The Specification of a Relational Database (PRECI) as an abstract data type and its realisation in HOPE

E. WONG and W. B. SAMSON*
Dundee College of Technology, Bell Street, Dundee DD1 1HG, Scotland

Abstract data types are used to describe the PRECI canonical database and its algebraic language which is based on relational algebra. The HOPE language provides a vehicle by which these abstractions may be implemented directly. A HOPE implementation is achieved and provides a model against which implementations of PRECI may be verified for completeness and correctness.

Received March 1984

1. INTRODUCTION

In order to achieve completeness and correctness in the implementation of any system, it is desirable to specify formally the properties of data types at an abstract level, independent of any particular implementation.[4] The resulting formal specifications may be used to verify the correctness of an implementation and as a guide to help programmers minimise the differences between various implementations.

Several authors[1, 8, 10–18] have used the algebraic specification techniques for abstract data type definition[13] to describe aspects of database.

A formal specification of the PRECI[9] data base and its algebraic language, PAL are described in this paper. The HOPE language[2, 6] allows a new data type to be defined in a way analogous to that used by Guttag & Horning[13] to describe abstract data types in terms of constructors, and functions defined for those constructors. This allows an essentially algebraic specification to become an implementation which is functionally defined, and for which formal correctness proofs are trivial compared with those needed for algorithms written in procedural languages.

2. PRECI

PRECI (Prototype of a RElation Canonical Interface)[9] is a generalised database system based on a canonical data model, i.e. one which is potentially capable of supporting user views of all major models through local schemas and appropriate data manipulation languages. The version under development has implemented the CODASYL and relational subschema facilities. A relational algebra to be used for Data Manipulation commands within Fortran programs has also been provided.

The design of PRECI is partly dictated by the need for a generalised system. It has been implemented at Aberdeen University primarily as a test vehicle for research in the various aspects of databases, with a modular design and flexible approach so that future changes will be straightforward. Run-time efficiency, minimal memory usage, optimisation facility, data independence and ease in restructuring and reorganisation are the major features of the model. Other implementations of PRECI are now envisaged, and the work described in this paper will assist programmers to produce consistent implementations.

This paper describes an attempt to isolate some features of PRECI which are implementation-independent, and rigorously defines the behaviour of these aspects of PRECI from the user's viewpoint.

* To whom correspondence should be addressed.

3. HOPE

HOPE is an applicative language first developed and implemented at Edinburgh University.[2, 6] It is a simple language which encourages clarity and manipulability of programs, and provides a means of testing ideas in programming methodology. The language allows maximum use of user-defined types and the techniques of data abstraction, which makes it ideal for realising the formal specifications of the abstract data types derived for the PRECI database. The version of HOPE which is used here was developed by Wu and Darlington at Imperial College London. It is a portable version of the language, implemented in PASCAL. The authors made the small modifications necessary for it to run on a DECsystem-20. This version of HOPE, which is still under development, has most significant features of the full language, apart from modularity.

4. PAL

The PRECI Algebraic Language, PAL, is a language based on the relational algebra,[3] and contains the following functions which return a **relation** as a result. The definitions of PAL functions given below are not the only possible ones, but are typical of the accepted definitions of relational algebra operations. These are the definitions used by the implementors of PRECI. It is beyond the scope of this paper to perform a critical comparison of alternative definitions.

4.1 Delete

The function, 'delete', is used to delete any required **tuple** from a given **relation**.

4.2 Difference

The 'difference' between two union-compatible **relations** A and B is the new **relation** with all its **tuples** belonging to A but not to B.

4.3 Division

The 'division' function divides a dividend **relation** A of degree $m+n$ (i.e. A has $m+n$ attributes) by a divisor **relation** B of degree n, and produces a result **relation** of degree m. The $(m+i)$th attribute of A and the ith attribute of B (i in range 1 to n) must be defined on the same domain. A is considered as a set of pairs of values (x, y), where x denotes the first m attributes and y denotes the last n attributes of A; B is considered as a set of single values y. The result of A 'divided by' B is the **relation** with the set of values x such that the pair (x, y) appears in A for all values y appearing in B.

4.4 Insert

The function, 'insert', is used for the addition of a suitable **tuple** to a given **relation**.

4.5 Intersection

The 'intersection' of two union-compatible **relations** A and B is the new **relation** with all its **tuples** belonging to both A and B.

