PROGRAMMING LANGUAGES AND DATA BASE MODELS: ON THE INTEGRATION OF CONCEPTS, CONSTRUCTS, AND NOTATIONS

J.W. Schmidt

Rapporteur: Dr. J.M. Rushby

1. Introduction: On Data Definition

Any high-level approach to data manipulation and data storage requires some mechanisms for the specification of operands by which the data can be stored and manipulated. These mechanisms generally provide the specification of:

(a) the structure of the operands, that is the rules by which they are composed out of others and by which components can be denoted;

(b) the operators applicable to the operands;

(c) the set of general constraints to be maintained on the operands.

Historically, the two areas, programming languages and data base models, developed rather independently and to a great extent introduced their own concepts and terminology. In the programming language community, data processing tools and tasks are described in terms of data structures, data types, typed variables, expressions, statements and so forth. Data base users are familiar with the concepts of data base models, schemata, data bases, queries, data base actions, transactions etc.

It is the main purpose of the paper to relate the two areas at the levels of concepts, constructs, and notations. The motivation is twofold. First, there is an increasing number of applications where the same person has to understand and even construct both programs and data bases; and it is not just a question of economy to minimise the number of concepts to be learned and taught. Second, it is the very nature of data bases and programs that within the same piece of software - even within the same statement - operands from both sources, data base and programs, are denoted.

2. Data Definition in Programs: The Specification of Types

For the purpose of data definition, programming languages provide so-called type generators that are used to generate user-defined data specifications or data types. Such specifications can in turn be applied to produce operands that serve for data storage and manipulation in accordance with the intended specification.

In the currently prevalent programming languages a user cannot write his own type generators; instead, he has to choose amongst a collection of so-called data structures, that is, type generators that are pre-defined by the language. A data structure is defined by the structuring method it provides, by the operators applicable to the operands structured by this method and possibly by some general constraints to be imposed on the values for that operands.

2.1 Structuring Methods for Data

It is one of the properties of a well-designed programming language that data structures for the most common structuring needs are provided. What these needs are depends, of course, on the application area. In this section, a few of the structuring needs that are supported by general purpose programming languages will be discussed.

Frequently, a fixed number of data of possibly different types have to be aggregated into a single operand, but the constituents must keep their individual names, for example, name f. The specification of types for this purpose is achieved by the data structure record:

```
type ftype = . . .;
     rtype = record . . . f:ftype; . . . end;
var  r : rtype;
begin . . . r.f . . . end.
```

If the data to be aggregated into a single object are of identical type and of fixed number, a proper type specification may be achieved by the data structure array. A denotation for the individual components of array variables is given by the values, for example, i, of some index type, itype:

```
type itype = . . .;
     elemtype = . . .;
     atype = array [itype] of elemtype;
var  a : atype;
begin . . . a[i] . . . end.
```

Structures that are defined by a fixed number of components, possibly of different type, and selected by some naming scheme, will subsequently be called tuple-like structures.

Sometimes, the number of identically-typed data to be collected in one operand is not fixed. In this case, a data structure set may be used:

```
type elemtype = . . .;
     stype = set of elemtype;
var  s : stype;
```

For set variables there are generally no mechanisms for the denotation of elements: the varying cardinality of sets does not allow the definition of a fixed naming scheme, and any identification of elements referring to their sequential order does not apply to (pure) sets.

In case the concept of order is applicable, operands for a varying number of identically-typed elements can be defined by the data structure sequence:

```
type elemtype = . . .;
     qtype = sequence of elemtype;
var  q : qtype;
```

Structures defined by a varying number of identically-typed elements with or without naming schemes for element selection are subsequently called set-like structures.

Quite often the data structuring needs demand a combination of several structuring methods. So, data of different types may be aggregated into one tuple-like structure and a fixed number of these compound operands may be aggregated to form another tuple-like structure:

```
type ftype = . . .;
     itype = . . .;
     artype = array [itype] of record . . . f:ftype; . . . end;
var  ar : artype;
begin . . . ar[i].f . . . end.
```

Type generators are not only characterised by their rules for the composition and decomposition of operands but also by the set of operators for the manipulation of operands.

2.2 Operations on Data

The operators defined for structured types - and inherited by their type generators - may roughly be divided into two categories:

component operators, which operate on structured operands by operating on their components, and

compound operators, which apply to structured operands as a whole.

