RELATIONAL ALGEBRAS, LOGIC, AND FUNCTIONAL PROGRAMMING

Patrick A.V. Hall
Cirrus Computers Ltd.,29/30 High Street, Fareham, PO16 7AD, England.

ABSTRACT

Relational algebras as developed by Codd and his followers are extended by noting an equivalence with functional languages. This leads to higher order relations, recursive definitions of relations, and the use of higher order relations as recursive data structures. This equivalence on the one hand enables the technology of databases to be used within the context of applicative languages, and on the other hand removes the adhoc mechanisms used for higher order operations in relational databases. This leads from 4th to 5th generation data management, exploiting the mathematical foundations of functional languages and logic programming, and the technology of relational database.

1.0 INTRODUCTION

Logic programming and functional programming have recently become very fashionable through interest in "5th generation" computing. Logic and functional programming have, of course, been around a long time. So has the application of computers to commerce, and the use of databases for that application. Logic programming has seen the challenge and responded to it [GM 78]. Functional programming appears not to have, except in the use of functional ideas in some query languages [BF 79, SHIP.81].

Yet it is databases that form the foundation of a very large bulk of computing applications, from COBOL systems to those using the most sophisticated data management software. In this area we are seeing the emergence of a large number of commercial systems, often styled "4th Generation", based on data dictionaries and providing various facilities through non-procedural or semi-procedural languages [e.g. GRAD 83, MCCR 80]. The facilities are often loosely founded on the relational theories of Codd [CODD 72] but add various additional facilities in ad hoc manner.

In this paper I start with relational databases, and interpret the relational algebra functionally, to extend this to embrace all the power and theoretical neatness of functional languages. In doing this my intentions are twofold:

- to provide a theoretical basis for the 4th generation data processing languages, to enable them to move into the 5th generation.
- to integrate functional, logic, and relational languages so that they can share technology.

This exploration is by no means complete, for a critical aspect, the time dimension and updates, has not been addressed. However this paper must be viewed as a necessary pre-requisite for that later study.

In section 2, the historical background to this study is described, and relationships to other work explained.

In section 3, the relational algebra is developed as an applicative language. This correspondence with applicative languages adds capabilities to the algebra, with recursive definitions, higher order relations, and recursive data structures, presented in Section 4 and 5. An example is given in Section 6.

2.0 BACKGROUND

During the development of the research relational database system PRTV [TODD 76] at IBM Peterlee, the relational algebra was used as the data manipulation language. In its original form [CODD 70], relations had been defined as subsets of cartesian products on a number of domains. In order to achieve a position independent notation for relations, the 'columns' were distinguished by the domain name, or where several domains were the same, by a 'role name'. However there were some technical problems with the name inheritance in the 'join' operations of the relational algebra. The key relational operation is the 'equi-join', where two relations are combined in a cartesian

product constrained by equality of values between matching domains of the two relations. The notation in the relational algebra used for the PRTV was a cartesian product associated with a 'selection' which indicated the columns to be equated, for example:

R1 * R2 : A=B

A and B are role names, where A is a column of R1, and B a column of R2. The result is a relation of n + m columns (n = number of columns of R1 and m = number of columns of R2. Two of these columns are identical - columns A and B, and clearly one should be removed, but which one? In the PRTV this was specified using a project operation, but we felt there must be better ways of solving the role inheritance problem.

At the same time Stephen Todd was investigating how to "escape" from the relational algebra into PL/I, and one method he came up with replaced a relation by a PL/1 procedure call. An equivalence between relations and procedures became evident, and we sought ways of reformulating the relational algebra so that the fact that a PL/I procedure was invoked would be transparent.

The result was a "generalised" relational algebra (see example below) in which:

- matching role names in a cartesian "natural" join caused an equijoin on these columns.
- the number of operations was reduced to four.

This was published at a conference in 1975 [HHT 75] together with an analysis of the restrictions caused by the directional properties of procedures. The paper was presented in a highly analytical manner, and is not easy to read.