4.6 Join

'Join' is a function of two **relations** A and B where each relation has an attribute defined over the same domain, they are then 'joined' over these two attributes. The result is a new **relation** in which each tuple is formed by concatenating two tuples, one from each of the original relations. The most common form of 'join' is the 'equijoin' where the two tuples have the same value in the two joining attributes. The equijoin with one of the two identical attributes eliminated is the 'natural join'.

4.7 Projection

'Projection' forms a vertical subset of a **relation** by extracting specified attributes and removing any redundant duplicate tuples in the result relation.

4.8 Selection

This function returns a new **relation** by taking a horizontal subset of a relation, i.e. all the tuples of the result relation satisfy some condition.

4.9 Union

The 'union' of two union-compatible **relations** A and B is the new relation with all its tuples belonging to either A or B or both.

4.10 Union-Compatibility

This is not an explicit PAL function; however, it is an essential condition which must be satisfied for the functions, 'difference', 'intersection' and 'union'. Date[7] gives the following definition.

Two **relations** of degree n, say $R(A_1,\ldots,A_n)$ and $S(B_1,\ldots,B_n)$ are said to be union-compatible with respect to a Correspondence C, if and only if C is a set of exactly n ordered pairs of **attributes** (A_i, B_j) $(i,j = 1,\ldots,n)$, and the following three conditions hold:

(a) each **attribute** for R is some A_i $(i = 1,\ldots,n)$;

(b) each **attribute** for S is some $B_j (j = 1,\ldots,n)$;

(c) within each pair (A_i, B_j) of the set, the **attributes** designated by A_i and B_j have the same corresponding domain;

It should be noticed that union-compatibility applies to the **schemes** of relations and is independent of the content of the tuples of the relations, and, indeed, the names of the attributes. Nevertheless, when we come to define the function 'union-comp', it will, for the sake of clarity, be applied to relations as arguments.

5. THE ALGEBRAIC SPECIFICATION OF DATA TYPES

The **abstract** (or **algebraic**) specification of any system consists of two parts:

Syntax – the operations of the system are specified indicating the number of arguments, the argument types and the result type.

Semantics – algebraic equations (axioms) are given that relate the values created by the operations.

The **algebraic** specification of data types, described by Guttag & Horning,[13] involves the choice of a number of **constructor operators** with which any element of that data type may be defined. Each function on the data type is then defined for each of the constructor operators for that type, hence is completely defined.

The types **num**, **char** and **truval**(boolean) are supplied as part of the HOPE environment. In addition, the following abstract data types required to be defined in order to realise PAL functions in HOPE: **value, identifier, attype, attribute, tuple, scheme, o_scheme, relation, schema, database**.

The hierarchy of abstract data types used to define the **PAL** functions is shown in Fig. 1.

Three fundamental data types, **num**(numerals), **char**(characters) and **truval**(boolean) are defined within the HOPE environment, they are used to construct the next level of abstract data types. On the lowest level in the hierarchy, there is type **identifier** which is used in the construction of an **attribute** and represents its name, and type **attype** which is used in the construction of an **attribute** and represents, in a limited way, its domain.[7] A **tuple** is constructed from **attributes** and corresponding **values**. A **scheme** which describes a **relation** is constructed from **attributes**. An **o_scheme** is an ordered scheme, i.e. a **scheme** which is constructed with a list of **attributes**, hence the order is taken into account. A **relation** is constructed from a **scheme** and a set of matching **tuples.** A **schema** is constructed from the **schemes** which describe the set of **relations** which make up the **database**. A **database** is constructed from a **schema** and a set of matching **relations. Value** is an occurrence which matches an **attype.** There is at present no notion of a *key* in this model. It will be seen that the functions defined do not require any key.

It is necessary to define data types at all of these levels in order that a HOPE realisation of PAL functions can be achieved.

5.1 Data type attype

The data type **attype** ('attribute type') is defined in a HOPE data declaration as follows:

```
data attype == chr(num)
            ++ dig(num);
```

'**data**' is a reserved word. '==' is pronounced 'is defined as' and '++' is pronounced 'or'. The functions **chr(num)** and **sig(num)** are **constructors** of data type **attype** which defines the type of an **attribute,** which is either a number (num) of characters (chr(num)), or a number (num) of digits (dig(num)). These represent the data types that are currently available on PRECI. It is possible, however, to generalise **attype** for any kind of domain by declaring it to be a type variable, using the polymorphic facility provided by the Hope language.

