Incomplete Information and the Functional Data Model

D. R. Sutton and P. J. H. King

Department of Computer Science, Birkbeck College, London WC1E 7HX, UK

This paper presents an experimental functional database language Fudal which is a further development of our group's work on persistent functional database languages. In this latest work we consider how unknown or partially known information can be treated in the functional context. The language we have implemented, Fudal, includes certainty and possibility operators. We outline the problems that are caused by the use of null values and truth functional logic in conventional database languages, and show how these problems can be overcome by defining the semantics of queries of a database containing partial information in terms of its 'completions'. If $D$ is a database containing partial information then a completion of $D$ is a database which is consistent with $D$ and contains no partial information. We demonstrate that, even when a database has a large number of completions, sensible queries can be constructed using certainty and possibility operators. Finally we show how these operators can be implemented and discuss the use of Fudal in practical contexts.

Received August 19, 1994, revised January 27, 1995

1. INTRODUCTION

In this paper we adopt the functional data model summarized in the EDBT paper by Poulovassilis and King (1990) which further discusses the computationally complete functional database language FDL first described by King and Poulovassilis (1988). The work we present here is essentially a further development of FDL which extends the treatment of incomplete or partial information. Preliminary work on Fudal has been presented by Sutton and King (1992a,b). This paper presents an improved, though still informal, semantics, investigates certain problems not touched upon in our previous work, and outlines methods for implementing and optimizing Fudal.

Our implementation is over a content-addressed quadruple store based on Nievergelt's grid file (Nievergelt et al., 1984) which provides the persistence (Derakhshan, 1989; King et al., 1990). This software is the same as was used to implement FDL but parameterized to store quadruples rather than triples. A detailed discussion of these implementation aspects is given by Sutton (1993). However, these implementation aspects are not relevant to the questions discussed in this paper and are not necessary to its understanding.

An important aspect of this work is the abandonment of truth functional logic and the introduction of certainty and possibility operators into logical expressions. After discussing our motivation in Section 2, we discuss the deficiencies of truth functional logic in Section 3 and, in Section 4, review the relationship between our work and previous, purely theoretical, endeavours. The language Fudal is then described in Section 5 and examples of its use are given in Section 6. An efficient implementation of Fudal is discussed in Section 7.

2. MOTIVATION

In Fudal, as in our previous work, the values of attributes of abstract entities are obtained by evaluating functions which take these entities as arguments. A very simple illustration is

\[
\text{age} \, \text{Bill} = 37; \quad \text{salary} \, \text{Ann} = 25;
\]

in which we have used Bill and Ann to denote abstract entities corresponding to people. (The notation used in this section differs only slightly from that used in other functional languages, e.g. Miranda. A more detailed description of our notation is given in Section 5.) If we wished to treat incomplete information in the same way as in the previous language FDL (and as in SQL) we would introduce a value null whose semantics is 'value at present unknown', for instance

\[
\text{age} \, \text{Jane} = \text{null};
\]

However, in the real world we often have partial knowledge; that someone's age is between 25 and 30 although not known exactly; that the expected requirement of a material by a building project is between 25 and 30 tons; that the expenditure on an item will exceed £2000 but will not be more than £2500; and so on. It seems natural that such knowledge should be specifiable as the right hand sides of function definitions. For example:

\[
\text{age} \, \text{Jane} = \text{Oneof} \{25,26,27,28,29,30\};
\]

with an obvious interpretation. Furthermore if we knew, say, that Jane was married to Mike who was 5 or 6 years older than her then we should be able to enter:

\[
\text{age} \, \text{Mike} = \text{age} \, \text{Jane} + \text{Oneof} \{5,6\};
\]

In Fudal, as in FDL, we use list comprehensions

\[
\text{age} \, \text{Jane} = \text{null};
\]

Downloaded from http://comjnl.oxfordjournals.org/ by guest on July 20, 2015
(Peyton-Jones, 1987) to pick out entities satisfying particular criteria (more details may be found in Section 5.1). For instance

\[\{x \mid x \leftarrow \text{all\_person} \mid (\text{age} \ x) > 28\}\]

would return a list containing the identifiers of all entities of the type person whose age is greater than 28. However, in the presence of incomplete information a problem arises. Mike should clearly be included in the list returned by this comprehension but what about Jane? This problem can be avoided by introducing certainty and possibility operators into the query. For example we could enter

\[\{x \mid x \leftarrow \text{all\_person} \mid \text{Surely}((\text{age} \ x) > 28)\}\]

if we want the list of all people who are known with certainty to be older than 28, thus excluding Jane, or

\[\{x \mid x \leftarrow \text{all\_person} \mid \text{Maybe}((\text{age} \ x) > 28)\}\]

if we want all people who might be older than 28, which would include Jane.

We might also wish to express conditional information. For example we might have a database concerning projects whose expense allowances are given by:

\[
\text{expense\_allowance} \ x = \text{if} \ (\text{location} \ x) = \text{UK} \ \text{then} \ 5000 \ \text{else} \ 10000;
\]

(Throughout this paper we use the operator ‘==’ to test for equality.) If the location of a particular project P is uncertain and is given by

\[\text{location} \ P = \text{Oneof} \ [\text{UK,France,Germany}]\]

then an answer to the query

\[\{\text{location} \ P,\text{expense\_allowance} \ P\}\]

should, in some way, express the fact that \{UK,5000\}, \{France,10000\} and \{Germany,10000\} are the only three possible answers; and, for example \{UK,10000\} is not a possible answer.

Incomplete or partial information of the kind we have described should not be confused with default information in the sense discussed by Reiter (1980), which is a quite distinct concept, and is discussed in Section 8. Neither, as is pointed out by Lipski (1979), should it be confused with ‘fuzzy sets’.

