Some High Level Language Constructs for Data of Type Relation

JOACHIM W. SCHMIDT
Universität Hamburg, West Germany

For the extension of high level languages by data types of mode relation, three language constructs are proposed and discussed: a repetition statement controlled by relations, predicates as a generalization of Boolean expressions, and a constructor for relations using predicates. The language constructs are developed step by step starting with a set of elementary operations on relations. They are designed to fit into PASCAL without introducing too many additional concepts.

Key Words and Phrases: database, relational model, relational calculus, data type, high level language, nonprocedural language, language extension
CR Categories: 4.22, 4.33, 4.34

1. INTRODUCTION

A certain class of programming problems involves the processing of data with the following properties: there is a large amount of data, the data has internal connections, and the data must be made available to many users.

Since Codd's original paper [5], relations have been increasingly used in the solution of such database programming problems.

The "classical" language constructs for the processing of data organized around relations are by now accepted to be essentially: (a) primitive instructions for altering relations at the level of individual tuples: insertion, deletion, and modification; and (b) powerful retrieval facilities operating on relations at the level of sets of tuples: relational calculus- and algebra-oriented query languages.

In recent years numerous data sublanguages have been proposed, and some implemented, which contain these constructs to a greater or lesser extent. They differ from one another mainly in the conceptions of what user friendliness means [2, 4, 17].

A version of this paper was presented at the International Conference on the Management of Data, 1977, in Toronto, Canada—an annual conference of ACM SIGMOD.

Author's permanent address: Universität Hamburg, Institut für Informatik, Schlüterstrasse 70, D-2000 Hamburg 13, West Germany. Present address: University of Toronto, Computer Systems Research Group, Toronto M5S 1A4, Canada.

Traditionally a database with its associated data language is seen as an independent system; data is interchanged with users or with programmed systems through fixed interfaces in the form of I/O areas. Problems which arise from the integration of database language constructs, such as a data structure relation, in high level languages have up to now seldom been investigated [7, 1, 14]. Such investigations are of interest from at least two points of view: first for the extension of existing high level languages, and second for the further development of database concepts themselves, in particular the relation concept.

The currently prevalent high level programming languages have no constructs for the processing of large amounts of interrelated data. The file concept does not offer a general solution to this problem; files may be able to hold large quantities of data but the connections between data elements are inadequately handled, both at the level of high level language operations and at the level of access paths. As demonstrated by the example of the programming language SAIL [8] which contains ALGOL 60 and LEAP [12] data structures, there exists quite clearly a need for an algorithmic language with efficient constructs for handling intricately connected data.

Such investigations can also serve as a vehicle for the further development of the relation concept itself. The necessity for such development shows up, for example, in the previously mentioned difference in power between altering instructions and retrieval facilities. Whereas for a single retrieval command, all connections between relation tuples which are necessary for the answer to a query are evaluated, only a single tuple in a single relation can be inserted, deleted, or updated by an altering instruction. The user must therefore, in general, code a consistent database alteration transaction as a sequence of such altering instructions, each affecting a single relation. The user thereby bears most of the responsibility for the central problem of the integrity of the database.

A similar problem situation in general purpose programming languages led to the development of the concept of abstract data types [10]. A stack variable, for example, could be implemented in a high level programming language by means of an array variable, a Boolean variable, and an integer variable. These variables must be altered consistently when the value of the stack variable is changed (e.g. by push or pop). We could analogously define alterations in a database as operations on abstract data types and implement them as procedures on relations. In more recent work, Schmid and Swenson [13] for example, the beginnings of such a development can be discerned. The further development of database language constructs, however, cannot be discussed in the limited context of data sublanguages. On the contrary, this assumes to a great extent the concepts of high level programming languages.

These tendencies will not be further discussed in this paper (see e.g. [15]). In Section 2 the definition of types of mode relation and the declaration of relation variables are briefly described, together with the elementary read and write operations for relations. Section 3 introduces a repetition statement controlled by a relation variable. The generalization of Boolean expressions to predicates is handled in Section 4. In Section 5 the concept of a generalized relation constructor is discussed. Finally Section 6 outlines the state of the implementation and some further de-

velopment. It should be noted that the language constructs have been designed to fit into PASCAL [16] without the introduction of too many additional concepts.