5.2 Data type identifier

The data type **identifier** is defined in HOPE as:

```
data identifier == lst(list(char)); !CONSTRUCTOR;
```

Identifier is the name of an **attribute** which is made up of a list of characters. Here, 'lst' is a programmer-defined constructor, with argument 'list(char)' – a predefined data type for a list of characters.

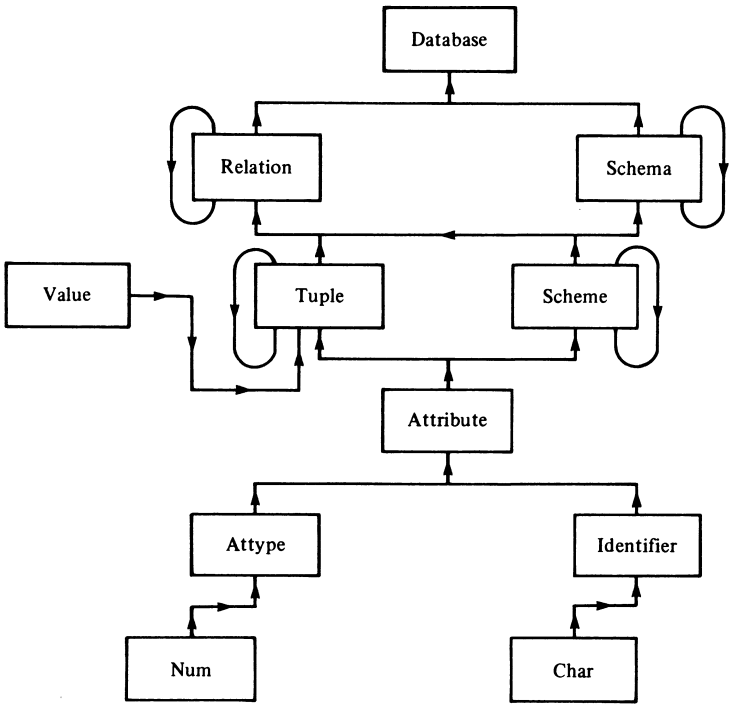

Figure 1. The hierarchy of abstract data types. The arrows indicate the use of lower-level types in the definition of higher-level types. Some types are defined recursively (type, scheme, relation and scheme).

There is one function on **identifier,** 'getlst', which gives the list of characters of the **identifier**.

```
dec getlst: identifier → list(char);    !SYNTAX;
--- getlst(lst(l)) ⇐ 1;                 !SEMANTICS;
```

The syntax line should be read as follows. The word **dec** is a reserved word of HOPE indicating a function definition. 'getlst' is the name introduced by the programmer for the function. The symbol ':' is read as 'takes a'. This is followed by the argument(s) of the function; in this case there is one argument, of type 'identifier'. Following the argument the symbol '→' which reads 'yields' and the type of the function, which in this case is 'list(char)', i.e. a list of characters, which is a predefined type.

The semantics line is a **recursion equation** which specifies the result in terms of the argument(s). In this case the programmer-defined constructor function 'lst(l)' is the argument, 'l' being a list of characters.

5.3 Data type attribute

The data type **attribute** is defined in HOPE as follows:

```
data attribute = = acons(identifier × attype);
```

An **attribute** is constructed from two other data types, **identifier** which is the name of the **attribute** and **attype** which gives its type. 'acons' is a programmer-defined constructor function.

Two examples of functions on type **attribute** are:

```
dec name : attribute → list(char);
--- name(acons(i,d)) ⇐ getlst(i);

dec getd : attribute → attype;
--- getd(acons(i, d)) ⇐ d;
```

The 'name' function returns a list of characters which is the name of the **attribute**, while 'getd' returns the **attype** of the **attribute**.

Two **attributes** are said to be compatible if they are drawn from the same **attype**. N.B. their **identifiers** are not necessarily the same. Hence the two following functions are defined:

```
dec atcomp : attribute × attribute → truval;
---atcomp(acons(i,d),a) ⇐ getd(a) = d;
                        !result is true
                        !if the attributes
                        !have the same attype.
dec compair : attribute × attype → truval;
---compair(acons(i,dl),d2) ← d1 = d2;
                        !result is true
                        !if the attribute is
                        !of the given type.
```

5.4. Data type scheme

In HOPE, the data type **scheme** is defined as follows:

```
data scheme = = senull
              + +secons(scheme × attribute);
```

A **scheme** describes a **relation**. From the scheme, one can deduce the degree of the relation, i.e. the number of **attributes** it has; and what the attributes are.

