Relational Database Programming: Naming, Typing and Binding [*]

Joachim W. Schmidt Florian Matthes

University of Hamburg
Department of CS
Schlüterstraße 70
D-2000 Hamburg 13
e-mail: schmidt@rz.informatik.uni-hamburg.dbp.de

Abstract

The DBPL language orthogonally integrates sets and first-order predicates into a strongly and statically typed programming language, and the DBPL system supports the language with full database functionality including persistence, query optimization and transaction management. Modern language technology with its sound naming, typing and binding schemes applied to database concepts results in new insights into the relationship between types and schemas, expressions/iterators and queries, selectors and views, or functions and transactions.

1 Introduction and Overview

Data-intensive applications may be characterized by their needs to model and manipulate *heavily constrained* data that are *long-lived* and *shared* by a user community. These requirements result directly from the fact that databases serve as (partial) representations of some organizational unit or physical structure that exist in their own constraining context and on their own time-scale independent of any computer system. Due to the size of the target system and the level of detail by which it is represented, such representational data may become extremely voluminous – in current data-intensive applications up to $O(10^9)$ or even higher. In strong contrast to the need for global management of large amounts of *representational data*, data-intensive applications also have a strong demand to process small amounts of local *computational data* that implement individual states or state transitions.

In essence, it is that broad spectrum of demands – the difference in purpose, size, lifetime, availability etc. of data – and the need to cope with all these demands within a single conceptual framework that has to guide the design of integrated database programming languages.

Here we present an introduction to DBPL [?], an integrated database programming language that addresses the need for a uniform language framework for advanced database application programming. DBPL is a successor to Pascal/R [?] and extends Modula-2 into three dimensions:

- bulk data management through a data type *set* (relation);
- abstraction from bulk iteration through associative *access expressions*;
- *database modules* and *transactions* that abstract from persistence, sharing, concurrency control and recovery.

[*]This research was supported in part by the European Commission under ESPRIT BRA contract # 3070 (FIDE).

An essential guideline for the design of DBPL can be characterized by the slogan "power through orthogonality". Instead of designing a new language (with its own naming, binding and typing rules) from scratch, DBPL extends an existing language and puts particular emphasis on the interoperability of the database concepts with those already present in the programming language. Furthermore, DBPL aims at a uniform treatment of volatile and persistent data, large and small data collections and a uniform (static) compatibility check between the declaration and the utilization of each name.

2 Naming, Typing and Binding in a Relational Environment

The first step towards a better integration of database concepts into a language environment is to identify the basic database concepts and to rephrase them in terms of an appropriate vocabulary of programming concepts. The following paragraphs illustrate how DBPL captures the main principles of set- and predicate-oriented database systems using the well-understood notions of naming, typing and binding in procedural programming languages.

2.1 Names and Types

Names in DBPL are arbitrarily long sequences of upper and lowercase letters and digits starting with a letter.

DBPL is a *statically and strongly typed* language: every name is associated with a unique type that is determined at compile-time. The compiler uses this information to assure that all names for values, expressions or operations are only used in an appropriate context. The advantages of such a typing scheme are well known: programs are less liable to errors and there are no time-consuming dynamic type checks.

The type compatibility rules for composite types in DBPL are based on *name equivalence*, i.e., two composite objects have the same type if and only if they have been declared by using the same type name. This should be seen in contrast to the rule of *structural equivalence* where two objects are type compatible if their fully expanded definitions are the same.

DBPL inherits from Modula-2 all standard *built-in types* (`INTEGER`, `BOOLEAN`, `CHAR`, `REAL`, ...) and the mechanisms to declare *user-defined* types and *subrange types* thereof. *Strings* are treated as composite objects consisting of a sequence of characters. Furthermore, DBPL provides *array* and *record* type constructors.