Component operators require that structured operands can be decomposed into components down to a level where they are either of some structured type with compound operators or of some base type pre-defined in the language and well-equipped with operators. The operators defined, for example, for the tuple-like structures record and array are mostly component operators; exceptions may be the assignnment and the test-on-equality operator. Compound operators are the dominant ones for set-like structures such as set and sequence. So, the operators union and intersection apply to set operands and yield set-valued expressions; the application of the operators set-inclusion or set-element-test results in Boolean-valued expressions. Assignment statements may be formed out of set variables and set valued expressions by the compound operator assignment. Compound operators are sometimes given in the form of procedures, for example, procedure first (q:qtype; qelem:elemtype), reads a sequence variable, q, and assigns the value of the first element to a variable, qe. The application of those operators leads, of course, directly to a statement.

To summarise, structured variables may contribute to programs in different ways depending on the type generator used for their declaration:

(a) In a rather restricted role, they may serve as parameters for a set of pre-defined procedures.

(b) More generally, they may act as variables in expressions and function calls as well as in assignment and procedure statements.

(c) Last but not least, they may constitute a pool of component variables for expressions, functions calls and statements.

2.3 Constraints on Data

In some programming languages the user can specify constraints to be maintained on operands. Imposing upper and lower bounds on the values to be accepted by an unstructed variable is a common example:

```
type ftype = 1 . . 19;
```

Constraints are inherited by structured types based on constrained types:

```
type elemtype = 1 . . 19;
     stype = set of elemtype;
var  s : stype;
```

The constraint composed on the set variable, s, reads

s <= { i in integer : (i>=1) and (i<=19) }.

In the currently prevalent programming languages the mechanisms for constraint specification are rather limited.

3. Data Definition in Data Bases: The Specification of Schemata

As with data definition in programs there is no general agreement on the 'optimal' structuring methods for data bases. While, however, programming languages offer a spectrum of data structures, each of the currently prevalent data base models is heavily biased towards one structuring methods. So, it is no surprise that there have been rather emotional discussions on what would be 'the best data base model'. This discussion has cooled down since ANSI/SPARC proposed a data base architecture with several structuring methods at various levels.

Regardless of their differences, the major data base models - at least those based on relations, hierarchies, and nets - may all be described in terms of the two classes of structuring methods introduced above: tuple-like structures allow the definition of data composed out of a fixed (and mostly small) number of components of different types, set-like structures are used if the elements are of identical type and if their number varies (and may be high). By the tuple-like method, data base components can be defined for the following two purposes: holding the description of either one 'real world' object given by its k attributes or one association between l 'real world' objects (k,l constant). The set-like structuring method leads to data base components that can keep n (n variable, n >> k,l) elements of identically-typed 'real world' objects or associations.

In the next section, the structuring methods underlying the three major data base models will be discussed in some detail.

3.1 The Relational Data Base Model

Three type generators are required to define a relational data base schema, one set-like and two tuple-like structures:

data base defines a relational data base by a fixed number of named components;

relation defines a data base component so that it can hold a varying number of relation elements of identical type;

relation elements are supposed to be defined by some tuple-like structure, traditionally by the data structure record.

The data base and record are tuple-like structures; relation is a set-like structure.

```
type ftype = . . .;
     reldbtype = database . . .;
                          relk:relation <f> of
                               record . . . f:ftype; . . . end;
                          . . .
                 end;
var  reldb : reldbtype;
```

The constraint imposed on the data base component, reldb.relk, - its key, f - means that no two elements of the relation may have the same value for the element component identified by f.

A data structure relation can be regarded as a generalisation of the data structure set - as defined, for example, in Pascal - in the following sense: the types for set elements, selemtype, are restricted to be unstructured, for example, of type character or of a constrained type defined on integers. The constraint defined on set types is uniqueness of set elements.

```
type selemtype = char;
     stype = set of selemtype;
```

Relation element types, relemtype, may be structured unlike set element types, selemtype. Therefore the constraint of uniqueness of elements can be generalised to uniqueness of element components or a list thereof.

```
type relemtype = record . . . f:...; g:...; . . . end;
     reltype = relation <f,g> of relemtype;
```

Further aspects of that generalisation will be seen when operations on relations are discussed.