The following function on **scheme** may be used to determine whether a particular attribute belongs to a given scheme:

```
dec attof : scheme × attribute → truval;
---  attof(senull,a) ⇐ false;
---  attof(secons(s, a1),a2) ⇐ if a1 = a2
                               then true
                               else attof(s,a2);
```

5.5. Data type o_scheme

The data type **o_scheme** is defined in HOPE as:

```
data o_scheme = = ocons(list(attribute));
```

An o_scheme, like a scheme, is made up of attributes. However, the attributes are in an ordered list, hence the order is fixed and provides a notation for correspondence between schemes, as described above in Section 4.10.

5.6. Data type tuple

The data type **tuple** is defined in HOPE as:

```
data tuple = =      tempty
                    + +tcons(tuple × attribute × value);
```

A **tuple** can be empty, or it can be constructed from **attributes**, each with its corresponding **value**. The need for an 'empty' tuple will become clearer when the 'project' operation is defined.

'Getval' is defined to facilitate the retrieval of the **value** of a particular **attribute** in a given **tuple.** 'tscheme' is for getting the **scheme** which the **tuple** is matched into.

```
dec getval : tuple × attribute → value;
---getval(tempty,a) ⇐ undefine;
---getval(tcons(t,a,v),a1) ⇐ if a = a1
                             then v
                             else getval(t,a1);
dec tscheme : tuple → scheme;
---tscheme(tempty) ⇐ senull;
---tscheme(tcons(tempty,a,v)) ⇐ secons(senull,a);
---tscheme(tcons(t,a,v)) ⇐ secons(tscheme(t),a);
```

Note that both of the above functions are recursively defined. Since a tuple must belong to a certain relation, it must match the scheme of that relation. Therefore the function shown below is defined to test the compatibility of a **tuple** and a **scheme**.

```
dec compatt : scheme × tuple → truval;
---compatt(senull,t) ⇐ true;
---compatt(s,tempty) ⇐ true;
---compatt(s,tcons(t,a,v)) ⇐      if attof(s,a)
                                  then compatt(s,t)
                                  else false;
```

5.7 Data type relation

In HOPE, the data type **relation** is defined as follows:

```
data relation = =  rnull
                  + + rcons(relation × scheme × tuple);
```

A **relation** can either be null, or it can be constructed from a **scheme** which describes it and a set of matching **tuples**.

Two functions are defined to facilitate the manipulation of relations. Getsche returns the scheme of the relation, tuple_of determines whether a particular tuple belongs to the given relation.

```
dec getsche : relation → scheme;
---getsche(rnull) ⇐ senull;
---getsche(rcons(r,s,t)) ⇐ s;

dec tuple_of : relation × tuple → truval;
---tuple_of(rnull,t) ⇐ false;
---tuple_of(r,tempty) ⇐ true;
---tuple_of(rcons(r,s,t1),t2) ⇐
                        if tequal(t1,t2)
                        then true
                        else if compatt(s,t2)
                        then tuple_of(r,t2)
                        else false;
```

Where 'tequal' tests for the equality of the two **tuples**. Two **tuples** are equal if for each **attribute** and its associated **value** in one **tuple**, there is a corresponding **attribute** with the same **value** in the other **tuple**. There is no specific limitation on ordering.

5.8 Functions of PAL

Delete

The Delete function is called 'tdelete' and is defined in HOPE as follows:

```
dec tdelete : relation × tuple → relation;
---tdelete(rnull,t) ⇐ rnull;
---tdelete(rcons(r,s,t1),t2) ⇐
                        if tequal(t1,t2)
                        then r
                        else rcons(tdelete(r,t2),s,t1);
```

Difference

The function Difference is called 'differ' and is defined as follows:

```
dec differ :    relation × o_scheme ×
                relation × o_scheme →
                relation;

---differ(r1,o1,rnull,o2) ⇐ orcons(r1,o1);
---differ(rnull,o1,r2,o2) ⇐ rnull;
---differ(rcons(r1,s1,t1),o1,r2,o2) ⇐
                        if not (union_comp(
                        rcons(r1,s1,t1),o1,r2,o2))
                        then undefine
                        else if tuple_of(r2,t1)
                        then tdelete(differ(
                        r1,o1,r2,o2),t1)
                        else rcons(differ(
                        r1,o1,r2,o2),
                        socons(o1,o2),t1);
```

The two '**o_scheme**' arguments provide a correspondence between the **attributes** to be matched up in the Difference. The scheme of the result is given by 'socons(o1,o2)', which returns a set of attributes whose identifiers are constructed by concatenating the identifiers of the corresponding attributes in the two relations.