A *relation type* specifies a structure consisting of elements of identical type, called the relation element type. The number of elements, called the cardinality of the relation, is not fixed. The declaration of the relation type specifies the relation element type and an ordered list of key components:

```
TYPE SupplierRel = RELATION Num OF SupplierRec;
     MadeFromRel = RELATION Num OF MadeFromRec;
     PartNumSet  = RELATION OF PartNum;
     Point2DSet  = RELATION OF ARRAY [1..2] OF REAL;
```

The relation key defines a list of components of the relation element type such that the relation always defines a function between its key and its element type. In other words, each key value uniquely determines (at most) one relation element. For example, the key constraint for a relation `Suppliers` of type `SupplierRel` can be expressed by the following predicate stating that the equality of the key component `Num` implies the equality of the relation elements:

```
ALL s1, s2 IN Suppliers (s1.Num = s2.Num) => (s1 = s2)
```

An empty key component list is a synonym for an enumeration of all components of the element type; in this case a relation is just a *set* of relation elements. The example above declares two relation types (with record elements), a set of natural numbers (`PartNumSetType`), and a set of points that are represented by their coordinates in the plane.

In order to create "containers" for values of these (relation) types, it is necessary to explicitly declare named *variables*.

```
VAR Suppliers    : SupplierRel;
    OldSuppliers : SupplierRel;
    MadeFrom     : MadeFromRel;
```

2.2 Scopes, Bindings and Lifetime

The above descriptions of the naming and typing rules have to be extended by rules defining the scope of names and the lifetime of objects denoted by these names. In order to do so we first have to introduce the concept of a *module*.

A module constitutes a sequence of name definitions and statements (to be discussed later in 2.3). A typical DBPL application consists of a multiplicity of modules. DBPL supports separate compilation, i.e. modules can be developed independently. A module can *import* names that are *exported* from other modules that include definitions for these names or that simply import these names from a third module. The compiler enforces the consistent use of names across module boundaries following the typing rules above.

The *scope* of a name `n` declared in a module `M` extends over the whole body of `M` and over all Module $\mathtt{M}_i$ importing `n`. Names have to be unique within a scope. Modules are in turn identified by names. In DBPL there is a single global scope for module names.

Using these rules it is straightforward to model the scoping rules of conventional relational database systems. A *database schema* is simply a module that declares and exports names for types of appropriate basic domains and declares and exports variables of relation types consisting of records with fields from the basic types (see Fig. 1). Similarly, an *application program* is a module that explicitly imports names from a database schema.

Since the import relationships between modules have to be declared statically, there is no possibility for name conflicts and ambiguities at runtime.

Modules can be defined as `DATABASE` modules. All variables declared within such a module are *persistent*, i.e. in contrast to other program variables their *lifetime* exceeds a single program execution. To be precise, the lifetime of a persistent variable is longer than that of any program importing it. Ordinary and persistent modules therefore allow the modelling of both, transient and persistent data objects.

Persistent variables are *shared* objects and can thus be accessed by several programs simultaneously. An access to a persistent variable must be part of the execution of a transaction (see section 3.3).

2.3 Expressions and Operations

For each type constructor of DBPL (record, array, relation), there is a *value constructor* to create objects of the composite type by enumerating its components:

```
v1:= SupplierRec{11, "John", important};
v2:= MadeFromSubRec{11, 100};
v3:= PartRec{3, "nut", base, 300.0, 20.3, 11};
```

In DBPL there are three kinds of *value selectors* for the selection of components of a structured value: Elements of an array are selected by an index value of their index type, enclosed

```
DATABASE DEFINITION MODULE SupplierPartDB;
TYPE
   PartNum        = [0..99999];
   SupplierNum    = [1000..9999];
   SupplierStatus = (unimportant, important, veryImportant);
   String         = ARRAY [0..29] OF CHAR;
   Dollar         = REAL;
   Kilo           = REAL;
   PartState      = (base, comp);
   SupplierRec    = RECORD
              Num     : SupplierNum;
              Name    : String;
              Status  : SupplierStatus;
            END;
   MadeFromSubRec = RECORD
              Num     : PartNum;
              Quantity: CARDINAL;
            END;
   MadeFromSubRel = RELATION Num OF MadeFromSubRec;
   PartRec = RECORD
              Num     : PartNum;
              Name    : String;
              CASE State : PartState OF
                base :
                  Cost        : Dollar;
                  Mass        : Kilo;
                  SuppliedBy  : SupplierNum;
              | comp :
                  MadeFrom    : MadeFromSubRel;
                  AssemblyCost: Dollar;
              END;
            END;
   SupplierRel = RELATION Num OF SupplierRec;
   PartRel = RELATION Num OF PartRec;
VAR
   Suppliers: SupplierRel;
   Parts    : PartRel;
END SupplierPartDB;
```