3.2 The Hierarchical Data Base Model

For the definition of hierarchical data base schema (or more exactly: tree-structured data base schemats) four type generators are required, two set-like and two tuple-like structures:

data base defines a hierarchical data base to be a set-like structure of elements;

elements of a hierarchical data base are defined by a structuring method tree and have a fixed number of components of different types; the components are defined either by the tuple-like structure record or by the set-like structure sequence.

Since the elements of a sequence are allowed to be of some tree type, a multi-level tree-structured schema can be defined:

```
type ftype : . . .;
     gtype : . . .;
     treedbtype = database <f> of
                      tree parent:record...f:ftype;...end;
                           children1:sequence <...> of
                                     tree parent: . . .;
                                          children1:...;
                                          ...
                                     end;
                           children2:sequence <g> of
                                     record...g:gtype;...end;
                           . . .;
                           childrenk:sequence <...> of...
                      end;
var  treedb : treedbtype;
```

The example data base, treedb, can hold a varying number of tree-structured elements each associating one parent component of some record type with k children components of different types. The constraint denoted by <f> may mean that there is at most one element of the data base that has a given value in the component f of its parent component. The children named children1 to childrenk are defined as sequences of elements since the concept of order shall apply. A constraint imposed on, say, the sequence treedb.children2 and denoted by <g> may mean that there is at most one element in the sequence with a given value in its component g.

By the given combination of set- and tuple-like structures 1:n associations may easily be defined. A generalisation to n:m relationships requires that the same element type can be used at various levels in the hierarchy and that there are means with which to relate identical elements at various levels. Then, of course, the structure is generalised from a hierarchy to a network.

3.3 The Network Data Base Model

The basic structuring method for the network data base model associates a fixed number of components of different types and may be called net: one component is of a tuple-like structure (owner) and k components are of a set-like structure (members). Each member component can hold a sequence of elements to be associated with the owner component thus forming k different owner/member relationships. A network data base basically consists of l components that are sets of identically-typed nets.

```
type ftype = . . .;
     gtype = . . .;
     elemtypek = record . . . f:ftype; . . . end;
     elemtypekl= record . . . g:gtype; . . . end;
     elemtypel = . . .;
     netdbtype = database . . .;
                      compk:set <f> of
                            net owner:elemtypek;
                                . . .;
                                membr:sequence <g> of elemtypekl;
                                . . .;
                            end;
                      . . .;
                      compl:set <...> of
                            net owner:elemtypel;
                                . . .;
                                membs:sequence <...> of elemtypekl;
                                . . .;
                            end
                end;
var netdb : netdbtype;
```

In a network data base an element type, for example, elemtypekl, may contribute to the definition of more than one net type. Furthermore, the network model provides operators to insert identical elements into different components of nets and to exploit that fact when navigating through a network data base. The distinguished indentifiers <f> and <g> may be considered as examples of constraints on the network data base:

compk:set <f> of net . . .; shall mean that there is at the most one net element in the set, compk, that has given value in the component f of its owner;
membr:sequence <g> of elemtypekl; shall mean that there is at the most one record element in the sequence, membr, with a given value in its component, g. One can consider the whole network data base to be formed out of 1 components where each one can hold a set of elements of some net type. For the sake of simplicity many details that may be considered as part of a schema definition have been omitted. Examples are order clauses for sequences or various declarations supporting element selection within sets and sequences. In the next chapter, additional properties of data base models will be discussed, primarily the mechanisms for element selection and the operators.

4. Selection Mechanisms

Ultimately, it is the 'raison d'etre' for any data base - as for any structured variable - to provide the user with components either for the purpose of reading or for writing. In this chapter some of the mechanisms for component selection as defined for data structures will be discussed and the results will be applied to the structuring methods of data base models.

4.1 Selection Mechanisms for Data Structures

Data structures in programming languages allow component selection in two different ways. Either they provide a denotation to select component variables or they provide operators to form expressions returning component values. The data structures array and record with their denotations indexing, a[i], and qualifications, rec.f, are of the first kind. The data structure sequence with a selection function first(q) or a procedure first(q,qe) is an example for the second kind.

Selection by component denotation is the more general one in the sense that it leads to variables, and variables may exist in both constructs, expressions (right-hand-sides) and statements (left-hand-sides).