Division

The function Division is called 'division' and is defined in HOPE as:

```
dec division :  relation × o_scheme ×
                relation × o_scheme →
                relation;
---division(r,o1,rnull,o2) ⇐ undefine;
---division(rnull,o1,r,o2) ⇐ undefine;
---division(rcons(r1,s1,t1),o1,r2,o2) ⇐
                        divide(r,r2,o2,
                        rcons(r1,s1,t1),o1)
                        where r = = project(rcons(
                        r1,s1,t1),
                        sominus(s1,o1));
```

The parameters of this function are, respectively, the dividend relation, the ordered subscheme for the attributes which correspond to those of the divisor relation, the divisor relation and its ordered scheme. 'sominus' is a function to return the scheme of the attributes which are not directly involved in the division. 'divide' performs the main part of the division operation and is defined as follows:

```
dec divide : relation × relation × o_scheme
          × relation × o_scheme → relation;

---divide(rnull,r1,o1,r2,o2) ⇐ rnull;
---divide(r1,rnull,o1,r2,o2) ⇐ undefine;
---divide(r1,r2,o2,rnull,o3) ⇐ undefine;
---divide(rcons(r,s,t),r2,o2,r1,o1) ⇐
                        if rsub(r1,o1,trjoin(t,r2),o2) then
                        rcons(divide(r,r2,o2,r1,o1),s,t)
                        else divide(r,r2,o2,r1,o1);
```

Where 'rsub' checks whether one **relation** is a sub-relation of another **relation**.

Insert

Insert is called 'tinsert' and is defined as:

```
dec tinsert : relation × tuple → relation;
---tinsert(r,tempty) ⇐ r;
---tinsert(rnull,t) ⇐ rcons(rnull,tscheme(t),t);
---tinsert(rcons(r1,s1,t1),t2) ⇐
                        if not(sequal(s1,tscheme(t2)))
                        then undefine
                        else if
                        tuple_of(r1,t2)
```

```
                then rcons(r1,s1,t1)
                else rcons(rcons(r1,s1,t1),s1,t2);
```

where 'tscheme' is the function which returns the **scheme** of a given **tuple**. Two **schemes** are equal if for each **attribute** in one scheme there is a corresponding one in the other scheme; again, ordering is not important. 'sequal' is the function which tests for this condition.

Intersection

This function is called 'inter' and defined in HOPE as:

```
dec inter :     relation × o_scheme ×
                relation × o_scheme →
                relation

---inter(r1,o1,rnull,o2) ⇐ rnull;
---inter(rnull,o1,r2,o2) ⇐ rnull;
---inter(rcons(r1,s1,t1),o1,r2,o2) ⇐
                        if not(union_comp(
                        rcons(r1,s1,t1),o1,r2,o2))
                        then undefine
                        else if tuple_of(r2,t1)
                        then rcons(inter(
                        r1,o1,r2,o2),
                        socons(o1,o2),t1)
                        else inter(r1,o1,r2,o2);
```

The first two parts of the function definition show that the intersection of the empty **relation** with any **relation** gives the empty **relation**. The third part defines the intersection of two non-empty relations. The two relations have to be checked for union-compatibility. The result relation is defined as tuples which belong to both relations. The recursive property of the function ensures that all tuples matching the criteria are put into the result relation. The two **'o_scheme'** arguments provide a correspondence between the attributes to be matched up in the Intersection. The scheme of the result relation is provided by the 'socons' function (– see definition of 'differ' above).

Join

The Join function is called 'join' and is defined as follows:

```
dec join : relation × relation × (tuple × tuple → truval)
      → relation;
!Maps two relations and a predicate on a
!pair of tuples into a relation.
---join(r,rnull,f) ⇐ rnull;
---join(rnull,r,f) ⇐ rnull;
---join(rcons(r1,s1,t1),r2,f) ⇐
                        runion(join(r1,r2,f),
                        condjoin(t1,r2,f));
```

where f represents a condition on two **tuples**, one from each of the two **relations** participating in the Join, and returns a boolean result. The function 'runion' is a simple union of two **relations**. 'Condjoin' is the function defined to check each **tuple** against the condition of the Join.

```
dec condjoin : tuple × relation × (tuple × tuple → truval)
                → relation;
---condjoin(t,rnull,f) ⇐ rnull;
---condjoin(tempty,r,f) ⇐ rnull;
---condjoin(t,rcons(r,s,t1),f) ⇐
                        if f(t,t1)
                        then rcons(condjoin(t,r,f),
                            sjoin(tscheme(t),s),
                            tcat(t,t1))
                        else condjoin(t,r,f);
```