An example of the generalised algebra is:

R1(X,Y,A,C)*R2(P,A,Q,X,F)

which causes an equi-join on columns A and X in the two relations, yeilding a result:

R3(X,Y,A,C,P,Q,F)

The resemblance of this to the predicate logic formula:

P1(x,y,a,c) & P2(p,a,1,x,f)

was initially coincidental, but it was soon pointed out to us by Bob Kowalski. The correspondence has, of course, now been explored in depth [e.g. GM 78].

At this time (1975) applicative languages were beginning their regeneration led by, among others, John Bachus [BACH 78]. It struck me then that there was a correspondence between the generalised join and functional application, but other things intervened. Then, recently, the appearance of two books [HEND 80, DHT 82] has renewed my interest and the result is this paper.

Applicative languages as they have developed today [as exemplified by the collection DHT 82] are still very much moulded in the LISP tradition. The critical limitation that this imposes is their dependence upon lists. For some applications lists are an appropriate structure, but all too frequently they are inappropriate, both as a means of expressing the problem, and as a representation to assist computation. Lists have even intruded into PROLOG! [e.g. CM 81]. Indeed one could conjecture a paper titled, "Lists considered harmful". For storage of information for commercial data processing lists are particularly inadequate. Relations as sets have no single ordering, and lists imposes one. Relations may be complexly interconnected: at the user level this interconnection may be considered associative, but internally they need explicit connections not permitted in lists. The structure present in the problem must be capable of direct expression, extra structure to aid processing must be capable of addition, extraneous structure which interferes with both must not be imposed. Thus a major motivation of this paper is to move away from lists while still remaining an applicative language.

A recent stimulating paper by Bob Kowalski [KOWA 83] notes the equivalence between predicates (relations) and functions, and explores logic programming capabilities to match functional programming. Here I go beyond that paper to produce a full relational applicative language. Perhaps this may in turn enrich logic programming.

Uniting relations with functional languages leads us to higher order relations, relations not in Codd 1st normal form. There has been a recent move in this direction, as relational databases have been applied to engineering applications [eg. LP 83, AB 84], and this paper completes this cycle of development with an algebra of relations appropriate to general hierarchical relations.

3.0 A RELATIONAL APPLICATIVE LANGUAGE

The basic language consists of relations, expressions involving these relations, and definitional equations. Examples are shown in figures, to be motivated in the discussion below.

The basic syntax is:

```
relation-definition ::= typed-relation "="
                        expression
typed-relation ::= relation-name relation-type
relation-type ::= "(" variable ":" type {","
                  variable ":" type} ")"
expression ::= expression operator expression
               |modadic-op expression
               |relational-constant
               |untyped-relation
               |"(" expression ")"
operator ::= "*" | "+"
monadic-op ::= "-"
relational-constant ::= relation-type "{" tuple
                        {"," tuple} "}"
tuple ::= "(" value {"," value} ")"
untyped-relation ::= relation-name "(" variable
                     {"," variable} ")"
```

Notation: terminals have been included in quotes, braces "{" and "}" are metasymbols and have been included in quotation marks where they are required as terminal symbols.

A relation is interpreted as a subset of the cartesian product of the types. A type is a set: initially this is not particularly significant but later we will need to invoke a complete

type-lattice to make our language general. Position in the parameters list is significant, as in programming languages. The variables are used for binding. A relation may be represented either extensionally, as an explicit set of "tuples" or intensionally by some rule such as a relational expression. The type of a relation is the list of its parameters and their type, where the variables are not significant but position is. Relation constants can be written as sets of tuples conforming to the relation type immediately preceding it. Alternative interpretations of relations are, as Predicates in Logic, and procedure calls in programming languages.

The operations are:

- `*` relational composition
- `+` relational alternation
- `-` relational complement

Alternation is normal set union for relations of the same type; inhomogeneous combinations are not permitted (though they could be allowed). Complement is relative to the base type of the relation, the usual set complement. Composition is the key operation, and requires explanation.