2. ELEMENTARY OPERATIONS ON RELATIONS

The definition of data types of mode relation is based upon data types of mode record. The value of a variable of a particular relation type is a set of tuples, each of which is in turn of the record type laid down in the definition of the relation type. The fields of these records will be taken to be of scalar type or of type string ("flat" records). Furthermore these types are presumed to be ordered types.

A second component of the definition of a relation type is the designation of certain fields of the relation tuples as a key. A key list characterizes a particular part of the tuple by enumerating the corresponding field identifiers. For these fields it holds that among the tuples of the relation a particular value assignation occurs at most once. The values of the key fields therefore uniquely determine a tuple in the relation. An ordering of the key values is defined by the presumed ordering on the values of the individual key fields and by the order of the key field identifiers in the key list.

Example 1. If employees are characterized by the attributes employee number (unique), employee status, and employee name, then we can define relation variable *employees* as follows:

```
.
.
type erectype = record enr, estatus:integer; ename:string end;
     erreltype = relation ⟨enr⟩ of erectype;
var  employees:erreltype;
.
.
```

Along with data types of mode relation, operations are defined which allow values of relation variables to be altered and to be read tuple by tuple.

2.1 Elementary Altering Operations

A value change of a relation variable can occur through insertion, deletion, or modification of tuples. These operations are not in fact "elementary" insofar as through them not single tuples, but whole sets of tuples (i.e. again a relation variable), can be inserted, deleted, or modified. However, these operations alter at any given time the value of only one relation variable.

The *insertion operator* :+ brings about the insertion of tuples into the target operand to its left, dependent on the source operand to its right:

```
rel1 :+ rel2;
```

Source and target operands are relation variables of the same type. Into *rel*1 are inserted copies of those tuples of *rel*2 whose key values do not already occur in any tuple of *rel*1. The source operand *rel*2 remains unchanged.

The *deletion operator* :— brings about the deletion of tuples in the target operand

dependent on the source operand:

```
rel1 :- rel2;
```

Source and target operands are relation variables of the same type. From *rel*1 are deleted those tuples also contained in *rel*2. The source operand *rel*2 remains unchanged.

The *replacement operator* :& brings about the replacement of tuples in the target operand dependent on the source operand:

```
rel1 :& rel2;
```

Source and target operands are relation variables of the same type. Each tuple of *rel*1 whose key value occurs in a tuple of *rel*2 is replaced by a copy of the corresponding tuple in *rel*2. The source operand remains unchanged. Note that key values cannot be changed with the aid of the replacement operator.

The *assignment operator* := assigns relation values between relation variables of the same type:

```
rel1 := rel2;
```

A generalization of admissable source operands towards relational expressions will be treated in Section 5.

2.2 An Elementary Relation Constructor

An anonymous 1-tuple relation can be constructed from a record variable with the aid of the elementary relation constructor [..]. Of course any combination of fields fulfills for this relation the requirements for a key; these relations are therefore type-compatible with every relation variable whose type definition is based on the type of the record variable, e.g.

```
rel := [rec];
```

The empty relation is denoted correspondingly by [ ].

Example 2. A tuple with employee number 2, employee name Nessie, and employee status 1 is to be inserted in the relation *employees*.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
var   erec:erectype;
      employees:ereltype;
begin .
      .
      .
      erec.enr := 2;
      erec.estatus := 1;
      erec.ename := 'nessie';
      employees :+ [erec]
end.
```

2.3 Elementary Retrieval Operations

Two standard procedures low(*rel*) and next(*rel*) are defined for the tuple-wise reading of relation variables. Furthermore the Boolean function aor(*rel*) (*all of* relation) and for each relation an implicitly declared buffer variable *rel*↑ are available.

The tuple of the relation *rel* with the lowest key value is assigned to the buffer *rel*↑ by the procedure call low(*rel*). The tuple with the next highest key value is assigned to *rel*↑ by the procedure call next(*rel*). If such a tuple does not exist then aor(*rel*) becomes true and *rel*↑ becomes undefined.

Example 3. Find those employees with the status of an assistant professor (employee status 2).

Solution 3.1.