where 'sjoin' is the join of two **schemes** and tcat' is the concatenation of two **tuples.** These two functions are defined as follows:

```
dec sjoin : scheme × scheme → scheme;
---sjoin(senull,s) ⇐ s;
---sjoin(s,senull) ⇐ s;
---sjoin(secons(s1,a),s2) ⇐ secons(sjoin(s1,s2),a);

dec tcat : tuple × tuple → tuple;

---tcat(tempty,t) ⇐ t;
---tcat(t,tempty) ⇐ t;
---tcat(tcons(t1,a,v),t2) ⇐ tcons(tcat(t1,t2),a,v);
```

To call 'join', it is convenient to set up a function specifying the required condition, using the internal mapping facility in HOPE. 'Join' is then called. The following is an example of an Equijoin performed over the third attribute of one **relation** with the fourth **attribute** of another **relation**.

```
dec rjoin1 : relation × relation → relation;
---rjoin(r1,r2) ⇐ join(r1,r2,(lambda(t1,t2)
        ⇒ (getval(t1,acons(1st('scty'),chr(10)))
            = getval(t2,acons(1st('pcty'),chr(10))))));
```

Projection

This function is called 'project' and is defined in HOPE as:

```
dec project : relation × scheme → relation;
---project(rnull,s) ⇐ rnull;
---project(r,senull) ⇐ r;
---project(rcons(r,s,t),s1) ⇐
                        if not(tuple_of(project(r,s1),
                            tproj(t,s1)))
                        then rcons(project(r,s1),s1,
                            tproj(t,s1))
                        else project(r,s1);
```

where s1 is the **scheme** of the result relation. 'tproj' is the function defined to pick out the required attributes from each tuple of the relation to form a new tuple to put into the resulting relation. In HOPE, it is defined as:

```
dec tproj : tuple × scheme → tuple;
---tproj(tempty,s) ⇐ tempty;
---tproj(t,senull) ⇐ tempty;
---tproj(t,secons(s,a)) ⇐
                        if atoftp(t,a)
                        then tcons(tproj(t,s),a,
                        getval(t,a))
                        else tproj(t,s);
```

where 'atoftp' tests whether an attribute belongs to a tuple.

Selection

The function Selection is called 'select' and is defined as follows:

```
dec select : relation × (tuple → truval) → relation;
---select(rnull,f) ⇐ rnull;
---select(rcons(rnull,s,t),f) ⇐
```

```
                    if f(t)
                    then rcons(rnull,s,t)
                    else rnull;
```

This function picks out the **tuples** in a **relation** which satisfies certain conditions and forms a new **relation**. To call 'select', it is convenient to set up a function for the condition, using the internal mapping facility of HOPE as in 'join' above. The condition in the following is that the value of the particular attribute in a relation is greater than 200.

```
dec select1 : relation → relation;
---select1(r) ⇐ select(r, (lambda t
                    ⇒ (getval(t,acons(1st("quantity"),
                        dig(3))) > 200)));
```

Union
The Union is called 'union' and is defined in HOPE as follows:

```
dec union : relation × o_scheme ×
            relation × o_scheme →
            relation;


---union(rnull,o1,r2,o2) ⇐ orcons(r2,o2);
---union(r1,o1,rnull,o2) ⇐ orcons(r1,o1);
---union(rcons(r1,s1,t1),o1,r2,o2) ⇐
                    if not (union_comp(
                    rcons(r1,s1,t1),o1,r2,o2))
                    then undefine
                    else if not(tuple_of(
                    union(r1,o1,r2,o2),t1))
                    then rcons(union(r1,o1,
                    r2,o2),
                    socons(o1,o2),t1)
                    else union(r1,o1,r2,o2);
```

Here, as in Intersection and Difference, the '**o_scheme**' arguments provide a correspondence between the attributes to be matched.