FIGURE 1
Example of Relational Composition as an Equijoin

```
READINGS                    SINE
(Pressure, Angle  )         (Angle,   Value)
 :real     :real            :real     :real
{(    1.3,     10 ),       {(    0, 0.0000 ),
 (    4.2,     20 ),        (   10, 0.1736 ),
 (    8.7,     45 ),        (   20, 0.3420 ),
 (   11.1,     70 )}        (   30, 0.5000 ),
                            (   40, 0.6428 ),
                            (   50, 0.7660 ),
                            (   60, 0.8660 ),
                            (   70, 0.9397 ),
                            (   80, 0.9848 ),
                            (   90, 1.000  )}
define
ANALYSIS(Pressure:real, Angle:real, Value:real)
= READINGS (Pressure, Angle) * SINE (Angle, Value)

ANALYSIS is {(1.3  , 10 , 0.1736  ),
             (4.2  , 20 , 0.3420  ),
             (11.1 , 70 , 0.9397  )}
```

In relational databases, composition covers the normal equijoin, where the columns to be equated are indicated by coinciding variables in the columns. Figure 1 shows an example. However, composition also covers set intersection, as shown in Figure 2, and selection as in Figure 3. From these examples it should be clear that the "tables" could be "virtual", and rather than being defined extensionally, could be defined intentionally by a program. The sine relation of figure 1 could, indeed should, have been a library program, and we used an implicit definition in Figure 3 to emphasise the use of composition as selection. We will use arithmetic and logical expressions informally in later examples. It was precisely for this composition operation that the original generalised algebra was developed [HHT 75].

FIGURE 2
Example of Relational Composition as Intersection

```
MORE_READINGS(Pressure:real, Angle:real)
 = (Pressure:real,angle:real)
    {(0.9,10),(1.3,10),(3.6,15),(4.2.25),(8.7,45)

define
CHECKED_READINGS(p:real, a:real)
 = MORE_READINGS (p,a) * READINGS(p,a)

CHECKED_READINGS is {(1.3,10),(8.7,45)}
```

FIGURE 3
Example of Relation Composition as Selection

```
define
GOOD_RANGE(Pressure:real, Angle:real)
   =angle<50 and Pressure>1.0
GOOD_READINGS(P:real, A:real)
   = READINGS(P,A)*GOOD_RANGE(P,A)

GOOD_READINGS is {(1.3, 10),(4.2, 20),(8.7,45)}
```

FIGURE 4
Relational Composition as Functional Application

```
FUNCTIONS
            Sine : R -> R
            Square_root : R+ -> R
            f : R -> R

f(angle) = square_root (sine (angle))

RELATIONS
Functions are 'converted' to relations by placing
an explicit results parameter last.

            SINE (angle:real,value:real)
            SQUARE_ROOT (Number:real, root:real)

F(Angle:real,Value:real, Result:real)
 = SINE(Angle,value) * SQUARE_ROOT(value,result)
```

The interpretation of the operations in logic programming are straight forward:

- `*` AND
- `+` OR
- `-` NOT

The interpretation in conventional (functional) programming languages is not so straightforward. The '*' allows functional composition, '+' allows

alternatives to be combined as in "case" and "if...then...else" constructs. All three together allow logic conditions. Figures 4 and 5 show examples, expressed both in the functional programming notation of Henderson [HEND 80] and in the relational notation.

However, our composition operator is far more powerful than traditional functional composition, for it allows functions to be used "backwards". (Kowalski [KOWA83] calls this "invertibility")

FIGURE 5 - Conditional Applicative Expressions

```
FUNCTIONS
            Sine : R -> R
            Square_root : R+->R

g(angle)= {if x> 0 then square_root(x)
                   else square_root (-x)
              where x=sine(angle) }

RELATIONS
            SINE (angle:real, value:real)
            SQUARE_ROOT (Number:real, root:real)

G(angle:real, value:real, result:real)
 = ((NON_NEG(value)*SQUARE_ROOT(Value,result)
  + NEG(value)*NEGATE(value,x)*SQUARE_ROOT(x,result)
   )*SINE(angle,value)
or
 = ((value>0)*SQUARE_ROOT(value,result)
  + (value<0)*(x=-value)*SQUARE_ROOT (x,result)
   )*SINE(angle,value)
```

An example of the use of SINE to compute arc-sine is shown in Figure 6. Now tabular functions are clearly able to be used bidirectionally, but if represented intensionally by programs would typically be only useable in a limited direction, This observation then leads to a further level of theory around the "effectiveness" of representations which was very thoroughly characterised in the 1975 paper [HHT 75]. This theory would be vital for any functional programming interpretation of the relational applicative language.