Selection by component denotation in turn may be divided into two categories: selection by expression as for example indexing of arrays, a[i+1], where the selector is computed by an expression, i+1, and can be stored in a variable; and selection by identifier as for example qualification of records, rec.f, where the selector, f, is defined in the type declaration, that is, in the program text.

4.2 Selection Mechanisms for Data Base Models

In chapter 3 it was shown how a complex data base schema definition can be composed or primitive types by repeated application of a few structuring methods. Now, some selection mechanisms for these structuring methods will be discussed allowing a data base to be decomposed into its components. The following discussion will mainly concentrate on the relational data base model.

A few general requirements for selection mechanisms on relations can be postulated. Frequently, only one or a few of the many elements of a relation are of interest for reading, updating, or existence testing. Therefore a denotation allowing the selection of individual relation elements as variables is desirable. The alternative would be to operate by compound operators on whole relation variables. Furthermore, since relations are intended to hold logically related data a selection mechanism should be based on selectors that in turn can be stored in the data base. In this case k elements, possibly from different relations, can be related by an element in a further relation holding the k element selectors. In addition, it is desirable that the selector for the relating element can be easily composed from the individual selectors of the related elements. Then the selection mechanism for relations would equally support the selection of the k related elements if the selector of the relating element is given and the selection of the relating element if the k selectors of the related element are at hand.

Symmetry arguments like these lead to a selection mechanism where the k selectors stored in the relating element are the selectors for that element and where the selectors of the related elements must be stored as components of these elements. In other words, an appropriate selection mechanism for the structuring method relation is based on the content of the relation elements as opposed to the name- or address-oriented selection mechanisms for data structures like record or array.

If the selection mechanism is based on those element components distinguished by the key of a relation it is guaranteed that unique elements are denoted:

```
type ftype = . . .;
     relemtype = record . . . f:ftype;. . . end;
     reltype   = relation <f> of relemtype;  . . . ;
     reldbtype = database . . . rel:reltype; . . . end;
var  reldb : reldbtype;
begin . . . reldb.rel <fe> . . . end.
```

fe denotes an expression of type ftype.

Elements in a hierarchical data base, for example treedb, as defined in section 3.2 are selected by

```
type . . . {see section 3.2};
     treedbtype = . . . ;
var  treedb : treedbtype;
begin . . . treedb<fe>.children2<ge> . . . end.
```

Acordingly, element selection for a network data base like netdb defined in section 3.3 reads

```
type . . . { see section 3.3 } ;
     netdbtype = . . . ;
var netdb : netdbtype;
begin . . . netdb.compk<fe>.membr<ge> . . . end.
```

fe and ge are expressions of type ftype and gtype.

It should be noted that commercial data base systems provide many extra selection mechanisms based on additional clauses in the schema definition.

The rest of the paper will concentrate on the operators defined with a data base model. While gradually extending the set of operators and generalising the selection mechanism, the requirements for the interface between a data base model and a programming language will be analysed.

5. Data Base Manipulation: A Minimal Approach

A data structuring method is not completely defined without the operators applicable to the operands being defined with this method. For the relational data base model, operators are required so that expressions and statements can be formed out of data base components (that is, relations) and relation elements (that is, records). In this chapter, a minimal set of operators for the relational model is introduced and it is shown how expressions on data base (component) variables can be used as operands in program statements and vice versa.

5.1 Operations on Relations

An operator absolutely necessary for relations or at least for relation elements is the assignment operator, := (or some equivalent procedure). Otherwise, there were no read and write operations on relational data bases.

If a relational data base, reldb, is defined by

```
type ftype = . . .;
     relemtype = record . . . f:ftype; . . . end;
     reltype   = relation <f> of relemtype; . . .;
var  reldb : database . . . rel:reltype; . . . end;
```

the assignment statement reldb.rel<fe> := e; requires that the expression, fe, is of type ftype, and that the expression, e, is of the type of the left-hand-side variable, that is, of type relemtype with the restriction that the value set for the component, f, is restricted to the value of fe. An additional operator, in, tests whether a relation, rel, contains an element equal to some expression.

The expression reldb.rel<fe> in reldb.rel becomes true iff the designated relation element is already initialised, that is, assigned.

