Language Technology for Post-Relational Data Systems [*][†]

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

Practice has proven that databases are the keystones for nearly all application systems with a wider functionality, utilization and availabilty. As a consequence, next generation database systems will have to provide their services with a degree of interoperability that has to be substantially improved over existing solutions. In this paper we argue that this objective can be achieved only through full exploitation of current developments in computer language technology.

We claim that the merits of modern computer languages' naming schemes, typing systems and binding mechanisms are vital also for database management systems if they want to improve the quality of their interaction with application languages and programming environments as well as their own functionality and extensibility.

This paper studies the technological basis of modern computer languages and presents DBPL as a working example of a highly interoperable database programming language that exploits such technology.

Requirements of future data systems are discussed by emphasizing the abstraction principles considered helpful for the adequate design and organization of data-intensive applications and for the modularization, localization and, finally, the implementation of data-based systems. We conclude by relating the potential of advanced language technology to such specific demands of the next generation of post-relational data systems.

Key words: database programming languages, query languages, interoperability, open systems, language design principles, naming schemes, type systems, binding mechanisms, DBPL.

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

[†]Appeared in: A. Blaser (Ed.), *Database Systems of the 90s*, Lecture Notes in Computer Science, Vol. 466, Springer, 1990

1 Introduction

There is no doubt that current research and development in computer languages will have a significant impact on the process of programming and program maintenance as well as on the architecture, functionality and, finally, the overall quality of software systems.

This quality improvement results only in part from the progress made in isolating and providing the basic building blocks for important classes of applications, e.g. the essentials of systems communication, recovery and persistence and by casting them into adequate *language primitives.*

More important, however, is the substantial progress made in understanding the *fundamental abstraction principles* that enable the user of a language to construct from a set of primitives a high-quality software solution, i.e. a well-structured system with a high degree of modularity, genericity, correctness and efficiency.

In the process of designing a computer language that aims for supporting such abstraction principles a closely related set of technical problems has to be solved that deals with the intimately related issues of naming, typing and binding. Although such *language technology* is already well developed and readily available its further extension and improvement is one of the main subjects of research in computer languages.

The purpose of this paper is to discuss the potential of language technology and to exploit it for the development of *advanced data systems.* The paper is organized as follows.

Section 2 presents some basic principles of modern computer language design and outlines the essentials of language technology. Furthermore, it emphasizes the role of naming, typing and binding for systems with high demands of interoperability. Section 3 discusses DBPL, a set- and predicate-oriented database programming language that exploits language technology to support abstraction principles and interoperability.

Requirements of post-relational data systems are addressed in section 4 with particular emphasis on future needs of data-intensive applications. The paper concentrates on demands for data modeling, interoperability and implementation modularity and extensibility. Finally, in section 5 we discuss the potential impact of advanced language technology for the design and development of post-relational data systems.

2 Language Technology and System Interoperability

Current discussions of new database models and systems tend to concentrate on the *expressiveness* of the particular query languages at hand. This view is justified in a traditional setting where query languages are the only means for interaction between end-users and databases systems.

Experience with existing database systems and their query languages (like Ingres, Oracle or DB2) demonstrates that the usability of a given language is often severely impaired by its inability to be integrated smoothly into a larger context. The classical example for such an integration is the embedding of a query language into a host language necessary to provide computational completeness, a prerequisite for application programming. But there are other situations that reveal the *limited support for interoperability* provided by existing database languages, such as the coupling with a customized user-interface or the exchange of data with standard software (e.g. spreadsheets like LOTUS 1-2-3).

There is a host of *ad-hoc solutions* to this problem of lack of interoperability ranging from simple data conversion utilities and query language cross-translators to sophisticated *fourth generation languages* (like Ingres 4GL, SQL/Forms or ADF). However, these solutions have to be considered as insufficient since they simply lift the interface problem to a higher level without addressing the need for intrinsic extensibility and orthogonality of concepts in order to attain the ultimate goal of interoperability. A typical example is a fourth generation language like Ingres 4GL that provides highly specialized support for convenient form management and menu interfaces but that still has to rely on programming language interfaces (without overall type security, without uniform naming mechanisms) to escape its own lack of generality and universality.

In section ?? we will present DBPL, a prototypical member of the family of integrated database programming languages. These languages aim to overcome the lack of interoperability and extensibility of todays databases by an integration of database functionality into a general-purpose language. In the long run, database programming languages are intended to replace the various language interfaces (query language, data definition language, physical schema description language, report definition language, etc.) by a single *linguistic framework* with consistent naming, typing and binding mechanisms that is expressive enough to unify this wide range of abstraction levels.

The following subsections review fundamental concepts of programming languages that are indispensable to attain such a high degree of safe interoperability and extensibility in open systems.

2.1 On the Design of Computer Languages

A computer language can be described by a set of *primitive concepts* (like values, variables, modules, semaphores, streams, databases, transactions ...) and a set of *abstraction mechanisms* (like nesting of expressions or statements, procedural abstraction, type abstraction, ...) that constrain and assist the composition of these primitives to larger semantic units.

The set of primitive concepts found in a particular language depends heavily on its application domain. Languages for mathematical applications might support complex numbers, arrays, vectors with their associated operations whereas database languages usually provide support for sets of entities and operations to access and manipulate these data sets.