Other developments of the functional approach such as O\textsuperscript{2}FDL (Mannino et al., 1990) and FAD (Bancilhon et al., 1987) have not, as far as we are aware, addressed problems of incomplete information of the type we consider here. There is now an extensive literature on operations on null values and other forms of incomplete information in databases (Liu and Sunderraman, 1990; Biskup, 1981, 1983; Imielinski and Lipski, 1981; Grant, 1979; Codd, 1979, 1986; Date, 1982; Reiter, 1984, 1986; Morrissey, 1990; Grahe, 1984; Levesque, 1984; Vassiliou, 1979, 1980).

### 3. PREVIOUS WORK USING TRUTH FUNCTIONAL LOGIC

A logical operator whose semantics can be described by a truth table is known as a truth functor. A system of logic is described as truth functional iff all its logical operators are truth functors. FDL, SQL and the other languages referred to in the previous section all use a truth functional logic for the evaluation of logical expressions. The introduction of the value null leads to a three-valued logic with truth values True, False and Maybe, the last being the value given to such expressions as \(\text{null} = 10\), \(\text{null} > 5\), etc. which may be either true or false depending on the unknown value represented by the null. [In this paper we assume that nulls are used to represent valid but unknown values, although other uses are possible (Codd, 1986).] This approach leads to anomalies which we now describe.

For convenience we shall use SQL in our discussion of truth-functional logic. However the problems we outline are common to all systems using this form of logic. In SQL queries are interpreted using a three-valued logic in which the operators and, or and not have the truth tables:

<table>
<thead>
<tr>
<th>P</th>
<th>not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>P and Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>P or Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

where T, M and F represent True, Maybe and False. ‘Select’ expressions retrieve only those tuples for which the ‘where’ clause evaluates to True (Chamberlin, 1980).

This method can give results which run counter to intuition (Codd, 1979). For instance consider the relation

<table>
<thead>
<tr>
<th>Staffdata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>Michael</td>
</tr>
<tr>
<td>Joanna</td>
</tr>
</tbody>
</table>

and the query

\[\text{Select} * \text{from Staffdata where} (\text{Age} > 23) \text{or not} (\text{Age} > 23)\]
This should, intuitively, select every tuple in Staffdata because its 'where' condition seems a tautology. However, when we test the 'Joanna' tuple, the 'where' condition will be evaluated as:

\((\text{null} > 23) \lor \neg(\text{null} > 23)\)

\(\rightarrow \text{Maybe or Maybe}\)

\(\neg\text{Maybe}\)

and hence the tuple will \text{not} be selected.

Two important points must be made:

1. We cannot 'fix' this problem by changing the truth tables. Whereas, intuitively, the expression

\((\text{null} > 23) \lor \neg(\text{null} > 23)\)

should evaluate to \text{True}, the expression

\((\text{null} > 23) \lor \neg(\text{null} > 23)\)

should evaluate to \text{Maybe}, which is what the three-valued logic actually gives. We cannot define truth tables which give the value of the first expression as \text{True} and that of the second as \text{Maybe} because, in both cases we end up evaluating

\((\text{Maybe or Maybe})\)

The problem is the assumption that \((A \lor B)\) can always be evaluated by determining the values of \(A\) and \(B\) and then consulting a truth table. When both \(A\) and \(B\) evaluate to \text{Maybe}, then sometimes \text{True} would be the intuitively correct result and other times \text{Maybe}. The problem we illustrate results from the use of a many-valued truth functional logic. It does \text{not} result from the \text{particular} truth tables used and cannot be solved by using different tables and/or introducing further truth values.

2. The difficulty is not limited to tautologies, and can \text{not} be dismissed by saying that a user would not, in practice, introduce a tautology as a selection condition. Consider an example, adapted from one suggested by Vassiliou (1979), in which a life insurance company database contains the table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Male</th>
<th>Drinker</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Michael</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Joanna</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Chris</td>
<td>null</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

If the company decides that men who drive and women who drink are high risks, and creates a view containing only such people using

```
Create view High_risk
as select * from Risk_factors
where (Male='Yes') and (Driver='Yes')
or (not (Male='Yes'))
and (Drinker='Yes')
```

then, for the 'Chris' tuple, the 'where' condition will evaluate as

\((\text{Maybe and True}) \lor (\text{Maybe and True})\)

which gives \text{Maybe}, and hence this tuple will be excluded. However, this is intuitively wrong. We may not know Chris's sex but since Chris is both a drinker \text{and} a driver we do not need to know. Chris is necessarily high risk since if he is male then he is a male driver, and if she is not male then she is a non male drinker. Again, the problem is that the logic used is truth functional. An expression of the form \((A \lor B)\) or \((C \land D)\) has been evaluated by evaluating \(A\), \(B\), \(C\), and \(D\), and then using truth tables.

4. PREVIOUS WORK USING MODAL LOGIC

The language described in this paper is similar, in its approach, to theoretical work undertaken by Lipski (1979, 1981), who describes a treatment of incomplete information in the restricted context of a relational database consisting of a single relation. Such a database can contain partial information and a \textit{completion} of a database \(D\) is defined to be a database in which all the partial information in \(D\) has been replaced by definite information in a manner consistent with \(D\). Thus if \(D\) contains the information that John's age is either 32 or 33 then there is at least one completion of \(D\) in which John is aged 32, one in which he is 33, but none in which he has any other age. An \textit{extension} of \(D\) is defined to be a database containing information which, although it may still be partial, is consistent with, and more specific than, that contained in \(D\). (Note that the meaning assigned by Lipski to the word 'extension' differs from the meaning it usually has in a database context, where it means the data itself, as opposed to the schema). Lipski then defines modal operators which work by considering the value of logical expressions in all extensions, or in all completions of a database.

The approach we have adopted has some similarities to that used by Lipski. Specifically it contains certainty and possibility operators and uses completions of a database to define the semantics of incomplete information. However, Lipski's work is confined to the context of a database containing a single relation with a restricted query language, whereas Fudal provides a computationally complete query language over a functional database with rich semantics. For this reason Lipski's work cannot be used as a formal basis for Fudal although it has influenced the work we describe here. We do not attempt to establish any precise formal relationship between the certainty and possibility operators used in Fudal and the modal operators defined by Lipski, nor with those used in formal systems of Modal Logic such as \(T\), \(S4\) or \(S5\) (Hughes and Cresswell, 1972; Chellas, 1980).