It is of some notational convenience if a notation is provided that allows the denotation of data base components without preceding them with the entire data base identifiers:

```
with reldb do
begin . . .;
     rel<fe> := e;
      . . . rel<fe> in rel . . .;
end.
```

5.2 Interfacing Programming Languages and Data Base Models

There are two different ways of providing operations for data bases. One approach regards a data base model as being self-contained, perhaps after adding a few more operators to increase the computational possibilities of the model. The other approach allows the definition of data bases and data base operations within the scope of programs written in some programming language: this is termed the host language approach. This discussion will follow the latter approach.

There are some general requirements a programming language and a data base model should meet when being interfaced:

(a) variables and expressions that are well-formed in the sense of a data base should be accepted by a program statement unless type conditions are violated;

(b) variables and expressions that are well-formed in the sense of a program should be accepted by a data base statement unless type conditions are violated.

A programming language to be interfaced with a relational data base model as defined above should therefore accept expressions of the relation element type, that is, the result of an element selection, and of Boolean type that is, the result of an element test. And programming language expressions should be accepted in statements that assign or test relation elements. In the following examples the interface requirements between the programming language Pascal and a relational data base model are discussed.

Example: Insertion

```
program dbuser (reldb);        { imports the relational database, reldb}
type ktype = 1 . . 100;
     relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype ; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
     rec : relemtype;
begin with reldb, rec do
      begin . . .; key := 10; . . .; { initialisation of rec}
          if not rel<key> in rel
          then rel<key> := rec        { insertion of rec }
      end
end.
```

Example: Replacement

```
program dbuser (reldb);          { imports the relational database, reldb}
type ktype = 1 . . 100;
     relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
     rec : relemtype;
begin with reldb, rec do
      begin . . .; key := 10; . . .;   { initialisation of rec }
          if rel<key> in rel
          then rel<key> := rec          { replacement by rec   }
      end
end.
```

So, in a minimal approach the insertion and replacement operations are accomplished by the assignment and the test operator of the data base model and the if-then-else control structure of the programming language.

A less stingy approach may introduce operators for assignment, insertion, replacement etc., as compound operators on relations: the insert operator, :+, is defined so that the statement rel :+ [rec]; has the same meaning as the statement with rec do if not rel<key> in rel then rel<key> := rec;

The relation constructor, [...], constructs one-element relation expressions out of record expressions.

Analogously, the replacement statement, rel :& [rec]; is defined to be equivalent to
with rec do if rel<key> in rel then rel<key> := rec;

Last but not least, the delete statement rel :- [rec]; deletes the relation elements equal to rec - if it exists.

In section 7.2 it will be shown how the relation update operators, :+, :&, :-, can be replaced by the ordinary assignment operator, :=, at the expense of a more complicated right-hand-side expression.

The denotation for relation components can also be used when data from the data base are to be processed by a program:

Example: Reading a Database

```
program dbuser (reldb);          { imports the relational database, reldb }
type ktype = 1 . . 100;
     relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
     rec   : relemtype;
     ck    :ktype
begin with reldb do
      begin ck : = 1;
          while ck <= 100 do
          begin if rel <ck> in rel
                then begin rec := rel<ck>;
                           . . . { processing of rec } . . .;
                     end
                ck := ck + 1
          end
      end
end.
```

Of course, this solution is intolerable if many of the relation elements are not assigned, for example, rel <ck> in rel = false, or when the cardinality of the key value set, that is, the value set of type ktype, is very large. If an order is defined on the value set of the key component(s) additional selection mechanism on relations can be introduced that return the value of a relation element:

The procedure low (rel,relem) assigns the value of the relation element with the lowest key value to a variable, relem; the procedure next (rel,relem) returns the value of the next highest element. Analogously, the pair of procedures high and prior can be defined. The procedure this (rel,relem) is defined to be equivalent to the statement

```
if rel<relem.key> in rel then relem := rel<relem.key>;
```

A Boolean procedure end-of-relation, eor(rel), becomes true if and only if the element to be selected does not exist.