FIGURE 6 - Using a table of SINES as Arcsine

```
SINE - as in Figure 1
given table of DATA values
DATA(setting:real,measure:real)=(...) {...}
define
ANALYSIS(setting;real,measure:real,angle:real)
 = DATA(setting,measure)*SINE(angle,measure)
```

One further operation is necessary. This is the "projection" of relational algebras. Rather than have a special operator for this, we will make it a property of the definitional equality "=". The relation being defined and placed on the left-hand side of "=" lists its columns, thus determing the positional significance of these: in principle, all variables on the right-hand side should appear. However, if some are omitted, this means that they are projected out. Thus in Figures 4 and 5 the intermediate "value" can be omitted, to make the relational expression identical in semantics to the functional programming expression.

As observed by Codd [CODD 72], projection involves existential quantification, thereby giving the relational algebra the power of the first order predicate logic.

One could reasonably also allow extra columns on the left hand side, interpretting these as cartesian products extending the relation defined by the right hand side of the definition.

4.0 HIGHER ORDER RELATIONS

We now consider relations where the columns themselves may contain relations. Now in relational databases this has been depracated ever since the first of Codd's papers [CODD 70]. However the seeds of it have recurred in the need to produce totals and subtotals, for which special measures outside the framework of the theory have had to be taken. An early attempt to handle this, based on the PRTV experience, was made by Hitchcock [HITC 76].

Figure 7 shows an example, the cardinality function which counts the members of a set. The tabulation only considers a few examples, and the full relation must of course be represented intensionally. At the moment we will assume some built-in function outside our programming language, but in the next section will see how to define this function recursively.

FIGURE 7
Example of Higher-order relation, Cardinality

```
given
CARDINALITY (set:(...), size:integer)
= (          set: (...)          , size : integer)
 {({(a),(b),(c),(d),(e),(f),(g),(h)},     8),
  ({(0,0,01),(0,0,1,0)},                  2),
  ({(1,0.3),(2,56.2),(3,1.7),(4,13.8)},   4),
  ({(FRED),(JOE)}                         2),
                . . .                       }

define
SIZE_OF_SET(size:integer)
= CARDINALITY(set,size)
  *(set:(...)){({(0,0,0,1),(10)},
    ({(FORTRAN),(COBOL),(ALGOL),(PASCAL),(LISP)})}

SIZE_OF_SET is {(2),(5)}
```

To make this and other definitions of higher order relations acceptable, we will have to consider further the specification of "type" in relations. Up to this point we have assumed a base set of types like "real", "integer", "string", but now

must extend this to include relations themselves. We have already defined the type of a relation as the list of the types of its columns, but that is too restrictive for higher order relations like TOTAL shown in Figure 8 which must sum the values of one column but does not care about the other columns, if any. We will use a "..." notation to indicate optional columns, treating this informally while recognising that this should be properly defined, perhaps as a lattice of types. Figure 7 and 8 have used the notation we intend to employ. The syntax is:

```
fullrelationtype ::= relationtype
                   |   "(...,"     variable:type
                  ",...)"
                   | "(...)"
```

We will also allow columns to contain individual tuples, and denote their type in the same way as relations, but preceding the type description by a prime:

```
tupletype ::= "'" fullrelationtype
```

The syntax for type is then:

```
type ::= simpletype
       | fullrelationtype
       | tupletype
simpletype ::= identifier
```

We will pick out the components of a tuple using the usual dot notation:

```
typecomponent ::= tuplevariable "." columnvariable
```

FIGURE 8 - The Higher-order relation, TOTAL

```
given
TOTAL (relation:(...,NUMBER:real,...),total:real)

define
ORDER_SUMMARY(supplier:string,total_cost:real)
= TOTAL(orders:(...,Cost,...),total-cost)
 *(supplier:string,orders:(item:integer,cost:real)
  {(BLOGG_BROS,{(1,57.21),(32,1.06),(127,42.43)})}

ORDER_SUMMARY is {(BLOGG_BROS, 100.70)}
```

Examples involving tuples as components only appear in section 4, Figures 11 and 12.