5. THE LANGUAGE FUDAL

Fudal handles incomplete information by introducing certainty and possibility operators into a functional
language, along with a new kind of null value. Section 5.1 introduces our notation and briefly reviews those features not unique to Fudal. In Section 5.2 we consider the issue of lazy and eager evaluation. Section 5.3 explains how to query a Fudal database and Section 5.4 details the aspects of Fudal which are unique to the language. Section 6 gives examples of its use.

5.1. Notation and general features

The data types of Fudal are the arbitrarily nested lists, products, and sum types of functional programming with base types integer, boolean and string. Lists are written using square brackets; \([1,30,2,5]\) is a list of type \([\text{integer}]\), \([]\) is the empty list; \(h : t\) denotes a list with head \(h\) and tail \(t\), the colon being an infix form of \(\text{Cons}\). Tuples are written using curly brackets; \(\{1, \text{'Fudal'}, \text{true}\}\) is a tuple of type \(\{\text{integer}, \text{string}, \text{boolean}\}\). Sum types are used to represent entity types, each instance being represented by a data constant of that sum type as in (in general, the elements of a sum type are referred to as data constructors; a data constant is a data constructor which takes no arguments.): 

\[
\text{student} :: \text{Matthew.Mark.Luke};
\]

Further instances being subsequently added by, for example, the instruction \(!\text{create}\ \text{student}:: \text{John. Peter};\) or deleted by, for example \(!\text{delete}\ \text{student}:: \text{Matthew};\).

Function types are written as in:

\[
\text{age : student} \to \text{integer};
\]

\[
\text{total : [integer]} \to \text{integer};
\]

\[
\text{map : (\%a} \to \%b) \to \%a \to \%b;\]

where identifiers beginning with \(\%\) identify polymorphic type variables. Note that \(\text{map}\) is a higher-order function, because its first argument is itself a function. Equations defining functions are written as in:

\[
\text{age Mark} = 32;
\]

\[
\text{age Luke} = 45;
\]

\[
\text{total} [] = 0;
\]

\[
\text{total} (h : t) = h + \text{(total} t);\]

\[
\text{map } f (\text{hit}) = (fh)(\text{mapft});
\]

\[
\text{map } f [] = [];\]

When more than one equation matches the arguments of a function, a best fit algorithm, adapted from that described by Field and Harrison (1988), is used to resolve the ambiguity.

Function definitions need not be complete. Thus, for some arguments, pattern-matching may fail because there is no equation defining the value of the function for those arguments. In this paper we shall not deal with this possibility, nor with the possibility that evaluation of an expression may not terminate. Both these subjects are considered in depth by Sutton (1993).

When a sum type is declared, the user may stipulate that it will contain only data constants by adding the qualifier /simple to its declaration. For any such sum type \$\$, Fudal automatically maintains a function \( \text{all}_S \) which takes no arguments and returns a list containing all data constants of that type. For example, if we had added \(/\text{simple}\) to the declaration of \text{student} above, then \( \text{all}_\text{student} \) would evaluate to \([\text{Mark, Luke, John, Peter}]\).

The ellipsis operator ‘. . ’ allows us to define a list containing all the integers in a certain range. For example \(4 . . 8\) evaluates to \([4,5,6,7,8]\). A list comprehension is a pair of square brackets enclosing an expression which is followed by ‘filters’ and/or ‘generators’ each of which is preceded by a vertical bar. For instance

\[
[x | x \leftarrow \text{all}_\text{student} \mid (\text{age} x) > 30];
\]

evaluates to a list of all students aged over 30. Here \(x \leftarrow \text{all}_\text{student}\) is a generator and \((\text{age} x) > 30\) is a filter.

5.2. Lazy and eager evaluation

A definition of a functional programming language should allow the user to determine whether any given function will evaluate its arguments lazily or eagerly (Peyton-Jones, 1987). However, all the examples in this paper give the same result under either eager or lazy evaluation. We thus avoid, in this paper, committing ourselves to either method. However, a choice of evaluation method must ultimately be made (Sutton 1993). In conventional functional programming languages the choice of lazy or eager evaluation may determine whether evaluation of an expression terminates, but not its value if it does. This is a consequence of the Church-Rosser property (see, e.g. Peyton-Jones, 1987). However, in Fudal, as is demonstrated by Sutton (1993), there are certain examples, albeit rather contrived and exceptional, of expressions which yield different values under lazy and eager evaluation.

5.3. Querying a Fudal database

A Fudal database consists of a set of function definitions and is queried by entering an expression, the value of which is the answer to the query. Any expression that can be evaluated can be used as a query. In consequence Fudal, like FDL, offers a computationally complete query language. It is never necessary for Fudal queries to be embedded in some other programming language as is the case with, for example, SQL.

5.4. Features unique to Fudal

In Fudal nulls take the form \(\text{Oneof} \ L\), where \(L\) is a closed list-valued expression. ['Closed' here means that the expression does not contain any free variables. Allowing \(L\) to be an\(y\) closed list-valued expression can make nulls difficult to interpret, for instance \(L\) may refer to some function whose definition itself contains a \(\text{Oneof} \ \text{null}\). However such difficulties will not arise in the examples presented in this paper. They are discussed by Sutton (1993).] Such an expression is a null denoting a
value which is not known precisely, but is known to be a member of the list \( L \). Thus

```
age::person -> integer;
age John = 35;
age Tracey = Oneof (18 . . 30);
age Michael = Oneof [33,35,37];
age Julie = Oneof (0 . . 120);
```

 specifies John's age as 35, Tracey's as between 18 and 30, Michael's as either 33, 35 or 37 although we do not know which and Julie's age as completely unknown, although we have assumed that no one is older than 120 (or younger than 0).

