The Type System of DBPL[*][†]

Florian Matthes Joachim W. Schmidt

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
D-6000 Frankfurt, Germany
e-mail: schmidt@dbis.informatik.uni-frankfurt.dbp.de

Abstract

This paper presents the type system of the database programming language DBPL [SEM88] within the framework proposed in [ADG$^+$89].

1 Rationale of DBPL

The database programming language DBPL [SEM88] is a successor of Pascal/R [Sch77] and integrates a set- and predicate-oriented view of relational database modelling into the system programming language Modula-2 [Wir85]. Based on integrated programming language and database technology it offers a uniform and consistent framework for the implementation of database applications.

A central design issue of DBPL was the development of abstraction mechanisms for database application programming [MRS89]. DBPL's bulk data types are based on the notion of (nested) sets. First-order predicates are provided for set evaluation and program control. Particular emphasis had been put on the interaction between these extensions and the type system of Modula-2.

The rationale of the design of DBPL could be characterized by the slogan "power through orthogonality". Instead of designing a new language from scratch, we started with a modern system programming language and extended its simple, but powerful concepts (e.g., for modularization, procedural abstraction, and procedure types) *orthogonally* into three dimensions:

- Bulk data management through a data type *set* (relation).
- Abstraction from bulk iteration through *access expressions*.
- *Persistent modules* and *transactions* for sharing, recovery, and concurrency control.

Section 2 presents the type system of DBPL within the framework proposed in [ADG$^+$89]. To simplify the comparison of DBPL with other languages, the presentation strictly follows the structure defined in the framework paper, although this leads to some omissions of essential features of DBPL (e.g., control abstraction, transactions, views).

[*]This research was supported in part by the European Commission under ESPRIT contract # 892 (DAIDA).
[†]Appeared in *Proc. of the 2nd International Workshop on Database Programming Languages, Portland, Oregon,* June 1989.

2 The Type System of DBPL

2.1 The Nature of the Type System

DBPL is a fully upward compatible extension of Modula-2. Therefore, it inherits the elaborate type system of Modula-2 that is extended orthogonally by a set (relation) type and specific function types for subset definition and set evaluation (selector and constructor types).

The inclusion of these "database" concepts into the type system of a strongly typed programming language aims to overcome the traditional competence and impedance mismatch between programming languages and database management systems, since it provides

- a uniform treatment of volatile and persistent data,
- a uniform treatment of large quantities of objects with a simple structure and small quantities of objects with a complex structure, as well as
- a uniform (static) compatibility check between the declaration and the utilization of each value.

DBPL is strongly typed; all type checks are performed statically.

Low level implementation details (e.g., storage layout of records, existence of index structures) are completely hidden from the DBPL programmer who uses types only for data modelling.

DBPL utilizes the notion of type compatibility to associate statically checked *access rights* with views on large collections of shared objects. The definition of access restrictions gains importance in connection with the module concept of DBPL to define "safe" database interfaces for different clients of the same database.

2.2 Expressiveness

DBPL offers the traditional ordinal types (`INTEGER`, `CARDINAL`, `LONGINT`, `LONGCARD`, `CHAR`, `BOOLEAN`) and real numbers (`REAL`, `LONGREAL`).

DBPL supports application-specific extensions of the set of basic types by means of enumeration types.

Subrange types are derived from either basic or enumeration types. They impose additional restrictions on the range of values described by the subrange type. Within expressions, values of a subrange type are compatible with values of their base type. Assignments of values of a base type to variables of a compatible subrange type are checked at runtime.

Bit strings are declared as sets of ordinal types.

DBPL provides the following type constructors:

Records: The scope of the field labels is the record definition itself. They are also accessible as field designators referring to components of variables of that record type and within a `WITH` statement.

Variants: DBPL inherits variant records from Modula-2, which are more general than those of Pascal because both, nested and consecutive variant sections are supported. Access to fields within a variant section requires a run-time check of the tagfield.

Arrays: Arrays are composed of a fixed number of elements. These elements are (positionally) designated by indices, which are values of the index type. The latter must be an enumeration type, a subrange type, or the basic type `BOOLEAN` or `CHAR`.

Although arrays are of limited use for the implementation of dynamic data structures, they allow an efficient implementation of data structures that have a statically known maximum size.

Strings are arrays of characters with a fixed maximum length. The compatibility rules for strings are relaxed in order to allow assignments of strings of length n to variables of length $m \geq n$.

References: DBPL provides pointer types only to be upward compatible with Modula-2. Dynamic data structures and data structures with shared subobjects are implemented in DBPL by means of (keyed) sets. Therefore, references are no first class citizens in DBPL, e.g., they can not be made persistent.

Procedures: The type of a procedure is derived from its signature. In DBPL there are ordinary procedures and transaction procedures. The latter are marked by the symbol `TRANSACTION` instead of `PROCEDURE`[1]. Each of them can be further subdivided into proper procedures and function procedures. The latter are activated by a function designator within an expresion, whereas proper procedures are activated by a procedure call statement.