We can write relations as column values within relation constants, as has been done in Figures 7 and 8. However we will need general mechanisms for converting given relations into "equivalent" higher order relations, and vice versa.

For this we will extend the relational-definition operation, so that the columns of a relation and its subrelation can be regrouped as shown in Figure 9. The intermediate "PURCHASE" relation is not necessary, but has been included for clarity. We will only use this operation in the form shown there, where a first order relation is converted to a second relation with only one column of relations. However a general operation could be defined, though it will not be done here, since it requires a very careful definition that would digress from our main argument. The operation of Figure 9 is the "GROUP BY" operation of SQL [ABCE 76] necessary for sub-totalling. Figure 9 shows this use.

FIGURE 9
Conversion from first-order to second-order

```
given
PURCHASE_DETAILS(supplier:string,item:integer,
                 number:integer,price:real)

define
PURCHASES(supplier:string,item:integer,cost:real)
= PURCHASE_DETAILS(supplier,item,number,price)
        *(cost = number * price)

PURCHASES_BY_SUPPLIER
(supplier:string,order:(item:integer,cost:real))
= PURCHASES(supplier,item,cost)

PURCHASES-SUMMARY (supplier:string,total_cost:real)
= PURCHASES_BY_SUPPLIER(supplier,orders(item,cost))
 *TOTAL(orders:(...,cost,...),total-cost)
```

This simple operation for converting between orders of functions provides a very powerful file conversion language, and could provide a theoretical basis for languages like EXPRESS [SHTG 77].

Higher order relations provide the ability to modify "programs" in the sense that by suitable relational expressions the transformations defined by a relation can be changed. Note that this is different from the kind of changes featured by LISP which changes program text. Of course here we could represent text by suitable relationships and do the sort of manipulations practiced in LISP, to write compilers and similar. However, that has not been my motivation, and my notation has not been aimed at that.

5.0 RECURSIVE DEFINITIONS

Recursive definition of relations, inspired by the methods of functional programming, follows very readily. Figure 10 shows that overworked example, the factorial function.

FIGURE 10
Recursive Relational Definition of Factorial

```
fact(n) = if n=0 then 1 else n*fact(n-1)

FACT(n:integer,r:integer)
 =  (n=0)*(r=1)
  + (n>0)*(n*x=r)*(y+1=n)*FACT(y,x)

Note:  y+1=n is used to compute y=n-1.
```

Recursive definitions will be most useful in the definitions of higher order relations, but before we can do that, we need to be able to recurse on the relations themselves, in the same way that LISP and its derivatives recurse on list structures. We could choose to treat relations as sequences and follow the practices of LISP, but relations are sets and no ordering may be natural.

What we need is some method of breaking a relation into parts, and then these parts into parts and so on recursively until atomic parts are encountered. We will see two alternative methods of approaching this.

A first method is the "choice" function of set theory. Given a set, it chooses an element and returns this together with the residue (the set less the chosen element). We also need a way of stopping the recursion, and a function for testing for the null set is assumed. This is the counterpart of the car, cdr and null functions of LISP. A device like this has been used by Hitchcock [HITC 74].

Figure 11 gives an example. The choice function applies to relations of arbitrary type. TOTAL is defined recursively to sum the values on the column "number" in the given relation. If the relation is empty the sum is zero, otherwise for a non-empty relation an element is chosen and the "number" column of that element is added to the sum of the residue.