With these selection mechanisms the previous version of the program reading a data base may be replaced by the following one:

Example: Reading a Database

```
program dbuser (reldb);   { imports the relational database, reldb}
type ktype = 1 . . 100;
     relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database  . . . rel:reltype; . . . end;
     rec : relemtype;
begin with reldb do
      begin low (rel, rec);
            while not eor (rel) do
            begin . . . { processing of rec } . . .;
                  next (rel, rec)
            end
      end
end.
```

These examples show that a data base model with just a minimal set of operators, assignment and test, and element-orientated, key-based selection mechanisms can easily be interfaced to a programming language if the data structures for element definition correspond, and if appropriate control structures exist. The interface requirements definitely become harder to fulfill if the selection mechanisms for relations are generalised.

6. Predicate-Oriented Selection Mechanisms

The two selection mechanisms introduced for relations both operate key-orientated and one-element-at-a-time: the denotation rel<key>selects one element as a variable; the procedures low, next, this, high, and prior return the value of one relation element. These selection mechanisms are based on some predicates that either test key values for equality or for some minimum or maximum condition. And since key values are guaranteed to be unique within relations, at the most one element is selected.

For some purposes it is of interest to equip a data base model with a selection mechanism based on a more general class of selection predicates. Users should be able to specify the data to be selected all at once, and this in turn may enable an implementation to find these data - or at least some of them - by one effort. Subsequently, a notation for a general predicate-orientated selection mechanism will be proposed.

In the Pascal language the elements for a set-valued expression can be specified by the denotation nlow .. nhigh; the expressions, nlow and nhigh, return values from some base-type value set and nlow .. nhigh denotes each value of that value set between these limits. In other words, set elements are selected from a base value set - that is unstructured and constant - by a denotation equivalent to

each e in base-set : (e >= nlow) and (e <=nhigh)

The (free) element variable, e, has to be introduced to denote the relation elements so that a selection criterion can be specified by the predicate following the colon. Adopting this notation for relation selection leads to
each r in rel : p(r),
where three generalisations have been made:

(a) the structure of the elements to be selected is no longer restricted to some unstructured base type as, for example, integer; instead any type that is admitted as a relation element type is allowed;

(b) the value set is no longer given by a constant set, for example, set of integers or a subset thereof; instead, it is given by any relation expression;

(c) the selection predicate is no longer restricted to predicates defining closed intervals; instead, any Boolean expression is admitted.

A data base model equipped with this general content-orientated mechanism that selects n elements at a time is, of course, more powerful than one supporting only one-element selection. On the other hand, it demands more, not only from the data base management system implementing that general content-orientated selection mechanism on address-orientated storage hardware, but also from the programming environment that wants to make use of this more powerful selection mechanism. In the next chapter, some of the consequences and alternatives for the interface between data base models and programming languages will be discussed.

7. Data Base Manipulation Revisited: Control Structures vs. Data Structures

In the previous chapter, a selection mechanism for relational data bases was introduced which selects n elements at a time. A data manipulating environment - as defined by some programming language - has to meet certain requirements before it can profit from this selection mechanism. In essence, there are two alternative solutions: the programming language has to provide structures either for controlling the execution of statements so that the elements selected from the data base are assigned one after the other, or for structuring variables so that the elements selected from the data base can be assigned all at once.

The two alternatives may be characterised by a trade-off between time and space. The time-oriented solution requires one assignment statement for each selected element; however, it needs only minimal space - that is, space for one relation element. The space-oriented

solution requires space for all the selected elements at once; however, it needs only time for one relation assignment. Both alternatives will be discussed in some detail.

7.1 Control Structures

The time-oriented solution demands a control structure that executes statements for each element of a given selection. The repetitive statement used when the number of repetitions is known beforehand is commonly called a for-loop. The interface between Pascal and the relational data base model may be achieved by admitting the n-element selection mechanism each r in rel : p(r) as a control mechanism within the for-statement:

```
for each r in rel : p(r) do
begin . . . { processing of r } . . . end;
```

It should be noted that this is the first time within our approach that an extension of the programming language is demanded; and the extension refers to a construct that is already present in the language.

A for-statement may be used to implement queries against a data base that have Boolean results. As an example, take

```
program dbuser (reldb);
type relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
     q : boolean;
begin with reldb do
      begin q := false;
            for each r in rel : true do q := q or p(r);
            if q then . . .
      end
end.
```

The Boolean variable, q, becomes true as soon as a relation element exists that makes the predicate p(r) true. The for-statement essentially implements a disjunction of Boolean expressions, p(ri), each evaluated for another relation element, ri. In terms of first-order predicate calculus this disjunction is equivalent to an expression defined by an existential quantifier. If q is initialised by false and if the operator or is replaced by and the value computed for q is equal to the value of a universally quantified expression.