A function result must have a basic or a reference type. There are two kinds of parameters, namely value and variable parameters. If the parameter is an array, the specification of the actual index bounds can be omitted. The parameter is then said to be an *open array parameter*.

Opaque Types: DBPL provides opaque types, which hide the representation of a type declared within a definition module. The implementation module contains the concrete type, which must be a basic type or a reference type[2]. If the hidden representation changes but the interface remains the same, client modules will not need to be reprogrammed, or even recompiled.

Sets: Sets are the most important types for the representation of bulk collections of simple or composite objects in DBPL.

A set type specifies a structure consisting of elements of identical type, called the set element type. The number of elements, called the cardinality of the set, is not fixed. A set variable never contains more than one element for a given value of the set element type (i.e., duplicates are not allowed).

Keyed sets have the additional constraint that a keyed set variable never contains more than one element for a given key value. The declaration of a keyed set type therefore defines a list of components of the set element type that uniquely determines each set element [3].

Note that relations of the relational data model are just the specific case of keyed sets of flat records.

Selectors: In DBPL set evaluation is achieved through *access expressions*:

```
EACH e IN Set: p(e, .. ci ..)
```

is an expression that denotes exactly those elements in the set, `Set`, that fulfil the selection predicate `p`, just as the arithmetic expression $5 + 7$ denotes 12. The predicate `p` may be existentially and universally quantified.

```
SetType{EACH e IN Set: p(e, .. ci ..)}
```

is a set expression that denotes the set of (copies of) the elements selected by the enclosed access expression.

[1] In contrast to ordinary procedures, nested and recursive calls of transactions are inadmissable. Persistent variables can only be accessed during the execution of a transaction. Transactions can be regarded as atomic with respect to their effects on the database.

[2] In most DBPL applications, the hidden representation of an opaque type is a key value for a set variable; keys act as a "handle" for further informations stored within set elements since keys are the relational solution of the identification problem.

[3] The current implementation doesn't support key components that are part of variant sections of a record type or that are of type set.

```
FOR EACH e IN Set: p(e, .. ci ..) DO .. END
```

provides selected iteration over those elements denoted by the enclosed access expression.

Finally, access expressions can be named and parameterized:

```
SELECTOR sp ON (S: SetType) WITH (.. pi: Ti ..): SetType;
BEGIN
  EACH e IN S: p(e, .. pi ..)
END sp;
```

Like procedures in Modula-2, selectors in DBPL have types based on the selector signature. Due to the importance of bulk data access in database programming, DBPL has selector types (and selector variables) as first-class language constructs.

Selected relation variables can be compared with updateable database views.

Constructors extend the notion of selectors; they are based on relationally complete access expressions (not just one-variable selections) and provide (through recursion) the power of deductive databases [JLS85], [SL85], [SGLJ89].

DBPL is a *type-complete* language: composite types (records, variants, arrays, procedures, sets, selectors and constructors) can be constructed from elements of any (simple or composite) DBPL type. Furthermore, objects of any type can be used in any context (e.g., as parameters of procedures or right hand sides of assignments) in a uniform way. This orthogonality within a programming language enhances its modelling power and comprehensibility.

An active line of research is the extension of the relational model towards relations with relation-valued attributes (nested relations, non-first-normal-form relations, NF^2 relations). In DBPL the concept of type completeness “naturally” leads to a data model where nested relations are just a special case of nested type structures.

Recursive data structures and sharing of subobjects have to be modelled in DBPL by means of associative identifieres (key values of set types), i.e., recursive types are not allowed (see [Lam85] and [LMS84] for a presentation of a recursive data model, implemented on top of the “flat” relational data model of Pascal/R).

DBPL is a monomorphic language. The only polymorphic functions are the predefined operations on sets (test on membership, insertions, deletions, updates, quantified predicates, selective and constructive access expressions).

2.3 Types and Values

Values of all DBPL types are first class language objects: They can be named, denoted by expressions, stored in (persistent) variables and passed as value and variable parameters.

DBPL has a uniform mechanism to construct values of a composite type (array, record, variant, set) by an enumeration of its elements: the elements are enclosed in curly brackets and preceded by the name of the type that is to be constructed. Furthermore, it is possible to select elements of a composite value by a path consisting of array index expressions, record components, and key values for sets.

Access expressions (see section 2.2) can be seen as a generalization of these single-element construction and selection mechanisms.

In DBPL, it is possible to emulate (persistent) objects using modules and opaque types. However, DBPL is no object-oriented language, since it neither supports inheritance nor late binding.

The predefined equality operator, $=$, of DBPL is overloaded:

- It denotes structural equality if it is applied to operands of a basic or a string type.
- It is not applicable to operands of record, variant or array types.
- It denotes set equality based on the values of the key components if it is applied to operands of a set type.

DBPL neither has null values nor object types.

2.4 Relationships among Types

There are two reasons why DBPL uses *name equivalence* to define the type equivalence relationship:

1. The type system of Modula-2 is based on name equivalence.
2. Persistent variables are introduced by "normal" variable declarations within a database definition module. All access to database variables is defined *statically* based on the standard import and export mechanisms of Modula-2.