FIGURE 11
The Choice Function and the definition of TOTAL

```
given
CHOICE(set:(...),element:'(...),residue:(...))
EMPTY(set:(...))

define
TOTAL(relation:(...,number:real,...),sum:real)
 = EMPTY(relation)*(sum = 0)
 +(-EMPTY(relation))*CHOICE(relation,elem,res)
   *TOTAL(res,sum_res)*(elem.number+sum_res=sum)
```

Note: we have used the notation "elem.number" to indicate the binding between the result of the CHOICE relation and the addition relation "(elem.number+sum_res=sum)".

The second method is a "split" function which devides the set into two arbitrary disjoint sets which are non-empty if that is possible. To stop the recursion a method of testing for a singleton is necessary. This approach is similar to Williams' tree function-composer [WILL 82].

An example is shown in Figure 12, where a second definition of TOTAL is given. For empty relations the result is zero (the long stop for empty arguments to TOTAL). For singletons the sum is the value of the NUMBER column. For other relations the result is the sum of the TOTALS for the two subsets selected by the SPLIT relation. Note that the SINGLETON relation has columns of the singleton relations and the elements contained in them. The NON-SINGLETON relation is defined as the relative complement of SINGLETON with the elements projected out.

FIGURE 12
The Split Function and another definition of TOTAL

```
given
SPLIT(set:(...),subset1:(...),subset2:(...))
SINGLETON(set:(...),element:'(...))

define
TOTAL(relation:(...,number:real,...),sum:real)
= SINGLETON (relation,element)*(element.number=sum)
+ NON_SINGLETON(relation)*SPLIT(relation,sub1,sub2)
  *TOTAL(sub1,sum1)*TOTAL(sub2,sum2)
  *(sum1 + sum2 = sum)
+ EMPTY(relation)*(sum=0)

where
NON_SINGLETON(rel:(...))= - SINGLETON(rel,elem)
```

We must be careful to distinguish recursive definitions, from equalities which define properties of the relation. For example, a relation could be defined as symmetric by the equation:
R(a:X,b:X) = R(b:X,a:X) This is a higher order relation, and fits into our theory, though the treatment of such cases is outside the scope of this paper.

6.0 A SIMPLE EXAMPLE

In this section a very simple example is chosen to illustrate the language developed in this paper. The problem domain chosen is scientific data analysis: commercial problems have been avoided because they typically involve update, and the handling of time and update is outside the scope of the study so far.

A survey is being made pigs being sold at the local meat market. It is required to find out the average weight of carcases, and the average price per kilogram. These figures should be broken down by producer and by purchaser.

An entity-relationship systems analysis [HOT 76, CHEN 76] shows three entities:
- PRODUCER
- CARCASE
- PURCHASER and one relationship:
- SALES

as shown in Figure 13. In these the columns "id" are the surrogates of [HOT 76].

We want to produce the reports:
- PRODUCER_REPORT
- PURCHASE_REPORT

with columns:
(name:string, average_weight:kilograms, weight_variance:kilograms, total_weight:kilograms, average_price:sterling_per_kilogram)

FIGURE 13 - Sales analysis of Pig Market

```
Entities are

PRODUCER(producer:id,name:string)
CARCASE(carcase:id,weight:kilograms,grade:grade)
PURCHASER(purchaser:id,name:string)

Relationships are

SALES(producer:id,carcase:id,purchaser:id,
                    price:sterling,date:date)

Utilities relations required are

TOTAL(relation:(...,number:real,...),total:real)
= EMPTY(relation)*(sum=0)
+ (-EMPTY(relation))*CHOICE(relation,elem,residue)
  *TOTAL(residue,sum_residue)
  *(elem.number+sum_residue=total)

TOTAL_SQUARED
(relation:( ...,number:real,...),total_sq:real)
= EMPTY(relation)*(total_sq=0)
+ (-EMPTY (relation))*CHOICE(relation,elem,residue)
  *TOTAL_SQUARED(residue,tsq_res)
  *(elem.number**2 + tsq_res = total_sq)

COUNT(relation:(...),count:integer)
= EMPTY (relation)*(count=0)
+ (-EMPTY(relation))*CHOICE(relation,el,res)
  *COUNT(res,cres)*(cres + 1 = count)
```

```
Definitions of Reports are

PRODUCER_REPORT(producer_name:string,
                average_weight:kilograms,
                weight_variance:kilograms,
                total_weight:kilograms,
                average_price:sterling_per_kilogram)

= PRODUCER_GROUP(producer_name,prods:(weight,price))
 *TOTAL(prods:(...,weight,...),total_weight)
 *TOTAL(prods:(...,price,...),total_price)
 *TOTAL_SQUARED(prods:(...,weight,...),tot_wt_sqd)
 *COUNT (prods:(...),num)
 *(average_price=total_price/total_weight)
 *(average_weight=total_weight/num)
 *square_root(var_sq,weight_variance)
 *(var_sq = tot_wt_sqd - average_weight**2)/(num-1)

 where
   PRODUCER GROUP(producer name:string,
           prods:(weight:kilograms,price:sterling))
   = PRODUCER(prod,producer_name)*
    *SALES(prod,carc,purch,price,date)
    *CARCASE(carc,weight,grade)

PURCHASER_REPORT(purchaser_name:string,...)
     as for PRODUCER_REPORT
```