Union-Compatibility
The function to check whether two relations are Union-Compatible is called 'union_comp' and is defined in HOPE as follows:

```
dec union_comp: relation × o_scheme ×
                    relation × o_scheme → truval;
---union_comp(rnull,o1,r2,o2) ⇐ undefine;
---union_comp(r1,o1,rnull,o2) ⇐ undefine;
---union_comp(rcons(r1,s1,t1),o1,r2,o2) ⇐
                    if not (oscom(s1,o1)) or
                    not(oscom(
                    getsche(r2),o2)))
                    then false
                    else if
                    not(oequal(o1,o2))
                    then false
                    else true;
```

Oscom checks whether the **attributes** in an **o_scheme** are same as those in a **scheme**, not necessarily in the same order. Oequal checks for the equality of the lists of attypes from two o_schemes.

Data type SCHEMA
A **schema** describes a **database** and is defined in HOPE as follows:

```
data schema == snull
            ++ scons(schema × scheme);
```

A **schema** can either be null, or it is constructed from the various **schemes** of the **relations** which belong to a given **database**.

Data type DATABASE
The type **database** is called 'dbase' and is defined in HOPE as:

```
data dbase == dbempty
           ++ dbcons(dbase × schema × relation);
```

A **database** can either be empty, or it can be constructed from a set of **relations**.

Example
The following is an example of three relations taken from C. J. Date's text book.[7] They are constructed in HOPE using the function 'tinsert'. A function 'prrel' is defined to print out the relations in their familiar tabular form.

The three relations are 'supplier', 'part' and 'shipment', and they are displayed as follows:

```
> : prrel(supplier(rnull));
> : [  'scty','sts','snam','sno',
       'paris',30,'blake','s3',
       'paris',10,'jones','s2',
       'london',20,'smith','s1'] : list(TV000)

> : prrel(part(rnull));
> : [  'pcty','wt','color','pnam','pno',
       'london',14,'red','screw','p4',
       'paris',17,'blue','screw','p3'
       'rome',17,'green','bolt','p2'
       'london',12,'red','nut','p1'] : list(TV000)

> : prrel(shipment(rnull));
> : [  'quantity','pno','sno',
       200,'p2','s3',
       400,'p2','s2',
       300,'p1','s2',
       400,'p3','s1',
       200,'p2','s1',
       300,'p1','s1'] : list(TV000)
```

The results of three functions 'join', 'select' and 'project' are shown below. As explained before, for convenience, separate functions 'rjoin1' and 'select2' are set up to call 'join' and 'select' with specific conditions.

This first example shows the result of an 'equijoin' performed over city of the supplier relation and that of the part relation:

```
dec rjoin1   !join relations supplier and part over city
   : (relation × relation) → relation;
---rjoin1(r1,r2) ⇐ join(r1,r2,(lambda(t1,t2)
                    ⇒ (getval(t1,acons(1st('scty'),
                        chr(10)))
                      = getval(t2,acons(1st
                        ('pcty'),chr(10))))));
```

```
> : prrel(jroin1(supplier(rnull),part(rnull)));
> : ['scty','sts','snam','sno','pcty','wgt','color'
    'pnam','pno',
'london',20,'smith','s1','london',14,'red',
    'screw','p4',
'london',20,'smith','s1','london',12,'red',
    'nut','p1',
'paris',10,'jones','s2','paris',17,'blue','screw',
    'p3',
'paris',30,'blake','s3','paris',17,'blue','screw',
    'p3'] : list( TV000)
```

The second example is to select those tuples from the shipment **relation** with the value of quantity greater than 200.

```
dec select2 : relation → relation;
---select2(r)
    ⇐ select(r,(lambda t ⇒ (getval(t,
                    acons(1st('quantity'),
                    dig(3))) > 200)));

> : prrel(select2(shipment(rnull)));
> : [ 'quantity','pno','sno',
    400,'p2','s2',
    300,'p1','s2',
    400,'p3','s1',
    300,'p1','s1'] : list( TV000)
```

This third example shows the result relation when a Projection is performed over the three attributes of the supplier relation.

```
> : prrel(project(supplier(rnull),new_scheme(senull)));
> : [ 'sts','snam','sno',
    30,'blake','s3',
    10,'jones','s2',
    20,'smith','s1'] : list(TV000)
```

where new_scheme is the scheme of the result relation.

8. DISCUSSION

The definition of the PRECI database using abstract data types benefits the design team in the following ways.

It provides an unambiguous specification for implementors to follow. Implementations may be tested against the HOPE model.

Test cases for all functions, including at least some of the extreme values and all special cases, are clearly shown in the specification.

Any inconsistency, redundancy or non-orthogonal property of functions in the design is highlighted.