```
.
.
type  ereclype = record enr, estatus:integer; ename:string end;
      erelype = relation ⟨enr⟩ of ereclype;
var   employees, result:erelltype;
begin .
      :
      result := [ ];
      low(employees);
      while not aor(employees) do
      begin if employees↑.estatus = 2 then result :+ [employees↑];
            next(employees)
      end
end.
```

The solution method underlying this example program accesses relations tuple-wise and in order, in the main controlled by the user. It is not generally satisfying for several reasons:

The ordering of tuple access by increasing key value is unnecessary from the point of view of program logic.

With respect to problem orientation the language constructs available for relations up to now are inadequate; this rapidly becomes evident with more complex examples. The above program is a solution in terms of the file concept and not in terms of relational databases.

For particular problems these elementary constructs are insufficient, e.g. nested loops accessing the same relation cannot be programmed.

The notation could be more concise.

There is little possibility for automatic optimization of accesses, for example by free choice of ordering or by processing of sets of tuples.

In the following sections we will develop step-by-step language constructs which should to a great extent overcome such objections.

3. THE REPETITION STATEMENT foreach

The foreach statement has the general form:

```
⟨foreach statement⟩ ::= foreach ⟨control record variable⟩
                            in ⟨range relation variable⟩
                           do ⟨statement⟩
```

The execution of a statement may be repeated with the aid of the foreach statement. For each repetition the control variable is assigned an arbitrary new tuple from the range relation until all tuples of the relation have been used. The control variable is declared implicitly to have the same record type that the range relation is based on. The scope of the control variable definition is the statement following **do**; within this scope the key values of the control variable and the range relation must not be altered. By using the foreach statement the solution for Example 3 becomes:

Solution 3.2.

```
.
.
{type definitions and variable declarations as in 3.1.}
.
.
begin .
      :
      result := [ ];
      foreach erec in employees do
        if erec.status = 2 then result :+ [erec]
end.
```

For a further example we introduce the relation *timetable* listing lectures held by employees and described by attributes employee number and course number, together with day, time, and place of each lecture.

Example 4. Find all those employees who lecture on Fridays.

Solution 4.1.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation <enr> of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation <tenr, tcnr, tday> of trectype;
var   employees, result:ereltype;
      timetable:treltype;
begin .
      :
      result := [ ];
      foreach erec in employees do
        foreach trec in timetable do
          if (erec.enr = trec.tenr) and (trec.tday = 'friday')
          then result :+ [erec]
end.
```

If we look at the example and its solution more closely we notice that:

The inner loop with range relation *timetable* is in general traversed too often; that is, whenever an entry for a particular employee's lecture held on Friday has been found, the remainder of the relation *timetable* is nevertheless processed.

If an employee lectures several times on a Friday, the corresponding employee record is inserted several times into the result. By virtue of the definition of the insertion operator this does not, however, affect the value of the target operand.

It would also be possible to avoid repeated insertions quite easily by programming an extra exit from the loop. In view of the developments in the following section, and for didactic reasons, we shall not bother to do this here.

The inner loop is only necessary to test a condition; in this particular case a condition on the value of the control variable *trec*. The control variable *erec* on the other hand is also needed for the construction of the result relation.

This state of affairs is more evident in the following solution for Example 4:

Solution 4.2.

```
.
.
{type definitions as in 4.1.}
.
.
var   employees, result:ereltype;
      timetable:treltype;
      some_trec_in_timetable:boolean;
begin .
      .
      .
      result := [ ];
      foreach erec in employees do
      begin some_trec_in_timetable := false;
        foreach trec in timetable do
          some_trec_in_timetable := some_trec_in_timetable or
          (erec.enr = trec.tenr) and (trec.tday = 'friday');
        if some_trec_in_timetable then result :+ [erec]
      end
end.
```

The additional exit from the inner loop is again omitted for didactic reasons.

Example 5. Find all employees who give no lectures.

This example contains a universal condition.