It increases the expressive power of the model, the efficiency of its implementation, and is of notational convenience if the data base model allows quantified expressions. In this case the previous example reads:

```
program dbuser (reldb);
type relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
begin with reldb do
      begin . . . if some r in rel (p(r)) then . . . end
end.
```

Furthermore, quantified expressions substantially extend the selection mechanism for relations. Given two relations, rel1 and rel2, a selection each r1 in rel1 : some r2 in rel2 (p(r1,r2)) can be formed.

7.2 Data Structures

The space-orientated solution demands a data structure such that operands can be defined that accept all the selected elements by one assignment. It fits into our approach to extend the programming language for that purpose by the same data structure relation introduced with the data base model. Then, any selection of elements can be transferred from a data base into a program by one assignment operation, provided that the relation constructor, [...], introduced in section 5.2 is generalised so that it constructs relation-valued expressions from n-element selections.

```
program dbuser (reldb);
type relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . .rel:reltype; . . . end;
     result : reltype;
begin with reldb do
      result := [each r in rel : p(r)];
      . . .
end.
```

This space-orientated solution may be contrasted with its time-orientated equivalent:

```
program dbuser (reldb);
type relemtype = record . . . key:ktype; . . . end;
     reltype   = relation <key> of relemtype; . . . ;
var  reldb : database . . . rel:reltype; . . . end;
     result : reltype;
begin result : = [ ];
      with reldb do
      for each r in rel : p(r) do result : + [r];
      . . .
end
```

Extending a programming language by the intrinsic structuring method of a data base model provides, of course, the most intimate interface between programs and data bases: relation variables from programs and data bases now can be used intermixed in element selections, relation expressions, and in statements.

A final example will demonstrate that the relation update operators, :+, :&, :-, introduced in section 5.2 are just shorthand notations of ordinary assignment statements.

An insert statement, for example,

```
rel1 :+ rel2;
```

is equivalent to the assignment statement

```
rel1 := [each r1 in rel1 : true,
         each r2 in rel2 : all r1 in rel1 (r1.key <> r2.key)];
```

In this example, a relation expression is given by a list of selections (for the full syntax, see appendix); the first list element consists of all the relation elements of rel1, the second list element is a selection of those elements of rel2 that have key values different to those in rel1. This intimate interface between programs and data bases allows, for example, that the data base components can even be used as parameters in procedures and functions. If the procedure concept has the appropriate parameter mechanisms 'transaction procedures' may be formed that can be treated as units of operation even if several of them are executed in parallel on the same data base.

8. Summary and Concluding Remarks

The paper addresses two topics. At first it defines data base models in terms of programming language concepts. Type generators (data base models) are used to write specifications (data base schemata) that define how data are structured and identified, manipulated and constrained. Expression (data base queries) are formed out of operands (data base components) and operators and denote rules for obtaining results. Statements (data base actions and transactions) denote operations that may modify their operands when being executed. Furthermore, the paper discusses some of the mutual requirements to be met by programming languages and data base models so that operands from both sources can be mixed within statements. The example given by the programming language Pascal and the relational data base model shows that, depending on the selection mechanism defined for relations, interfacing needs either no modification of the language at all or it requires the generalisation of an existing control structure or the introduction of a new data structure.

Neither the conventional approach for an interface based on a third component (user working area) which is mainly defined by a data structure mediating between programs and data bases nor the more advanced idea of extending a data base model by a control structure are discussed in this paper. Also, new concepts for data definitions, for example, user-defined type generators as in Euclid, Modula, or Ada (cluster, module, or package) are not exploited.

A Pascal system extended by relations and relational data bases (Pascal/R, see appendix) is implemented on a DECsystem/10.

Discussion

Professor van de Riet asked why, in Pascal/R, relations were taken as a generalisation of the set type rather than the file type, and also since relations are collections of records and Pascal allows pointers to records, why were there no pointers to tuples in relations? Professor Schmidt replied that they had started with the set type because it provided a mechanism already present in Pascal for the non-procedural generation of a collection of elements. Work had been done on attempting to integrate the notion of reference with that of relation - he recalled a paper from University of Toronto CSRG of a couple of years back - the techniques were similar to those used in Euclid: a reference had to be bound not merely to a type but to a variable.