Figure 1: A typed relational database schema in DBPL

in square brackets (e.g., `vector[7]`); Fields of a record are selected by their field name (e.g., `supplier.Name`); Elements of a relation are selected by their key value, enclosed in square brackets (e.g., `Suppliers[7]`). Variable designators of DBPL therefore consist of a name followed by a path of value selectors.

In addition to these element-wise operations, DBPL provides specialized set-oriented *query expressions* for relation types. There are three kinds of query expressions, namely boolean expressions, selective and constructive expressions.

Quantified Expressions yield a boolean result (i.e. `TRUE` or `FALSE`) and may be nested:

```
SOME Supplier IN Suppliers (Supplier.Name = "John")
ALL Supplier IN Suppliers (Supplier.Status = important)
ALL Part IN Parts (Part.State <> base) OR
  SOME Supplier IN Suppliers (Part.SuppliedBy = Supplier.Num)
```

Selective Access Expressions are rules that select subrelations.

```
EACH Supplier IN Suppliers: Supplier.Status = important
```

selects all elements `Supplier` of the relation variable `Suppliers` that fulfil the selection predicate `Supplier.Status = important`.

A selective access expression within a relation constructor denotes a relation of all selected tuples:

```
SupplierRelType{EACH Supplier IN Suppliers: Supplier.Status = important}
```

Constructive Access Expressions are rules for the construction of relations based on the values of other relations:

```
NameRec{p.Name, s.Name} OF EACH p IN Parts, EACH s IN Suppliers:
     (p.State = comp) AND (p.SuppliedBy = s.Num)
```

where `NameRec` is a record of two strings defined as `RECORD Part, Supplier: String END`. The construction rule above defines how to derive the names of all base parts with their suppliers from the two stored relations `Parts` and `Suppliers`.

The application of a relation constructor to a constructive access expression creates a relation that contains the values of the target expression (preceding the keyword `OF`), evaluated for all combinations of the element variables (`p`, `s`) that fulfil the selection expression `(p.State = comp) AND (p.SuppliedBy =s.Num)`:

```
NameRel{{p.Name, s.Name} OF EACH p IN Parts, EACH s IN Suppliers:
       (p.State = comp) AND (p.SuppliedBy = s.Num)}
```

where the result relation type has to be defined as `TYPE NameRel = RELATION OF NameRec`

Note, access expressions do not denote relations; only in the context of a relation constructor `NameRel{...}` do they evaluate to a relation. Other contexts in which access expressions can be used are given below.

In addition to these (side-effect free) expressions, DBPL provides specialized *set operators* (`:=`, `:+`, `:-`, `:&`) for relation updates which assign, insert, delete, and update sets of relation elements:

```
Parts:= PartRel{};
Suppliers:- SupplierRel{EACH s IN Suppliers: s.Status=important}
```

The types of the expression and the variable on the left-hand side have to be compatible according to the rules of section 2.1. As illustrated by the examples above, the nesting of DBPL expressions captures the essence of relational query languages, namely to provide *iteration abstraction* by means of high-level set-oriented selection, construction and update mechanisms.

3 Interoperability for Database Programming

The above presentation of structures, expressions and statements of DBPL departs from the main stream of "standardized" query languages in order to achieve interoperability with strongly typed programming languages by means of uniform naming, typing and binding mechanisms.