Ideas such as named and unnamed relations as well as virtual relations[7] are eliminated from the specification, since there is no notion of temporal logic in this functional model – i.e. the database is not 'saved' between function calls. Clearly, a 'real' implementation will require to save relations after updates and insertions, and will require to distinguish between those relations which may be updated and those which may not. These problems are beyond the scope of the present study.

This HOPE implementation provides an ideal vehicle for the investigation of new attribute types or domains and new functions on relations and domains. There is no reason, in principle, why extremely complex types should not be used for domains. Graphics, events and processes, text, images, speech, etc. are possible candidates for new domain types. A set of functions based solely on relational algebra limits the usefulness of a database. Domain-related functions and inferencing functions would be useful extensions to the repertoire. In addition, future parallel machines, such as ALICE[5] may some day allow an implementation of this kind to become a viable software product.

The current implementation of HOPE does not provide a large repertoire of explicit routines for input and output. While it is recognised that these functions would go against the mathematical purity of the language, they are of course necessary for the practical implementation of a database, in order to allow updates to be made and saved for future user transactions to take place in a meaningful way.

Acknowledgements

The authors would like to thank Dr S. M. Deen, Mr C. M. I. Rattray, Dr J. Darlington, Mr V. Wu, Dr D. Sannella and the referee for their valuable comments, also members of staff of the Computer Centre of Dundee College of Technology. All errors and omissions are, of course, the authors' responsibility. One of the authors, W. B. Samson, acknowledges financial support from the Science and Engineering Research Council under grant GR/B/82288.

REFERENCES

1. M. L. Brodie and J. Schmidt, What is the use of abstract data types in databases?, *Proceedings of the 4th International Conference on Very Large Data Bases*, pp. 140–141 (1978).
2. R. M. Burstall, D. B. MacQueen and D. T. Sannella, *Hope: an Experimental Applicative Language*. University of Edinburgh, Department of Computing Science Internal Report CSR-62-80 (1980).
3. E. F. Codd, Relational completeness of data base sublanguages. In *Data Base Systems*, Courant Computer Science Symposia Series, vol. 6. Prentice-Hall, Englewood Cliffs. N.J. (1972).
4. O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, A.P.I.C. Studies in Data Processing no. 8. Academic Press, London (1972).
5. J. Darlington and M. J. Reeve, Alice: a Multi-processor Reduction Machine for the Efficient Evaluation of Applicative Languages. *Proc. MIT/ACM Conference on Functional Languages and Computer Architecture* (1982).
6. J. Darlington and V. Wu, An introduction to functional programming using HOPE, *Seminar Notes* (1983).
7. C. J. Date, *An Introduction to Database Systems*, vols I and II. Addison-Wesley Publishing Company, London (1982).
8. C. J. Date, A formal definition of the relational model. From *An Introduction to Database Systems*, vol. 2 (1982).
9. S. M. Deen, D. Nikodem and A. Vashishta, The design of a canonical database system, *Computer Journal* **24**, 200–209 (1981).
10. H. Ehrig, H-J. Kreowski and H. Weber, *Algebraic*

Specification Schemes for Data Base Systems, Internal publication of Hahn-Meitner-Institut, Berlin (1978).
11. N. Gehant, Specifications: formal and informal – a case study. *Software Practice and Experience* **12**, 433–444 (1982).
12. F. Golshani, T. S. Maibaum and M. R. Sadler, A model system of algebra for database specification and query/update language support. *Proceedings of the 9th International Conference on Very Large Data Bases* (1983).
13. J. V. Guttag and J. J. Horning, The algebraic specification of abstract data types, *Acta Informatica* **10**, 27–52 (1978).
14. J. Guttag, Notes on Using Types, and type abstraction in functional programming. In *Functional Programming and its Applications*, edited J. Darlington, P. Henderson and D. A. Turner. Cambridge University Press (1982).
15. B. Liskov, Specification techniques for data abstractions. *IEEE Conference on Reliable Software Engineering* **1**, 72–87 (1975).
16. G. Louis and A. Pirotte, A denotational definition of the semantics of DRC, A domain relational calculus. *Proceedings of the 8th International Conference on Very Large Data Bases*, pp. 348–356 (1982).
17. T. S. Maibaum, Mathematical semantics and a model for data bases. In *Information Processing*, pp. 133–137. IFIP, North-Holland Publishing Company, Amsterdam (1977).
18. A. Pirotte, A precise definition of basic relational notions and of the relational algebra. *ACM SIGMOD Record*, pp. 30–45 (1982).