As there is no dynamic binding between database variables and the programs using them, all compatibility checks can be performed at compile and link-time.

The handling of abstract, "concrete" and unnamed types in DBPL is adopted without changes from Modula-2.

Modules in DBPL are not first-class values, but merely units of encapsulation, compilation and persistence: Any compilation unit of DBPL (interface, implementation or program module) can be declared as a `DATABASE` module. All variables declared at the outmost scope level of such a module will be *persistent*, i.e., in contrast to other program variables, their lifetime is longer than that of all programs importing this database module.

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. Transactions guarantee serializability (and failure recovery). The (user-defined) initialization of database variables is performed only once during database lifetime, namely before any access to the persistent variables has been made.

2.5 Types and Subtypes

DBPL only provides primitive subtyping rules for its base types (integers, reals) and subrange types.

DBPL neither has subtyping rules for record or object types as in Oberon [Wir87] and Modula-3 [CDG$^+$88], nor has it type inference rules for selector and constructor types (e.g., see Machiavelli [OBBT89] in a ML-style type system).

2.6 Database Issues

Persistence is an attribute of a module (see section 2.4). All variables declared within a database module are persistent and shared objects.

Changes to a (database) definition module only affect clients of that interface, which have to be recompiled. Changes to a (database) implementation module are invisible to its clients.

The conversion of persistent data of an “old” version of a database module into a representation suitable for a revised version of that module has to be done explicitely. This process is supported by specific tools, outside the scope of the language DBPL, and so is the definition of access paths etc.

2.7 Other Issues

In the DBPL project there is a strong commitment to implementability. A multi-user DBPL system under VAX-VMS is used at the university of Frankfurt since 1985 for courses on database programming. Based on this implementation, there are several prototype extensions for concurrency control (optimistic, pessimistic and mixed strategies), storage structures for complex objects, recursive queries and distribution. The construction of a distributed DBPL system is based on ISO/OSI communication standards and involved a re-implementation of the compiler to generate native code for the IBM-PC/AT.

3 Concluding Remarks

The most prominent feature of the DBPL type system is the type-complete integration of sets and first-order predicates into a stronly typed, monomorphic language with persistence as an orthogonal property.

This integration is motivated by the insight that the developers of data-intensive applications need language support for efficient, *set-oriented* access to both, private and shared collections of data.

Although Type systems are a crucial part in the design of database programming languages, there are other issues of particular importance for DBPLs:

- How to decompose large database application programs into manageable units, which can be developed and maintained independently?
- How to support concurrency, sharing and failure recovery by appropriate transaction models?
- What about data and function distribution and communication?

Issues like these influenced the design and implementation of DBPL [MRS89] [JGL+88] [MRS84] but are de-emphasized in this paper.

References

[ADG+89] A. Albano, A. Dearle, G. Ghelli, C. Martin, R. Morrison, R. Orsini, and D. Stemple. A framework for comparing type systems for database programming languages. In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, pages 203–212, June 1989.

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Menlo Park, California, 1988.

[JGL+88] W. Johannsen, L. Ge, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. Database application support in open systems: Language support and implementation. In *Proceedings of the IEEE Fourth International Conference on Data Engineering, Los Angeles, California*, February 1988.

[JLS85] M. Jarke, V. Linnemann, and J.W. Schmidt. Data constructors: On the integration of rules and relations. In *Proceedings of the Eleventh International Conference on Very Large Databases, Stockholm*, August 1985.

[Lam85] W. Lamersdorf. Recursive data models for non-conventional database applications. In *Proceedings of the IEEE First International Conference on Data Engineering*, April 1985.

[LMS84] W. Lamersdorf, G. Müller, and J.W. Schmidt. Language support for office modelling. In *Proceedings of the Tenth International Conference on Very Large Databases, Singapore, Taiwan*, pages 280–290, August 1984.

[MRS84] M. Mall, M. Reimer, and J.W. Schmidt. Data selection, sharing and access control in a relational scenario. In M.L. Brodie, J.L. Myopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*. Springer-Verlag, 1984.

[MRS89] F. Matthes, A. Rudloff, and J.W. Schmidt. Data- and rule-based database programming in DBPL. Esprit Project 892 WP/IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, March 1989.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, 1989.

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977. (Also appeared in ACM TODS, 2(3), September, 1977 and A. Wasserman (editor), IEEE Tutorial on Programming Language Design, and M. Stonebreaker (editor), Readings in Database Systems, Morgan Kaufmann Publishers, 1988 and 1993).

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.

[SGLJ89] J.W Schmidt, L. Ge, V. Linnemann, and M. Jarke. Integrated fact and rule management based on database technology. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

[SL85] J.W. Schmidt and V. Linnemann. Higher level relational objects. In *Proceedings of the Fourth British National Conference on Databases*. Cambridge University Press, July 1985.

[Wir85] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1985.

[Wir87] N. Wirth. From modula to oberon. Technical report, Department Informatik, ETH Zürich, Switzerland, 1987.