The following sections illustrate the advantage, in terms of increased data manipulation, data description and data abstraction power, that is obtained by investing language technology into the relational data model

3.1 Computational Completeness

The main reason to couple a DBMS with an algorithmically complete programming language is to utilize the expressive power of the language environment for arbitrary complex operations on the data stored in the database. DBPL incorporates all data types, operations and control structures of the system programming language Modula-2, including recursive functions and procedures, higher-order functions and elaborately structured statements. DBPL is therefore an ideal environment for the implementation of complex database application programs.

It should be noted that database and programming language features of a database programming language should not simply reside side-by-side. On the contrary, one needs many interfaces to create synergy between these features. DBPL therefore allows the orthogonal combination of the concepts inherited from both worlds, for example:

- Relation types are allowed to appear in arbitrary contexts, i.e. not only as types for database variables, but also as types of local variables within procedures, or as types of value- or variable-parameters;
- Quantified expressions (see 2.3) can appear not only within query expressions but also in conditionals or as termination conditions of loops;
- Relation constructors can be used freely within expressions of arbitrary types. A relation constructor can contain function calls, arithmetic operations etc.

As it turns out, the use of a (generalized) relational calculus instead of a relational algebra facilitates such an approach, since predicates as boolean-valued expressions can be utilized in a broader range of contexts than pure relation-valued algebra expressions.

3.2 Type Completeness

In addition to the concept of computational completeness, DBPL adheres to the language design principle of *type completeness*, i.e. all type constructors of DBPL (relation, record, variant, array) have equal status within the language. It is therefore possible to apply these type constructors to arbitrary other types (e.g. to declare arrays of relations or relations of variant records containing relations of integers). Furthermore, values of these types can be used in expressions, assignments or as parameters in a uniform way.

Finally, DBPL provides *orthogonal persistence* for values of the base types and values constructed by means of the above type constructors. The persistent variables declared within a database module (see 2.2) are not limited to relation types. This makes it possible to declare, for example, a persistent boolean variable within a database module.

As illustrated in the database schema of Fig. 1, the concept of type completeness therefore naturally leads to a data model that supports the declaration of *complex objects* and non-first-normal-form relations thereby breaking the restrictions of the classical relational data model that is limited to relations of records with attributes from the basic domains.

3.3 Completeness of Abstraction Mechanisms

Up-to-date programming languages provide two important abstraction mechanisms to achieve localization of information in large software systems. *Process abstraction* allows programmers to abstract from the implementation of a subroutine and to perform complex operations simply through reference to its name with an appropriate list of actual parameters. *Type abstraction* allows programmers to abstract from the implementation of a data structure and to operate on it only via a well-defined interface, i.e. a set of operations defined for an abstract data type.

DBPL embodies both abstraction mechanisms by means of procedures that abstract over statements, functions that abstract over expressions and opaque types that abstract over type expressions. In addition, DBPL provides *selectors* that abstract over selective access expressions and *constructors* that abstract over constructive access expressions (see 2.3). These two abstractions capture the essence of updateable and non-updateable *views* in relational databases since selector applications can appear wherever a relation variable is expected and constructor applications can appear wherever a relation expression is expected.

The following selector named `ImportantSuppliers` defines an updateable view on the supplier relation, selecting those suppliers having `important` as their status (see also p. 5).

```
SELECTOR ImportantSuppliers: SupplierRel;
BEGIN  EACH S IN Suppliers: S.Status = important  END ImportantSuppliers;
```

The constructor `SuppliersForParts` (see also p. 5) names a non-updateable view that is derived from the base relations `Parts` and `Suppliers` and that contains pairs of parts and supplier names for all base parts with their respective suppliers.

```
CONSTRUCTOR SuppliersForParts: NameRel;
BEGIN
  NameRec{p.Name, s.Name} OF EACH p IN Parts, EACH s IN Suppliers:
      (p.State = comp) AND (p.SuppliedBy = s.Num)
END SuppliersForParts;
```

Without going into details it should be noted that naming (of statements, expressions etc.) naturally leads to the concept of *recursion*. The semantics of a recursive query expression in DBPL is not defined operationally (as it is common practice for procedures) but as a least fixed point of recursive set equation [?, ?]. Thereby constructors are at least as expressive as recursive DATALOG programs with stratification semantics.