To produce these we need to gather together all the sales of a particular producer or purchaser, and count these and sum these over carcase weight, price, and weight-squared, and from these compute the statistics required. The higher-order relations required, and their use, are shown in Figure 13.

Note that further facilities would be required to control the format of the reports arranging presentation sequence, layout, etc. Note also that efficiency has been assumed to be left to the implementation, so that, for example, the use of the functions TOTAL, TOTAL-SQUARED and COUNT could be optimised to require only one traversal of the data.

7.0 CONCLUSIONS

What I have done is develop a rapprochement between Relational Algebras, Logic Programming, and Functional Programming.

Each of these has its own implementation technology. Hopefully through this rapprochement these technologies will become integrated to lead to really effective implementations of non-procedural languages manipulating large volumes of data.

The correspondence between relational databases and logic has been known for a long time and led to fruitful results [e.g. GM 80]. The correspondence of these two with functional programming languages is the interesting one. The seeds have been there in the database literature for some time particularly in the work of Buneman [BF 79, BFN 82] and Shipman [SHIP 81], though neither of the approaches allow general higher order operations. The development of the correspondence with functional programming has enriched relational algebras, and in particular moved away from a lot of the constraints imposed by first normal form.

For logic programming the correspondence with a higher-order relational algebra may suggest a way of moving logic programming into higher order logics, as desired by Kowalski [KOWA 83].

A recent paper by MacLennan [MACL 83] takes a relational view of functional programming. His motivation is very different from mine, wishing to advance functional programming through the less restrictive facilities offered by binary relations. The final facilities are, nevertheless, appealingly similar and his paper makes an excellent complement to my discussions.

What I have not entended to do is to develop yet another database system. The notations developed here are very compact and potentially difficult to use. At best they could serve as an abstract syntax, with the concrete syntax containing many more operations to enable ease of use and, therefore, accurracy of use.

While elaboration of a concrete syntax on the relational model may appear attractive with its tabular view of data, I believe the way forward will be through functional programming using a functional view of data. This needs development. Perhaps a dual view within a single system could emerge.

One particular advantage of the relational view was its neutrality on "direction". Relations can be used in any direction; functions engender a very strong direction from domain to range which can help the user, but, perhaps, in the end constrain him. In implementation, too, the neutrality of relations can be an advantage.

The next step in this line of development of databases must be the incorporation of update and time. The usual approach of before and after states inherent in most formal approaches to computing [e.g. JONE 80] is not judged adequate, and something more along the life history approach of Michael Jackson [JACK 83] is expected to be the fruitful approach.

8.0 REFERENCES

AB 84 Abitebout S. and Bidoit N., "Non First Normal Form Relations to Represent Hierarchically Organised Data", Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 1984.

ABCE 76 Astrahan M.M., Blasgen M.W., Chamberlin D.D., Eswaren K.P., Gray J.N., Griffiths P.P., King W.F., Lorie R.A., McJones P.R., Mehl J.W., Putzolu G.R., Traiger I.L., Wade B.W., and Watson V. "System R: Relational Approach to Database Management" ACM Transactions on Database Systems, Vol. 1, no. 2, June 1976 page 97-137.

