data structures which are used frequently;
- constructor functions such as CONS should be evaluated lazily;
- data which are logically related should be stored physically clustered.

In the case of FDL, database I/O is handled by the Level 0 BRSS, which does indeed include a buffer manager. Main memory is utilised during query evaluation and integrity constraint enforcement; in particular, the LHS and RHS of most frequently used equations are cached, as are the current semantic integrity constraints. Constructor functions are evaluated lazily, since query evaluation only proceeds to weak head-normal form. Finally, given the clustering properties of the grid file, the LHSs of equations for single-argument functions are clustered via the function name. The triples constituting the RHS of any equation are also clustered by virtue of their common first field. However, the LHSs of equations for multi-argument functions are each stored in more than one triple and so may not be clustered.

6. CONCLUDING REMARKS

We have described the implementation of the computationally complete functional database language FDL. We have discussed how a BRSS is utilised for the storage, update and retrieval of FDL's persistent structures, in particular showing how

- the left- and right-hand sides of equations are decomposed into sets of triples;
- meta data is stored and manipulated in the same way as data;
- FDL's left-to-right best-fit pattern matching is implemented by identifying a single path from the root of a function's match tree to its leaves, with no back-tracking;
- the update of a function is achieved by a similar traversal of its match tree.

We have also shown how the technique of graph reduction is adapted for the evaluation of FDL queries by extending it with inverse functions and with the retrieval of equations from the BRSS. Our implementation compares well with other database languages which adopt a functional data model.

P/FDM[12,23] utilises the UNIX NDBM data management utility for data storage and access. Abstract entities are accessed by specifying the values of a number of key functions thereof (cf. FDL, where entities are uniquely identified by means of surrogates). Equations can only be accessed by specifying the function and arguments, so inverse functions must be stored explicitly (cf. FDL, where inverses of base functions are derived and data is not replicated). Derived functions are translated into Prolog and stored as source code (cf. FDL, where derived functions are stored in a pre-interpreted form and are evaluated functionally).

EFDM[17,18] is implemented in the persistent programming language PS-Algol. The extent of each abstract entity type is stored as a linked list. Each abstract entity is accompanied by a pointer to a further list containing the values of all single-argument base functions of the entity. The inverse of a single-argument base function is therefore derived by traversing the entire extent of its argument type (cf. FDL, where the inverse of such a function does not require a scan of the argument type). Derived functions are stored as abstract syntax trees, analogously to FDL. However, the functional component of EFDM is limited and cannot support the persistence of arbitrary recursive functions.

Some areas of FDL require further work. The pattern-matching algorithm could be modified to incorporate back-tracking up the match tree while also retaining the desirable property that no update causes a function definition to become ambiguous. Also, the graph reduction of λ expressions is in general less efficient than the reduction of semantically equivalent expressions consisting only of combinators[24] since, with the latter, overheads arising from copying sub-trees and instantiating variables are greatly reduced. Clearly, the implementation of FDL could be modified so that all λ expressions are transformed into combinator expressions before being stored or evaluated.

Acknowledgements

The author would like to thank Mir Derakhshan, Peter King and Carol Small of Birkbeck College for their invaluable encouragement during the development of FDL. This work has been supported by a scholarship from the Greek State Scholarships Foundation (I.K.Y.) and subsequently by a Postdoctoral Fellowship from the S.E.R.C. at University College London.

REFERENCES

1. A. Albano, L. Cardelli and R. Orsini, Galileo: a strongly typed, interactive conceptual language. *ACM Transactions on Database Systems* **10** (2) (1985).
2. F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, *FAD*, a powerful and simple database language. *Proc. 13th VLDB Conference, September 1987.*
3. D. S. Batory, T. Y. Leung and T. E. Wise, Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems* **13** (3) (1988).
4. D. Beech, A foundation of evolution from relational to object databases. In *Advances in Database Technology (EDBT 88)*, Lecture Notes in Computer Science no. 303. Springer-Verlag, Heidelberg (1988).
5. P. Buneman and R. E. Frankel, An implementation technique for database query languages. *ACM Transactions on Database Systems* **7** (2) (1982).
6. L. Cardelli, Amber. In *Combinators and Functional Programming Languages*, edited G. Cousineau *et al.* Springer-Verlag, Heidelberg (1985).
7. L. Cardelli, Basic polymorphic typechecking. *Science of Computer Programming* **8** (2) (1987).
8. U. Dayal *et al.*, Simplifying complex objects: The PROBE