We say that a completion of a database \( D \) is a database in which every expression of the form \( \text{Oneof } L \) has been replaced by a substitute, by which we mean a value from the list \( L \). Thus a completion of database 1 is:

```
age::person -> integer;
age John = 35;
age Tracey = 18;
age Michael = 37;
age Julie = 18;
```

A Fudal expression may have different values in different completions of the database. If such an expression is posed as a query then its answer will be \( V_1 \text{ OR } V_2 \text{ OR } \ldots \) where \( V_1, V_2, \ldots \) are the results of reducing the expression in each completion, with duplicates removed. For example, in database 1, the query \( (\text{age Michael}) + 5 \) has the answer 38 OR 40 OR 42. We shall refer to \( V_1, V_2, \ldots \) as the possible values of the expression or as the possible answers to the query.

We shall not, in this paper, consider the case in which an expression cannot be evaluated in some completions of the database, this eventuality being examined in detail by Sutton (1993). The order in which possible answers to a query are presented will not be important in any of the examples we consider here but, again, is considered by Sutton (1993).

5.5. List-valued queries

The semantics outlined in the previous section create a problem when answering list-valued queries. Suppose we have a sum type \( \text{person} \) and a function \( \text{age} \) and that there are \( n \) people whose age is completely unknown (everyone else's age being known with certainty). Consider then the query

\[
[x \mid x <- \text{all_person} \mid (\text{age } x) > 28]
\]

For each of the \( n \) people of unknown age there will be a completion in which that person is a member of the list generated by the query and a completion in which he is not. However this means that there are \( 2^n \) possible answers to the query, each of which is a list. For instance over database 1 this query gives 4 possible answers, i.e. \([\text{John, Michael}]\) OR \([\text{John, Michael, Julie}]\) OR \([\text{John, Tracey, Michael}]\) OR \([\text{John, Tracey, Michael, Julie}]\). As \( n \) becomes large this situation will clearly become unmanageable. The certainty and possibility operators introduced in the next section can be used to render this problem tractable.

5.6. Certainty and possibility operators

In Fudal, a boolean expression may contain the operators \( \text{Surely} \), or \( \text{Maybe} \). If \( P \) is a boolean expression which can be evaluated in all completions of the database then \( \text{Surely} P \) evaluates to true if \( P \) evaluates to true in all completions of the database, and to false otherwise, whereas \( \text{Maybe} P \) evaluates to true if there is at least one completion in which \( P \) evaluates to true, and to false otherwise. We shall not consider here the situation in which \( P \) cannot be evaluated in some completion of the database, either because of a failure of pattern matching or because evaluation of \( P \) does not terminate in that completion. This eventuality is considered by Sutton (1993).

So, for database 1, the expression

\[
\text{Maybe } ((\text{age Michael}) == 33)
\]

evaluates to true because there are completions of the database in which \( ((\text{age Michael}) == 33) \) is true. However

\[
\text{Surely } ((\text{age Michael}) == 33)
\]

evaluates to false because \( ((\text{age Michael}) == 33) \) is not true in all completions of the database. In contrast \( \text{Surely } ((\text{age Michael}) > 30) \) evaluates to true because \( ((\text{age Michael}) > 30) \) is true in all completions of the database.

We shall refer to \( \text{Maybe} \) and \( \text{Surely} \) collectively as modal operators and to expressions of the forms \( \text{Maybe} P \) and \( \text{Surely} P \) as modal expressions. (However, as mentioned in Section 4, we have not attempted to establish any precise relationship between our modal operators and those used in formal systems of modal logic.) A modal expression need not be closed. That is to say it can contain variables that are introduced as formal parameters of a function definition, or on the left hand side of a generator in an enclosing list comprehension. Such variables must be replaced by actual arguments before the modal expression can be evaluated. As mentioned in Section 5.2 all the examples in this paper give the same result regardless of whether this substitution is performed lazily or eagerly.

Returning to the example of Section 5.5, we note that the query

\[
[x \mid x <- \text{all_person} \mid \text{Surely}((\text{age } x) > 28)]
\]

has only one possible answer as opposed to the \( 2^n \) possible answers to the query of Section 5.5. For instance, over database 1, the query above would give...
the single answer [John, Michael]. Similarly if we pose
the query
\[ \{ x \mid x < - \text{all\_person} \mid \text{Maybe}((\text{age } x) > 28) \} \]
we get the single, list-valued, answer
[John, Tracey, Michael, Julie]
If we wish to know who is possibly but not definitely older
than 28 we pose the query
\[ \{ x \mid x < - \text{all\_person} \mid \text{Maybe}((\text{age } x) > 28) \text{ and }
\quad (\text{not Surely}((\text{age } x) > 28)) \} \]
which again gives a single list-valued answer, i.e.
[Tracey, Julie]

In addition to the three boolean operators and, or and
not Fudal includes operators representing implications
and ‘entailment’ (Hughes and Cresswell, 1972;
Chellas, 1980). If \( E_1 \) and \( E_2 \) are boolean expressions
then \( E_1 \rightarrow E_2 \), which is read as \( E_1 \text{ implies } E_2 \), is
exactly equivalent to \((\text{not } E_1) \text{ or } E_2\). The expression
\( E_1 \rightarrow E_2 \) is read as \( E_1 \text{ entails } E_2 \) and is exactly
equivalent to \( \text{Surely}(E_1 \rightarrow E_2) \). In other words it
means that \( E_1 \) implies \( E_2 \) in all completions of the
database.