Dr. Atkinson asked why, since they had introduced a new data type, had they not also introduced new operators for that type - for example the algebraic operators. Professor Schmidt replied that they had introduced primarily a new data structuring tool. Since quantification is allowed in relation selectors, operators such as join etc. can be built up out of these more basic things. But, Dr. Atkinson continued, since they provided expressions like R:-E, why not generalised set differences, for example? Professor Schmidt responded that these things can be done with the basic tools provided; R:-E is not a new operator - merely a shorthand.

Closing the discussion, Professor Randell asked what experience they had with Pascal/R as a teaching tool. Professor Schmidt said that they had taught it for a number of terms and had recently used it for student exercises. He thought their experience could be summed up in the words of one student who, after taking a course, asked 'what has Pascal/R to do with Data Bases? - it's just programming'. Professor Schmidt regarded this as a compliment and suggested that it proved that relational data bases were more naturally seen as an extension to the data structuring tools available in programmming languages, not as some special subject.

APPENDIX

Syntax of the Pascal/R language given as extensions to the definition of the Pascal language (see Pascal Report in K. Jensen, N. Wirth: Pascal User Manual and Report. Springer Verlag, New York, Heidelberg, Berlin 1975, 2nd Edition):

Notation, terminology, and vocabulary

```
<special symbol> ::= ...
:+ | :- | :& | all | some | each | relation | database
```

Data type definitions

```
<unpacked structured type> ::= <array type> |
    <record type> | <set type | <file type> |
    <relation type> | <database type>

<relation type> ::=
    relation < <relation key> > of <relation element type>
<relation key> ::=
    <key component identifer>  {,<key component identifier>}

<key component identifier> ::= <identifier>

<database type> ::=
   database <database section> {;<database section>} end
<database section> ::= <database component identifier>
                       {,<database component identifier>} :
                       <relation type> | <empty>
```

Declarations and denotations of variables

```
<component variable> ::= <indexed variable> |
    <field designator> | <file buffer> |
    <database component designator> | <selected variable>

<database component designator> ::=
    <database variable>.<database component identifier>
<database variable> ::= <identifier>
<database component identifier> ::= <identifier>

<selected variable> ::=
    <relation variable> [ <expression> {,<expression> } ]
<relation variable> ::= <variable>
```

Expressions

```
<factor> ::= <variable> | <unsigned integer> |
             <function designator> | <set> | <relation> |
             <quantified expression> | ( <expression> ) |
             not <factor>
<relation> ::= [ <relation element list> ]
<relation element list> ::=
     <relation element> {,<relation element> } | <empty>
<relation element> ::= <expression> | <selection> |
                       <component selection>
<selection> ::= <element denotation list> : <selection expression>
<component selection> ::= <component list> of <selection>
<element denotation list> ::=
      <element denotation> {,<element denotation>}
<element denotation> ::=
     each <element variable> in <relation expression>
<component list> ::=
    < <component designator> {,<component designator>} >
<component designator ::=
    <element variable>.<component identifier>
<element variable> ::= <variable identifier>
<variable identifier> ::= <identifier>
<selection expression> ::= <Boolean expression>
<relation expression> ::= <expression>
<Boolean expression> ::= <expression>

<quantified expression> ::= <quantifier> <element variable>
                            in <relation expression> <predicate>
<quantifier> ::= some | all
<predicate> ::= (<selection expression>) |
                <quantified expression>
```

Statements

```
<assignment statement> ::= <variable> := <expression> |
    <function identifier> := <expression> |
    <relation variable> <relation update operator>
                            <relation expression>
<relation update operator> ::= :+ | :- | :&

<for statement> ::=
     for <control section> do <statement>
<control section> ::=
     <control variable> := <for list> | <selection>

<with statement> ::=
     with <with variable list> do <statement>
<with variable list> ::= <with variable> {,<with variable>}
<with variable> ::= <record variable> | <database variable>
```

Relation handling procedures and functions

low (r, relem), next (r, relem), this (r, relem),
high (r,relem), prior (r,relem),
eor (r).