BACK 78 Backus J.W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." CACM, August 1978.

BF 79 Buneman O.P., Frankel R.E. "FQL - A functional query language ", In Proc. ACM Sigmod, Int. Conf. Management of Data, Boston May 30th - June 1, 1979, ACM, New York 1979. Pages 52-57.

BFN 82 Buneman P., Frankel R.E., Nikhil R. "An Implementation Technique for Database Query Languages" ACM Trans. on Database Systems, Vol. 7 No.2 June 1982 pages 164-186.

CHEN 76 Chen P.P-S "The Entity-relationship model: Towards a unified view of data", ACM Trans Database Systems. Vol. 1, No 1, March 1976, page 9-36.

CM 81 Clocksin W.F. and Mellish C.S. "Programming in Prolog" Springer 1981.

CODD 70 Codd E.F. "A Relational Model of Data for Large Shared Data Banks" CACM Vol. 13, No.6, June 1970. Pages 377-387.

CODD 72 Codd E.F. "Relational Completeness of Data Base sublanguages" in Data Base Systems (Ed. R. Rustin) Prentice Hall, 1972. Pages 65-98.

DHT 82 Darlington J., Henderson P., Turner D.A., "Functional Programming and its Applications" Cambridge University Press, 1982.

GM 78 Gallaire H. and Minker J. "Logic and Databases" Plenum Press 1978.

GRAD 83 Gradwell D.J.L. "ICL's Data Dictionary System" in Database 83 Conference: BCS

HEND 80 Henderson P. "Functional Programming, Application and Implementation" Prentice-Hall 1980.

HHT 75 Hall P.A.V., Hitchcock P.J., and Todd S.J.P. "An Algebra of Relations for Machine Computation". Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, Jan. 1975. Pages 225-232.

HITC 74 Hitchcock P. "Fundamental Operations on Relations" IBM UK Scientific Centre, Report UKSC-0051 May 1974.

HITC 76 Hitchcock P. "User Extensions to the Peterlee Relational Test Vehicle", 2nd VLDB Conference, Brussels 1976.

HOT 76 Hall P.A.V., Owlett J. and Todd S.J.P. "Relations and Entities" in Modelling in DataBase Management Systems, Editor G.M. Nijssen, IFIP/North Holland, 1976. Pages 201-220.

JACK 83 Jackson M.A. "System Development" Prentice Hall 1983.

JONE 80 Jones C.B. "Software Development. A Rigorous Approach". Prentice Hall 1980.

KOWA 74 Kowalski R.A. "Predicate logic as a programming language" in Information Processing 74, North Holland 1974. Pages 569-574.

KOWA 83 Kowalski R.A. "Logic Programming" Invited Paper for IFIP 83, to appear.

LP 83 Lorie R., and Plouffe W. "Complex Objects and Their Use in Design Transactions", in Engineering Applications, Database Week, ACM SIGMOD Annual Meeting, May 1983.

MACL 83 MacLennan B.J. "Overview of Relational Programming" SIGPLAN Notices, Vol. 18 No. 3 March 1983. Pages 36-45.

MCCR 80 McCracken D.D. "A Guide to NOMAD for Applications Development" Addison-Wesley 1980.

SHIP 81 Shipman P.W. "The Functional Data Model and the Data Language DAPLEX" ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981. Pages 140-173.

SHTG 77 Shu N.C., Housel B.C., Taylor R.W., Ghosh S.P., and Lum V.Y. "EXPRESS: a Data Extraction, Processing, and REStructuring Systems" ACM Trans. on Database Systems, Vol.2, No.2 June 1977 Pages 136-171.

TODD 76 Todd S.J.P. "Peterlee Relation Test Vehicle: a system overview" IBM Systems Journal, Vol. 15, No.3 August 1976.

WILL 83 Williams J.H. "Note on the FP style of Functional Programming" page 73-102 in DHT 82.