Solution 5.1.

```
.
.
{type definitions as in 4.1.}
.
.
var   employees, result:ereltype;
      timetable:treltype;
      all_trec_in_timetable:boolean;
begin .
      .
      .
      result := [ ];
      foreach erec in employees do
      begin all_trec_in_timetable := true;
        foreach trec in timetable do
          all_trec_in_timetable := all_trec_in_timetable and (erec.enr ≠ trec.tenr);
        if all_trec_in_timetable then result :+ [erec]
      end
end.
```

It seems natural for such problems, in which relations are used in testing one or all of their tuples against a certain condition, not to use repetition constructs but to introduce a special construct more related to this problem.

4. PREDICATES OVER RELATIONS

Examples 4 and 5 each require two nested loops. However, in both cases the outer loop and the inner loop have quite different significances in the logic of the program.

The outer loop over the range relation *employees* uses a logical condition evaluated by means of the inner loop—this is especially clear in Solutions 4.2 and 5.1. In both examples the condition governs the insertion of the value of the outer control variable into the result relation. Consequences will be discussed in Section 5.

The inner loop over the range relation *timetable* evaluates this condition. The condition itself is respectively an **or** connection (Example 4) or an **and** connection (Example 5) of Boolean terms involving tuples of the relation *timetable*, and is implemented by the foreach statement.

In the context of predicate logic these conditions are first-order predicates. This is more evident if one introduces range coupled quantifiers, i.e. quantifiers for which, together with the control variable (bound variable), the scope over which its value ranges must also be given (analogous to the foreach statement).

Predicates over relations will therefore be defined thus:

⟨*predicate*⟩ ::= ⟨*quantifier*⟩ ⟨*control record variable*⟩
in ⟨*range relation variable*⟩
(⟨*logical expression*⟩)
⟨*logical expression*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ ⟨*logical operator*⟩ ⟨*logical expression*⟩
⟨*quantifier*⟩ ::= **some** | **all**
⟨*logical operator*⟩ ::= **and** | **or**

Terms consist of components of the control variable, program variables, or constants, connected by the relational operators $=$, $<$, $>$, $\neq$, $\leq$, $\geq$; terms may also be predicates.

The implicitly declared control variable of the predicate is again of that record type in the range relation declaration.

4.1 The Existential Quantifier some

The predicate

some *rec* **in** *rel* (⟨*logical expression*⟩)

is true iff at least one value of the control variable *rec* makes the logical expression true. The values of the control variable are defined by the tuples of the range relation *rel*. In general the logical expression will contain apart from bound variables further program variables and constants.

The solution for Example 4 using the **some** quantifier now looks like this:

Solution 4.3.

```
.
.
{definitions and declarations as in 4.1.}
.
.
begin .
        .
        .
        result := [ ];
        foreach erec in employees do
          if some trec in timetable ((erec.enr = trec.tenr) and (trec.tday = 'friday'))
          then result :+ [erec]
end.
```

The tiresome problem of an additional exit from the loop has now disappeared for the user, and has been shifted on to the implementor responsible for the efficient implementation of predicates. An efficient implementation can above all exploit the fact that inside a predicate individual tuples of the range relation are not used as statement variables. The sequential processing of individual tuples can therefore be replaced by processing tuple sets in parallel. Comparing the three solutions developed for Example 4, we see that:

Solution 4.1, "**while not** aor(*timetable*) **do**," describes a sequential tuple-wise processing ordered by key values.

Solution 4.2, "**foreach** *trec* **in** *timetable* **do**," proceeds sequentially and tuple-wise but with no specific ordering.

Solution 4.3, "**some** *trec* **in** *timetable*," can be implemented by processing tuple sets in parallel.

If one considers that these tuple sets are so large in practice that they must be kept on secondary storage, these differences are highly significant.

4.2 The Universal Quantifier all

The predicate

all *rec* **in** *rel* (*(logical expression)*)

is true iff all values of the control variable *rec* make the logical expression true. The values of the control variable are defined by the tuples of the range relation *rel*.