5.7. Conditional information

In Fudal an expression of the form \( \text{Oneof } L \) can appear
anywhere where an element of the list \( L \) would be legal.
This allows Fudal to express forms of incomplete
information that cannot be conveyed in previous work,
e.g. that of Lipski (1979, 1981). For example we can use
if expressions to specify that a certain state of affairs is
possible only under certain circumstances. As an
illustration suppose that Tracey’s age is in the range
18 . . 30 and that if Tracey’s age is 21 then so is Julie’s, but
otherwise Julie’s age could be anything between 0 and
120. These facts may be expressed as
\[
\begin{align*}
\text{age Tracey} & = \text{Oneof} (18 . . 30) ; \\
\text{age Julie} & = \text{if} ((\text{age Tracey}) == 21) \text{ then } 21 \\
& \quad \text{ else } \text{Oneof} (0 . . 120) ;
\end{align*}
\]
Similarly, to express the fact that, say, Sarah is 5 or 6
years older than Tracey we enter
\[ \text{age Sarah} = (\text{age Tracey}) + \text{Oneof}[5,6] ; \]

5.8. Inverse functions in Fudal

In order to use Fudal as a database language it is
necessary to maintain inverses of at least some of the
functions which define the extensional database. If list-
valued inverse functions are used we must take care to
avoid the combinatorial problems described in Section
5.5.

In Fudal two inverses may be maintained for a
function \( f \), of type \( t \rightarrow s \). These are the ‘Maybe
inverse’ function \( \text{minv}_f \), and the ‘Surely inverse’

\[
\begin{align*}
\text{minv}_f &: s \rightarrow [t] ; \\
\text{minv}_f y &= \{ x \mid x < - \text{Dom\_f} \mid ((f x) == y) \} ; \\
\text{sinv}_f &: s \rightarrow [t] ; \\
\text{sinv}_f y &= \{ x \mid x < - \text{Dom\_f} \mid (\text{Surely}(f x) == y) \} ;
\end{align*}
\]
where \( \text{Dom\_f} \) is a list containing all values in the domain
of the function \( f \) (i.e. all values \( x \) for which \( f x \) has a
defining equation).

As an illustration, consider the inverse functions
\( \text{minv\_age} \) and \( \text{sinv\_age} \) for database 1 in section 5.4.
The expression \( \text{sinv\_age} 35 \) evaluates to [John] and for
any integer \( n \) not equal to 35 the expression \( \text{sinv\_age} n \)
evaluates to []. However \( \text{minv\_age} 35 \) evaluates to
[John, Michael, Julie]. In fact Julie will be a member of
the list returned by \( \text{minv\_age} n \) for all \( n \) such that
0 \leq n \leq 120.

To indicate that Fudal should maintain the inverses of
a function the user must add the qualifier \text{/invertible}
in its type declaration, as in
\[ \text{age} : \text{person} \rightarrow \text{integer} \text{/invertible} ; \]
The function to be inverted must be a type of \( t \rightarrow s \) such
that \( t \) is either integer, boolean or a sum type
containing only data constants and \( s \) is first order (i.e.
not a function type).

Furthermore, the right hand sides of the function’s
defining equations must contain only data constructors,
constants, the names of other functions (either user-
defined or built-in) and the reserved word \text{Oneof}. If a
variable appears on the right hand side of an equation
defining a function declared as \text{/invertible} then an
error will be reported.

6. SOME EXAMPLES

Consider our Staff data example from Section 3 which,
in Fudal, is specified by:
\[
\begin{align*}
\text{employee} &:: \text{John, Michael, Joanna} \text{/simple} ; \\
\text{age} &: \text{employee} \rightarrow \text{integer} ; \\
\text{age John} &= 20 ; \\
\text{age Michael} &= 25 ; \\
\text{age Joanna} &= \text{Oneof} (0 . . 120) ;
\end{align*}
\]
where we assume that Joanna’s age, although unknown,
cannot be greater than 120. If for a query we used the list
comprehension
\[ \{ x \mid x < - \text{all\_employee} \mid (\text{age } x) > 23 \} \]
then Fudal would give the result [[Michael]] OR [Michael, Joanna]
showing that for some completions of the database the
result is [Michael] and for the others
[Michael, Joanna].
However the query
\[ \{ x \mid x < - \text{all\_employee} \mid (\text{Surely}(\text{age } x) > 23) \} \]
would give the result [Michael] since the query is now
requiring the condition to be true in all completions
of the database. Similarly

\[ x \mid x <\text{- all\_employee} \mid \text{Maybe}((\text{age } x) > 23) \]

would give [Michael, Joanna].

Consider now the case we discussed with a tautologous
condition, i.e. the query

\[ x \mid x <\text{- all\_employee} \mid ((\text{age } x) > 23) \text{ or } (\text{not } ((\text{age } x) > 23)) \]

There are two classes of database completions in relation
to this condition, those for which Michael and Joanna
satisfy the first term of the condition and John satisfies
the second, and those for which Michael satisfies the
first term and John and Joanna the second. Thus for
all completions the result is [John, Michael, Joanna]
which accords with our intuition. Joanna is not ignored
as with FDL and SQL. Note that there would have been
no difference had the condition been prefixed by the
Surely operator since the condition evaluates to true for
all instances of x in all completions.

The reader can readily verify that our Risk\_f actors
example from Section 3 can be translated into Fudal
in a straightforward manner so that Chris is picked
out as a high-risk client and not omitted as with the
corresponding SQL query.

As a more complicated example, suppose that we
know that a murder has been committed either by a
certain Colonel Mustard using a revolver in the study,
or by Miss Scarlett, in either the study or the drawing room,
using a weapon about which we know only that it was
not a rope [A possible situation in the board game
'Cluedo'. Other Cluedo examples are given by Sutton
and King (1992a,b).] This situation may be represented
by:

\[
\begin{align*}
\text{person} & : \text{Col\_Mustard, Miss\_Scarlett,} \\
& \quad \text{Prof\_Plum, Rev\_Green,} \\
& \quad \text{Mrs\_White /simple;}
\end{align*}
\]

\[
\begin{align*}
\text{utensil} & : \text{Lead\_Piping, Revolver, Rope,} \\
& \quad \text{Crowbar, Aubergine /simple;}
\end{align*}
\]

\[
\begin{align*}
\text{scene} & : \text{Study, Drawing\_room, Kitchen,} \\
& \quad \text{Lounge, Library /simple;}
\end{align*}
\]