Another important abstraction mechanism of DBPL is the *transaction* that allows database programmers to abstract from concurrency and recovery issues when accessing persistent and shared database variables. Transactions can be regarded as atomic with respect to their effects on the database. In particular, the implementation of DBPL guarantees that concurrent transactions will be executed in a serializable schedule.

```
TRANSACTION DeleteSuppliers(Suppliers: SupplierRel): BOOLEAN;
(* returns TRUE on success *)
BEGIN
  IF SOME bp IN Parts (bp.State = base) AND
    SOME s IN Suppliers (bp.SuppliedBy = s.Num) THEN
    RETURN FALSE; (* referential integrity violated *)
  ELSE
    SupplierRel:- Suppliers;
    RETURN TRUE
  END;
END DeleteSuppliers;
```

3.4 External Interfaces

Even in a sound, computationally-complete and self-contained language like DBPL, there are situations that demand communication with external components like user interface management systems (such as Motif or Open Look), network services, or simply with existing software components coded in other standard programming languages like Pascal, C or COBOL.

The challenge to integrate database programming languages into an *open system architecture* is mainly a technological problem and not a language design task. The current implementation of DBPL [?, ?] (running under VAX/VMS) takes the following approach to interoperability in a heterogeneous multi-language environment:

There is a special class of modules (called `FOREIGN DEFINITION MODULES`) that contain signatures of procedures coded outside the scope of the DBPL compiler. The use of these procedures is analogous (w.r.t. naming, type checking and binding) to ordinary DBPL procedure declarations.

The DBPL compiler generates for every DBPL module an object code file in the standard VAX-VMS linker format. The linker is therefore capable of linking DBPL modules with object code generated by virtually all VAX/VMS compilers.

Procedures and variables declared within individual DBPL modules can be used from other languages, as long as they do not contain relation, selector or constructor types.

The DBPL compiler generates debugger tables that can be interpreted by the standard VAX/VMS multi-language debugger. This makes it possible to set breakpoints as well as display and modify individual variables during the execution of compiled DBPL programs.

The need for an *ad-hoc query interface* is addressed in DBPL by means of a language-sensitive editor that was constructed using a powerful system for the generation of language-sensitive tools. The main idea is to simplify the task of an end-user that wants to query a database not by an ad-hoc restriction of the language to a subset of DBPL, but by providing *immediate* feedback (e.g. on undeclared names or type mismatches) during textual or form-oriented query input following the paradigm of *direct manipulation*.

4 Summary and Conclusion

Practical experience with the use of DBPL in lab classes demonstrates that students quite willingly accept the clear and improved interaction of a database model in form of "typed relational sets" and "declarative set expressions" with other concepts needed and found in procedural languages, like strong typing, fine-grained scoping and dynamic parameterization. However, while set-oriented expressions are readily used for the more standard data retrieval and manipulation tasks, experience also seems to indicate that there is a tendency to fall back to procedural solutions for more complex tasks, e.g. by embedding set expressions into iterators or recursive procedures.

The reason behind such user behavior may originate from the fact that simple query expressions can still be understood by refering back to an operational semantics in terms of set construction by loops, conditionals and assignments. However, this view becomes less appropriate for complex queries such as recursive ones that require a more abstract understanding in terms of model-theory and fixpoint semantics.

For designers and teachers it is definitely quite a challenge to reconcile this kind of "abstraction mismatch" inside their languages, a problem hard to overcome without leaving the framework of traditional computer languages (and definitly ruling out the "PL1 + SQL + "*" "-approach). We are convinced that only advanced language technology with higher-order and polymorphic functions, taxonomic typing systems and reflection will form the appropriate basis for next-generation database programming languages [?, ?, ?].