Using the **all** quantifier, the solution to Example 5 can be written thus:

Solution 5.2.

```
.
.
{definitions and declarations as in 5.1.}
.
.
begin .
        :
        result := [ ];
```

```
        foreach erec in employees do
          if all trec in timetable (erec.enr ≠ trec.tenr)
          then result :+ [erec]
end.
```

4.3 Nested Quantifiers

Predicates may contain several quantifiers. For an appropriate example we introduce a third relation that describes courses by their attributes course number (unique), course level, and course title.

Now an extension to Example 5:

Example 6. Find those employees who give no lectures above the first year (course level 1).

This example requires both a universal and existential condition.

Solution 6.1.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
      crectype = record cnr, clevel:integer; cname:string end;
      creltype = relation ⟨cnr⟩ of crectype;
var   employees, result:ereltype;
      timetable:treltype;
      courses:creltype;
begin .
      .
      .
      result := [ ];
      foreach erec in employees do
        if all trec in timetable ((erec.enr ≠ trec.tenr) or
          some crec in courses ((trec.tcnr = crec.cnr) and (crec.clevel = 1)))
        then result :+ [erec]
end.
```

It becomes evident with examples like this that data elements in a database are typically not processed in isolation but together with their mutual logical connections: A particular tuple of the relation *employees* is only further processed when it stands in a particular relation to the tuples of *timetable* and furthermore there is a particular tuple in the relation *courses* such that ... .

The actual processing of the tuples is so far identical in all examples: Dependent upon a predicate they are inserted into the result relation or not. The solutions to a large class of problems in retrieving data from databases can be programmed in this way.

We are now ready to go one step further and develop a specific language construct for this standard programming problem based on the constructs introduced so far.

5. GENERALIZING THE RELATION CONSTRUCTOR

With the concepts developed so far the value of a relation variable can be altered by deleting, inserting, or modifying tuples and by assigning a relation-valued expression. These expressions are, however, up to now limited to single relation variables and to the elementary relation constructor introduced in Section 2.2. The elementary constructor is only capable of making a 1-tuple relation $[x]$ from a record variable x. This restriction has led, in all the examples handled so far, to the construction of new relations according to the following schema: tuple-wise access to the source relation, sequential processing of these tuples as records, and tuple-wise construction of the result relation. On the other hand, in the quantifiers and the logical expressions we already have the necessary prerequisites for a generalization of the relation constructor along the lines

$[x \textbf{ in } X\colon P(x, r, s, \ldots)]$

where x is the free variable which describes the result tuple, X is a range relation which holds the possible value tuples for x, and P is some logical expression which depends on the free variable, and possibly on further bound variables r, s, etc., and on constants.

5.1 Construction of Subrelations

A first step in generalizing the relation constructor leads to the definition

⟨*general relation constructor*⟩ ::= [**each** ⟨*control record variable*⟩
in ⟨*range relation variable*⟩:
⟨*logical expression*⟩]

The implicitly declared control variable is again of the same record type as that of the range relation. The logical expression has the usual logical and relational operators, and it may also contain quantifiers. As operands, the logical expression may contain components of the free control variable of the constructor and possibly of the bound variables of predicates, as well as program variables and constants.

With the aid of the general relation constructor, the solutions of the previous examples may be further simplified:

Solutions 3.3, 4.4, 5.3, 6.2.

```
.
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
      crectype = record cnr, clevel:integer; cname:string end;
      creltype = relation ⟨cnr⟩ of crectype;
var   employees, result3, result4, result5, result6:ereltype;
      timetable:treltype;
```

```
        courses:creltype;
begin .
        .
        .
        result3 := [each erec in employees:erec.estatus = 2];
        result4 := [each erec in employees:some trec in timetable ((erec.enr = trec.tenr) and
                        (trec.tday = 'friday'))];
        result5 := [each erec in employees:all trec in timetable (erec.enr ≠ trec.tenr)];
        result6 := [each erec in employees:all trec in timetable ((erec.enr ≠ trec.tenr) or
                        some crec in courses ((trec.tcnr = crec.cnr) and (crec.clevel = 1)))]
end.
```

In the proposed form the relation constructor can only create subrelations from one relation variable. The general case of the construction of relations with the aid of several free variables and arbitrary result tuples made up of their components will be treated in the next section.

5.2 The General Relation Constructor

In its most general form a relation constructor can be defined using several free variables. Its value is a relation defined by tuples whose components come from components of the free variables of the constructor, and maybe program variables and constants:

```
⟨general relation constructor⟩ ::= [each (⟨target component list⟩)
                                    for  ⟨control record variable list⟩
                                    in   ⟨range relation variable list⟩:
                                         ⟨logical expression⟩]
⟨target component list⟩ ::= ⟨target component⟩ | ⟨target component⟩; ⟨target component list⟩
⟨target component⟩ ::= ⟨control record component variable⟩ | ⟨variable⟩ | ⟨constant⟩ | empty
```

The correspondence between the control variables and the range variables is implied by their position in the respective lists. The previously defined relation constructor of Section 5.1 is a special case of this more general constructor.

A fourth relation will be introduced for the last example; this relation contains the publications of the employees, described by the title, the year of publication, and, to identify the associated employee, an employee number.

Example 7. For those employees who give lectures, find the names and the title and year of their publications.

Solution 7.1.

```
.
.
type  erectype = record enr, estatus:integer; ename:string end;
      ereltype = relation ⟨enr⟩ of erectype;
      prectype = record ptitle:string; pyear, penr:integer end;
      preltype = relation ⟨ptitle, penr⟩ of prectype;
      trectype = record tenr, tcnr, ttime:integer;
                        tday, troom:string end;
      treltype = relation ⟨tenr, tcnr, tday⟩ of trectype;
var   employees:ereltype;
      papers:preltype;
```

```
        timetable:treltype;
        result:relation ⟨rname, rtitle⟩ of
              record rname, rtitle:string; ryear:integer end;
begin .
        .
        .
        result := [each (erec.ename; prec.ptitle; prec.pyear)
                    for erec, prec in employees, papers:
                    (prec.penr = erec.enr) and
                    some trec in timetable (erec.enr = trec.tenr)]
end.
```

Specific problems with constructed relations, such as the definition of their keys and the possibility of checking keys at compilation time, will be treated elsewhere.

The relational constructor has certain advantages over the previous methods of solution:

It is a nonprocedural (very high level) language construct, in the sense that the user does not program a procedure which produces the result, but gives merely a declaration of certain properties of the result.

The notation is concise and keeps the important information textually together. This increases the readability for the user and facilitates a more efficient implementation.

The relation constructor may be given well-defined semantics in a similar way to predicate calculus.

The set of meaning preserving transformations of constructors is, in the context of predicate calculus, comprehensible. The freedom in the execution of the constructor thus obtained is very desirable for optimization.

At this point similarities to Codd's data sublanguage ALPHA [6] should be stressed. In particular the representation of queries in this calculus oriented language is closely akin to the relational constructor presented here. However, Codd considers relational calculus as an application of predicate calculus, whereas the relational constructor has been developed from, and integrated in, the concepts of a programming language.

6. SYSTEM IMPLEMENTATION AND FURTHER DEVELOPMENT

The various language constructs treated here are being incorporated into the PASCAL compiler for the DECsystem-10 [9] of the Institute for Informatics at Hamburg University. Apart from the necessary modifications to the compiler, a run-time system is being produced for the execution of the relational constructor and the predicates. This basically consists of an algorithm derived from that of Palermo [11] with additional optimization and supported by some advanced access methods. A first version of the system is expected to be available during the summer of 1977.

For the user, a relational database counts as an external variable and can—in analogy to external PASCAL files—be connected to a user program through a formal parameter in the program header. For the examples in this paper the program would

appear thus:

```
program dbuser (informatics77);
type  .
      .
      .
      ereltype = ... ;
      preltype = ... ;
      creltype = ... ;
      treltype = ... ;
var   informatics77:database employees:ereltype; papers:preltype;
                             courses:creltype; timetable:treltype end;
      result1, result2, ...:ereltype;
      result7:relation ⟨rname, rtitle⟩ of record ... end;
begin with informatics77 do
      ⋮
end.
```

In parallel to the implementation of the constructs so far developed, we are evaluating various further developments starting from standard PASCAL with respect to their applicability to database problems. With the aid of the **class** concept in [3], objects in a database can be declared in a form which hides the details of the database realization from the user, while improving data integrity and data security (see [15]). Problems of simultaneous access can be investigated using the **monitor** concept in [3].

7. SUMMARY

We have proposed three language constructs for use with data types of mode relation as extensions to a high level language, in particular to PASCAL. These constructs are: a repetition statement controlled by a relation; predicates, as extensions of Boolean expressions; and a general relation constructor, dependent on predicates. The language constructs have been developed stepwise from the most elementary operations on relations.

For the purpose of information retrieval from a relational database the relation constructor provides a solution which, in the context of a general purpose programming language, seems satisfactory.

For other questions, such as the problem of altering data consistently, or simultaneous processing of a database by several users, the proposals can serve as a framework for further investigations.

ACKNOWLEDGMENTS

I would like to thank H. Fischer, M. Jarke, D. Meyer, H.-H. Nagel, and W. Ullmer for their encouragement and the many critical and constructive comments.

REFERENCES

1. ALLMAN, E., STONEBRAKER, M., AND HELD, G. Embedding a relational data sublanguage in a general purpose programming language. SIGPLAN Notices (ACM) *8*, 2 (Feb. 1976), 25–35.

2. BOYCE, R.F., CHAMBERLIN, D.D., KING III, F.W., AND HAMMER, M.M. Specifying queries as relational expressions: The SQUARE data sublanguage. *Comm. ACM 18*, 11 (Nov. 1975), 621–628.
3. BRINCH HANSEN, P. The programming language Concurrent Pascal. *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 199–207.
4. CHAMBERLIN, D.D., AND BOYCE, R.F. SEQUEL: A structural English query language. Proc. ACM SIGMOD Workshop, Ann Arbor, Mich., May 1974, pp. 249–264.
5. CODD, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
6. CODD, E.F. A data base sublanguage founded on the relational calculus. Proc. ACM SIGFIDET Workshop, San Diego, Calif., Nov. 1971, pp. 35–68.
7. EARLEY, J. Relational level data structures for programming languages. *Acta Informatica 2*, 4 (Dec. 1973), 293–309.
8. FELDMAN, J.A., LOW, J.R., SWINEHEART, D.C., AND TAYLOR, R.H. Recent developments in SAIL—an ALGOL-based language for artificial intelligence. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 1193–1202.
9. FRIESLAND, G., GROSSE-LINDEMANN, C.-O., LORENZ, F.H., NAGEL, H.-H., AND STIRL, P.-J. A PASCAL compiler bootstrapped on a DEC-system-10. 3. GI-Fachtagung über Programmiersprachen, *Lecture Notes in Computer Science, Vol. 7*, B. Schlender and W. Frielinghaus, Eds., Springer-Verlag, Berlin, 1974, pp. 101–113.
10. LISKOV, B., AND ZILLES, S. Programming with abstract data types. SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50–59.
11. PALERMO, F.P. A data base search problem. Proc. 4th Comptr. and Inform. Sci. Symp., J.T. Tou, Ed., Plenum Press, New York, pp. 67–101.
12. ROVNER, P.D., AND FELDMAN, J.A. The LEAP language and data structure. Information Processing 68, North-Holland Pub. Co., Amsterdam, 1969, pp. 579–585.
13. SCHMID, H.A., AND SWENSON, J.R. On the semantics of the relational data model. Proc. ACM SIGMOD Conf., San Jose, Calif., May 1975, pp. 211–223.
14. SCHMIDT, J.W. Untersuchung einer Erweiterung von Pascal zu einer Datenbanksprache. Mitteilung Nr. 28, Inst. für Informatik., U. Hamburg, Hamburg, Germany, March 1976.
15. SCHMIDT, J.W. Type concepts for database definition: An investigation based on extensions to Pascal. Bericht Nr. 34, Inst. für Informatik., U. Hamburg, Hamburg, Germany, May 1977.
16. WIRTH, N. The programming language PASCAL. *Acta Informatica 1*, 1 (May 1971), 35–63.
17. ZLOOF, M.M. Query by example. Proc. AFIPS 1975 NCC, AFIPS Press, Montvale, N.J., pp. 431–438.

Received March 1977; revised May 1977