\[
\begin{align*}
\text{murderer} & : \text{person;}
\end{align*}
\]

\[
\begin{align*}
\text{murderer} & = \text{Oneof [Col\_Mustard, Miss\_Scarlett];}
\end{align*}
\]

\[
\begin{align*}
\text{weapon} & : \text{utensil;}
\end{align*}
\]

\[
\begin{align*}
\text{weapon} & = \text{if } (\text{murderer} = \text{Col\_Mustard}) \\
& \quad \text{then Revolver} \\
& \quad \text{else Oneof [Lead\_Piping, Revolver,} \\
& \quad \quad \text{Crowbar, Aubergine];}
\end{align*}
\]

\[
\begin{align*}
\text{room} & : \text{scene;}
\end{align*}
\]

\[
\begin{align*}
\text{room} & = \text{if } (\text{murderer} = \text{Col\_Mustard}) \\
& \quad \text{then Study} \\
& \quad \text{else Oneof [Study, Drawing\_room];}
\end{align*}
\]

In which we have arbitrarily limited the types person,
utensil and scene to contain only those values
indicated in their definitions above.

1. Suppose that we wish to know who could (possibly)
have committed the murder using the revolver. Now
for any person x the expression

\[
\text{Maybe}((\text{murderer} == x) \\
\quad \text{and } (\text{weapon} == \text{Revolver}))
\]

evaluates to true if there is a completion of the
database in which x committed the murder using the
revolver and false otherwise. So the list of all people
who could have committed the murder using the
revolver is the list of all people for whom the above
condition is true. Thus the query is:

\[
\begin{align*}
\text{[x} \mid x <\text{- all\_person} \\
\quad \text{| Maybe(} (\text{murderer} == x) \\
\quad \quad \text{and } (\text{weapon} == \text{Revolver})\])
\end{align*}
\]

which evaluates to

[Col\_Mustard, Miss\_Scarlett]

2. Suppose we were told that, if we knew the name of the
murderer, then, with the information already avail-
able to us, we would be able to deduce what weapon
had been used. Who could then be the murderer?
What this means is that if we let m denote the murderer
and w the weapon, then the following two statements
must be true:

(i) There is a completion of the database in which m is
the murderer.

(ii) In all completions in which m is the murderer, the
weapon is w.

Now (i) will be true if and only if the expression
\[
\text{Maybe}(\text{murderer} == m)
\]
will be true and (ii) will be true if and only if
\[
(\text{murderer} == m) \Rightarrow (\text{weapon} == w)
\]
evaluates to true. So, if we want to
know who could possibly have committed the
murder, we must look for values of m and w which
satisfy both of these conditions. This can be expressed
as the query

\[
\begin{align*}
\{m & \mid m <\text{- all\_person} \\
\quad \text{| Maybe(} (\text{murderer} == m) \\
\quad \quad \text{and } (\text{weapon} == w)\})
\end{align*}
\]

which evaluates to

[Col\_Mustard].

3. Suppose that we were told that, if we knew the
identity of the murderer, then we would still be
unable to say where the murder had been committed.
To know who might then have committed the murder
what query should we pose? From what we have been
told it follows that if we let m be the murderer then
there must be some room r such that:

(i) There is a completion of the database in which m
committed the murder in room r.

(ii) There is a completion in which m committed the
murder, but not in room r.
7. IMPLEMENTATION

In this section we sketch out an implementation of Fudal, concentrating on those aspects of Fudal that are new. We shall assume that the reader is familiar with the various techniques that may be used to store function definitions and to evaluate expressions in conventional functional languages (Peyton-Jones, 1987; Field and Harrison, 1988).

If Fudal did not contain the special operators Oneof, Surely and Maybe then its implementation would be straightforward. A number of well understood techniques (Field and Harrison, 1988; Peyton-Jones, 1987) could be used for the evaluation of queries and the storage of function definitions. We shall therefore imagine that we already have an implementation of Fudal that is sufficient to deal with definitions and queries that do not contain the special operators referred to above and then show how such an implementation can be extended to deal with these operators.

We first outline a treatment of Oneof nulls which, for simplicity, ignores the fact that there may be several ways of assigning substitutes to nulls which all yield the same possible answer to a query, since duplicate answers could be eliminated by recording answers as they occur. We then outline a treatment of the modal operators Maybe and Surely. In Section 7.1 we briefly discuss the optimisation of our implementation, which would otherwise be highly inefficient.

A unique integer subscript is assigned by the parser to each occurrence of the Oneof operator that appears in a query or function definition. In this section we shall sometimes include these subscripts when writing expressions and definitions. For instance

\[
\text{age John} = \text{Oneof}_1 [20,21];
\]
\[
\text{age Charlie} = \text{Oneof}_2 [30,31,32];
\]

indicates that the parser has assigned the subscripts 1 and 2 to the two Oneof nulls shown.

The evaluation of expressions is extended to cover Oneof nulls by including a completion stack. This is a stack of quadruples of the form \((i, S_i, R_i, Q'_i)\) where \(i\) is an integer and \(S_i, R_i, Q'_i\) are Fudal expressions.

The intention is that the completion stack defines an assignment of substitutes to nulls which is used in the calculation of a possible answer to some query \(Q\). The elements of a quadruple on the completion stack have the following meanings: \(i\) is the subscript of a particular Oneof null; \(S_i\) is the substitute assigned to that null during the evaluation of the possible answer we are currently computing; \(R_i\) is a list containing all the substitutes that could be assigned to the Oneof null in the evaluation of the possible answers we have not yet computed. \(Q'_i\) is the expression to which \(Q\) had been reduced when the Oneof null was encountered, i.e. when the expression \(Q\) had been reduced to \(Q'_i\), the next redex (i.e. the next subexpression to be reduced) was the Oneof null. For example suppose that the query \((\text{age Charlie}) + 5\) is posed and that, during the evaluation of a possible answer to this query, the quadruple \((2, 30, [31,32], \text{Oneof}_2 [30,31,32] + 5)\) is on the stack. This quadruple indicates that, during the evaluation of this possible answer, the Oneof null is to be replaced by 30; that in the calculation of subsequent answers it may be replaced by 31 or 32; and that, when the Oneof null was encountered, the query had been reduced to \(\text{Oneof}_2 [30,31,32] + 5\).