approach to modelling and querying them, presented at the International Workshop on the Theory and Applications of Nested Relations and Complex Objects, Darmstadt, April 1987.
9. M. Derakhshan, A Development of the Grid File for the Storage of Binary Relations. *Ph.D. Thesis*, Birkbeck College, University of London (1989).
10. A. J. Field and P. G. Harrison, *Functional Programming*. Addison-Wesley, London (1988).
11. R. A. Frost, Binary relational storage structures. *The Computer Journal* **25** (3) (1982).
12. P. M. D. Gray, D. S. Moffat and N. W. Paton, A Prolog interface to a functional data model database. In *Advances in Database Technology (EDBT 88)*, Lecture Notes in Computer Science no. 303. Springer-Verlag, Heidelberg (1988).
13. M. M. Hammer and D. J. McLeod. The semantic data model: a modelling mechanism for database applications. *Proc. of the ACM SIGMOD Conference, 1978*, pp. 26–35.
14. S. Heiler and S. Zdonik, Views, data abstraction and inheritance in the FUGUE data model. In *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science no. 334, Springer-Verlag, Heidelberg (1988).
15. J. R. Hindley and J. P. Seldin, *Introduction to Combinators and the Lambda Calculus*. Cambridge University Press (1986).
16. P. King, M. Derakhshan, A. Poulovassilis and C. Small, TriStarp – an investigation into the implementation and exploitation of binary relational storage structures. *Proceedings of BNCOD-8*. Pitman Publishing Co., Bristol (1990).
17. K. G. Kulkarni and M. P. Atkinson, EFDM: extended functional data model. *The Computer Journal* **29** (1) (1986).
18. K. G. Kulkarni and M. P. Atkinson, Implementing an extended functional data model using PS-Algol. *Software Practice and Experience* **17** (3), 171–185 (1987).
19. S. H. Lavington and C. Wang, *A Lexical Token Converter for the IFS*. Internal Report IFS/5/84, Department of Computer Science, Manchester University (1984).
20. R. Milner, A proposal for standard ML. *Proc. ACM Symposium on LISP and Functional Programming, 1984*, pp. 184–197.
21. R. Morrison *et al.*, *The Napier88 Reference Manual*. Universities of Glasgow and St Andrews, PPRR-77-89.
22. J. Nievergelt, H. Hinterberger and K. C. Sevcik, The grid file: an adaptable symmetric multikey file structure. *ACM Transactions on Database Systems* **9** (1) (1984).
23. N. W. Paton and P. D. M. Gray, *Object Storage in Databases*. Research Report AUCS/TR8803, Department of Computing Science, Aberdeen University.
24. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey (1987).
25. A. Poulovassilis, *FDL*: an integration of the functional data model and the functional computation model. *Proceedings of BNCOD-6*, Cambridge University Press (1988).
26. A. Poulovassilis, The design and implementation of FDL, a Functional Database Language. *Ph.D.Thesis*, Birkbeck College, University of London (1989).
27. A. Poulovassilis and P. King, Extending the functional data model to computational completeness. In *Advances in Database Technology (EDBT 90)*, Lecture Notes in Computer Science no. 416, Springer-Verlag, Heidelberg (1990).
28. D. W. Shipman, The functional data model and the data language DAPLEX-*ACM Transactions on Database Systems* **6** (1) (1981).
29. E. H. Sibley and L. Kershberg, Data architecture and data model considerations. *Proc. of the AFIPS National Computer Conference* (1977).
30. J. M. Smith, S. Fox and T. Landers, *ADAPLEX Rationale and Reference Manual*. Computer Corporation of America, CCA-83-08.
31. D. A. Turner, Miranda – a non-strict functional language with polymorphic types. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science no. 201, Springer-Verlag, Heidelberg (1985).
32. G. M. A. Verheijen and J. Van Bekkum, NIAM: an information analysis method. In *Information Systems Design Methodologies: A Comparative Review*. North-Holland, Amsterdam (1982).

Book Review

B. SAMWAYS and T. BYRNE-JONES
Collins Gem Basic Facts: Computers, 3rd edn Harper Collins, Glasgow, 1991. £2.99. ISBN 0 00 459250 6 (UK edition); 0 00 459251 4 (export edition).

This is one of a series of 'Basic Facts' booklets on various subjects intended mainly for schools. Many smallish books are referred to as pocket books, although one would need very large pockets to contain them; I even once knew a publication of telephone directory size that called itself a pamphlet. This little dictionary, on the other hand, really is a pocket book. Its size is only $4\frac{1}{2}$ in. by $3\frac{1}{4}$ in. by less than $\frac{1}{2}$ in. thick, and its cover is made of a plastic material that would stand up to a lot of wear. If you do need a computing dictionary to carry around in your pocket, this one certainly has advantages.

Within that small size, it runs to 247 pages and over 700 definitions which, in general, seem to be clear and accurate. Among a few errors and oddities noted: (1) under 'segment display' it is shown how the segments combine to form each digit from 1 to 9 but 0 is omitted; (2) there is the common muddle over whether or not K means kilo and when 1000 is indicated and when 1024. This is carried forward into the definitions of megabyte and gigabyte. Then we find tetrabyte, which should be terabyte surely, defined as '1000 gigabytes, or 1024 megabytes'; (3) 'character' is defined as 'any keyboard symbol'. This is followed by some examples, but there is no suggestion that non-keyboard characters may exist; (4) an appendix entitled 'Abbreviations and acronyms' contains a number of entries that do not really fit that heading e.g. AND, OR, NOT, PASCAL, QWERTY (or can QWERTY be considered as an abbreviation?). Adding 'etc' to the heading would be advisable.

In general, however, I find it an attractive booklet that can be recommended.

I. D. HILL
London