We modify the query evaluation procedure as follows. If a query \(Q\) has been reduced to \(Q'\) and if, at this stage, the next redex is of the form \(\text{Oneof}_i L\) then we apply the following algorithm. Note that, because of Fudal's type-checking, we can assume that \(L\) is list-valued:

\[
\begin{align*}
&\text{Examine the completion stack to see if it contains a quadruple } (i, S_i, R_i, Q'_i) \text{ where } j = i. \text{ If it does then replace the redex } \text{Oneof}_i L \text{ with } S_i \text{ and continue reduction of the expression } Q. \\
&\text{Otherwise reduce } L \text{ to weak head normal form (Peyton-Jones, 1987). In this form } L \text{ will either be of the form } h : t, \text{ in which case replace } \text{Oneof}_i L \text{ with } h \text{ and push onto the completion stack the quadruple } (i, h, t, Q'), \text{ or it will be an empty list in which case report an error.} \ (\text{We assume that the user will not intentionally use empty lists in the definition of nulls.})
\end{align*}
\]

\textbf{Algorithm 1}

(reduction of Oneof nulls).
We reduce \( Q \), applying algorithm 1 where necessary, until it has no further redexes. We have then calculated, and will output, one of the possible values of \( Q \).

In order to calculate the next possible value (if there is one) we apply the following algorithm:

1. If the completion stack is empty then no more possible values can be calculated so stop.
2. Otherwise, if the top record on the completion stack is \((i, S_i, R_i, Q'_i)\), then reduce \( R_i \) to weak head normal form. In this form it will be either an empty list, in which case pop the top record from the completion stack and go back to step (1), or else it will be of the form \( h : t \), in which case replace the top record on the stack by \((i, h, t, Q'_i)\).
3. Reduce the expression \( Q'_i \) and output the result as a possible value for the query.

**Algorithm 2**

(calculation of the next possible value).

Repeated application of algorithm 2 will result in all possible answers to the query being output (provided that the query can be evaluated in all completions of the database).

We now illustrate the method by considering the database:

\[
\text{person} :: \text{Lorraine, Tracey; age} : \text{person} \rightarrow \text{integer; age Lorraine} = \text{Oneof}_1 \{20, 21, 22\}; \text{age Tracey} = (\text{age Lorraine}) + \text{Oneof}_2 \{5, 6\};
\]

and the query \{age Lorraine, age Tracey\}. The reader will note that, in the course of this example, we will make assumptions about reduction order (i.e. about what the next redex is at any stage in the evaluation) and about what happens if the same subexpression is encountered more than once during reduction of a particular expression. None of these assumptions is crucial to an understanding of the example, whose purpose is to illustrate the operation of algorithms 1 and 2 above.

We shall represent the various stages of evaluation by figures of the form

<table>
<thead>
<tr>
<th>Completion stack</th>
<th>Expression being reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>{empty}</td>
<td>{age Lorraine, age Tracey}</td>
</tr>
</tbody>
</table>

Thus initially we have

\{age Lorraine, age Tracey\} reduces to \{Oneof, \{20,21,22\}, age Tracey\} Applying algorithm 1 then gives

\[
(1, 20, \{21, 22\}, \{\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey}\})
\]

\[
(20, \text{age Tracey})
\]

\( (20, \text{age Tracey}) \) is reduced to \( (20, (\text{age Lorraine}) + \text{Oneof}_2 \{5, 6\}) \) and thence to \( (20, \text{Oneof}_1 \{20, 21, 22\} + \text{Oneof}_2 \{5, 6\}) \). Algorithm 1 tells us that the \text{Oneof}_1 null must be replaced by 20, giving \( (20, 20 + \text{Oneof}_2 \{5, 6\}) \). Applying algorithm 1 to the \text{Oneof}_2 null then gives

\[
(2, 5, \{6\}, \{20, 20 + \text{Oneof}_2 \{5, 6\}\})
\]

\[
(1, 20, \{21, 22\}, \{\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey}\})
\]

\( (1, 20, \{21, 22\}, \{\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey}\}) \) gives

\( (20, 20+5) \)

We reduce \( (20, 20+5) \) to \( (20, 25) \) which is output as a possible answer to the query. We then apply algorithm 2 to give

\[
(2, 6, \{\}, \{20, 20 + \text{Oneof}_2 \{5, 6\}\})
\]

\[
(1, 20, \{21, 22\}, \{\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey}\})
\]

\[
(20, 20+6)
\]

Applying algorithm 1, \( (20, 20+\text{Oneof}_2 \{5, 6\}) \) reduces to \( (20, 20+6) \) and thus \( (20, 26) \) is output as a possible value. Applying algorithm 2 then gives

\[
(1, 21, \{22\}, \{\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey}\})
\]

\[
(\text{Oneof}_1 \{20, 21, 22\}, \text{age Tracey})
\]

Reduction continues until the remaining possible values, i.e. \{21, 26\}, \{21, 27\}, \{22, 27\} and \{22, 28\} have been output.

We now outline a method for evaluating expressions of the form \text{Maybe} \( E \) or the form \text{Surely} \( E \). We shall assume that \( E \) can be evaluated in all completions of the database. The possibility that \( E \) may be non-terminating or undefined in some or all completions of the database is considered by Sutton (1993).

When the evaluator is invoked for an expression of the form \text{Maybe} \( E \) it creates a new, initially empty, completion stack (without deleting any existing one). Using this new stack the evaluator calculates possible values of the expression \( E \) until either

1. one of these possible values turns out to be \text{true}, in which case the evaluator returns \text{true} as the value of \text{Maybe} \( E \), or
2. all the possible values prove to be \text{false}, in which case \text{Maybe} \( E \) is also \text{false}.

An expression of the form \text{Surely} \( E \) is treated in a similar way, the only difference being that now the value of this expression is \text{false} if one of the possible values of \( E \) is \text{false} and \text{true} otherwise.
7.1. Optimization

The method outlined in Section 7 is, in many cases, extremely inefficient. In particular it may consider a large number of assignments of substitutes to nulls in order to produce a much smaller number of possible answers to a query. As a simple example, suppose we have the database

```
person :: John, George, Paul, etc.
salary : person -> integer;
salary John = Oneof (1000 .. 5000);
salary George = Oneof (1000 .. 6000);
salary Paul = Oneof (1000 .. 7000);
```

then to answer the query \( (\text{salary John}) > 2000 \) the method above will consider all 4,001 possible assignments of substitutes to the null which appears in the definition of the John's salary, yet the query has only two possible answers: true and false.

In Sutton (1993) we discuss the optimization of this rather simple query and of some more complex ones. In particular we show that there is a large class of queries and sub-queries that can be converted into integer linear programming problems or, more specifically, that can be answered by determining whether there is a solution to a system of linear inequalities of the form

\[
\begin{align*}
 a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &> b_1 \\
\vdots \\
a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &> b_m
\end{align*}
\]

where \( \forall i, j \) such that \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \) the values \( a_{i,j}, x_i \) and \( b_j \) are integers.

**FIGURE 1.**

As an example consider the query

\[
[x \mid x <- \text{all_person} \\
| \text{Maybe}((3\ast\text{salary x}) + \text{salary Paul}) > 7000) \\
\text{and} (3\ast(\text{salary x}) - \text{salary(Paul)} + \text{salary John}) > 3000] \}
\]

over database 2.

It can readily be seen that determining whether any particular person belongs to the list produced in answer to this query is equivalent to determining whether a system of inequalities of the form indicated above has a solution. For instance the list will contain George iff there is a solution to

\[
\begin{align*}
 3x_2 + x_3 &> 7000 \\
 3x_2 - x_3 + x_1 &> 3000 \\
 1000 &\leq x_1 \leq 5000 \\
 1000 &\leq x_2 \leq 6000 \\
 1000 &\leq x_3 \leq 7000
\end{align*}
\]

for which \( x_1, x_2 \) and \( x_3 \) are all integers (informally \( x_1, x_2 \) and \( x_3 \) represent substitutes assigned to the Oneof nulls occurring in the salaries of, respectively, John, George and Paul). This can readily be converted to a problem in the form of Figure 1.

Determining whether a system of the form of Figure 1 has a solution is, in general, an NP-complete problem (Schrijver, 1986). However in many particular cases we can determine relatively easily that such a system does, or does not, have a solution. Moreover, Lenstra (1983) has demonstrated that, for any fixed integer \( n \) there is a polynomial-time algorithm which determines whether a set of linear inequalities in \( n \) variables, of the form illustrated in Figure 1, has a solution.

Informally the set of queries that can be converted to integer linear programming problems includes, but is not limited to, all queries where we wish to select all entities of a particular type whose attributes satisfy some set of linear equalities and/or inequalities, and where for any particular entity of this type, the values of the attributes involved are either known with certainty or are completely unknown or, in the case of integer valued attributes, are known to be in some interval \([l_{\min}, l_{\max}]\). Note that it is not necessary that the attributes involved be of type integer.

In (Sutton 1993) we describe how the optimization discussed in this section can be implemented. The optimization is performed at run-time and will not interfere with other optimizations performed at compile-time. The current version of Fudal uses an incomplete version of the optimization strategy described in Sutton (1993) and optimizes a useful subset of the queries that would be optimized by a full implementation.

8. CONCLUSIONS AND FURTHER WORK

We have presented here a way in which modal operators can be used in the context of the functional data model by their integration into a computationally complete functional database language. We have briefly described how this language is implemented. This first implementation, written in C under Unix and using our research group's existing persistent quadruple store software, is now being used for initial application experimentation.

Further work required includes a fuller specification of the semantics and larger scale application experiments. In Sutton (1993) we discuss a number of other suggestions for further work, including the incorporation of floating point numbers into Fudal. Whilst this work has been
carried out in the context of a persistent functional database language, it is possible to extend the ideas outlined here to other areas, for example relational database languages. This topic is briefly discussed by Sutton (1993).

Another subject for further research is the use of default values in Fudal. At present it is sometimes possible to specify defaults in Fudal. For instance if we added more entities of type person to database 1 and wished to add the default information that anyone whose age was not explicitly specified should be assumed to 18 then we could do so by adding the equation

$$\text{age } x = 18$$

However, defaults do not interact benignly with incomplete information as specified in Fudal. If we wished to specify that anyone whose age was not explicitly given should be assumed to be aged between 0 and 120 then we could not simply enter

$$\text{age } x = \text{Oneof}(0 \ldots 120);$$

The snag is that, in any completion of the database, this equation is replaced by the equation

$$\text{age } x = v;$$

where $v$ is an integer value in the range 0 to 120. This means that we are effectively assuming that all people whose age is unspecified have the same age, although we do not know what that age is. As an illustration, if $P_1$ and $P_2$ are people whose age has not been explicitly specified then the query age $P_1$ will, correctly, return 0 OR 1 OR ... OR 120, as will the query age $P_2$. However the query (age $P_1$) == (age $P_2$) will return true since although age $P_1$ has different values in different completions, it always has the same value as age $P_2$. Although a solution to this problem would not be particularly difficult to devise, its implementation would involve some cumbersome internal book-keeping and consequently has been left as a topic for further work.

ACKNOWLEDGMENTS

The authors wish to thank the Science and Engineering Research Council (SERC) and IBM UK Laboratories, Hursley, for their financial support of this work. We also extend our thanks to members of the TriStarp group and other colleagues for their encouragement and constructive criticism.

REFERENCES


Reiter, R. (1984) Towards a logical reconstruction of relational