Despite the superificial disparity of the various general-purpose and dedicated languages, there is a relatively small set of universal concepts that reappears one way or another in all of them. This paper concentrates mainly on the universal concepts of *naming, typing* and *binding* and on abstraction principles relevant for database programming.

The design of any computer language has to follow some *guiding principles.* There have to be some criteria for evaluating the merits of particular proposals. One such generally agreed upon criterion is the *orthogonality* of its concepts (e.g. see [?, ?]). This principle states that all language concepts (e.g. typing, procedural abstraction, assignment, ...) should be mutually independent of one another so that there is no interference when they are used in arbitrary combination. The principle of orthogonality thereby increases both the expressiveness of a language (more combinations are possible) and its understandability (there a fewer rules and exceptions to be learned).

Even in the restricted scope of relational query languages there is a general awareness of the need for orthogonality of language concepts in order to design expressive and understandable languages.[?] enumerates at length violations of the principle of orthogonality in the current SQL standard. In particular, it is shown that SQL does not fulfil the following minimum requirement for a good language design [?]: A language should provide, for each class of objects it supports, at least the following:

- a constructor function, i.e. a means for constructing an object of the class from literal (constant) values and/or variables of lower classes;
- a means for comparing two objects of the class;
- a means for assigning the value of one object in the class to another;
- a selector function, i.e. a means for extracting component objects of lower classes from an object of the given class;
- a general, recursively defined syntax for expressions that exploits to the full any closure properties the object class may possess.

The relevant classes of objects of a relational database system are tables, columns, rows und scalars. However, there are *no* means to construct, compare or assign individual rows in SQL. Furthermore, it is *not* possible to transform arbitrary nested algebra expressions directly into nested "table expressions" of SQL since there are limitations on the allowable nestings.

Sections ?? and ?? will introduce additional criteria to evaluate the quality of a language design.

2.2 Language Technology: Naming, Typing, Binding

A *name* is a token used to identify some entity that can be talked about by a computer language. A name is therefore said to be bound to an entity. Traditionally, a *binding* consists of a name-value pair [?] that can be augmented by a type [?] and an indication whether the value is constant or mutable [?]. Such augmentations serve the purpose to restrict the binding of a name to those entities that make sense in the context where the name is used – and to formally assure such semantic statements. Binding can take place at compile time (static binding) or during program execution (dynamic binding) [?].

A binding is always performed with reference to a particular environment. The *scope* of a name determines those parts of a program where it can be used (e.g. to define other bindings). In static scoping, the scope can be detected by a static analysis of program texts, whereas in dynamic scoping these parts may vary from one program execution to another.

Values that can be named include atomic values (like integers, characters, booleans etc.), composite values (like arrays, records, relations of other values) but also variables (mutable values or "containers" for values), functions, procedures or views in database systems.

A *type* can be associated with a binding to ensure that a name is only used in an appropriate context, e.g. the field selection `Supplier.Name` should only be admissable if the name `Supplier` is bound to a value that is a record with a `Name` attribute. Dynamic type checking occurs when the run time system executes code to ensure that the data is of the correct types. This typically occurs even in so-called statically checked languages in projections out of a union (e.g. field selection from variant records in Pascal).

Type systems are one of the most important and critical issues in language design (e.g. see [?]). The advantage gained by the use of type information is in a sense independent of the language it is embedded in: it adapts equally well to functional, imperative, object-oriented, and algebraic programming [?].

A type system serves different purposes:

- Types support the selection of suitable (i.e. space or time efficient) machine *representations* for data values. This was the main purpose of type information in the early programming languages (e.g. FORTRAN or COBOL). Especially in the presence of bulk data this aspect must not be underestimated.

- Type information guarantees that operations are only applied to correct arguments, i.e. they help to avoid errors like in the comparison `"A" > 7`. The widespread use of type information can thus be regarded as a partial specification of a program. In this sense, types can be seen as *specifications*, and typechecking as a limited form of program *verification*.

  Computer languages make heavy use of type information to increase programmer efficiency and productivity in the construction of large information systems. For example, most of the errors occurring in the (untyped) interaction between programs and databases using embedded DMLs can be detected even by the simplest type checker [?].

- Types are *descriptions*. A type declaration in a DBPL serves similar purposes as a schema description in a Data Description Language of a DBMS. A particular database state is thus an instance (a value) of that type. An important aspect of type systems is therefore the ability to *name* types (e.g. Age, Dollar, Sex) and re-use type descriptions.

- Powerful type systems support type *abstraction* to hide irrelevant program information or to protect private information from external access. Type abstraction thereby supports the ordered evolution of large information systems [?, ?].

- Finally, modern type systems support *genericity*, i.e. the possibility to write operations with uniform behavior on values of more than one data type. This kind of *polymorphism* is of particular interest in information systems: A formalized notion of similarity of data and algorithms enables the employment of re-usable and exchangeable solutions to repeating patterns of information processing requirements.

2.3 Language Technology for Interoperability

The designer of a computer language is faced with the problem of finding appropriate naming, typing and binding mechanisms that enable the interaction of as many computational units as possible and thus promote program genericity and system interoperability.

Traditionally, programming language designers advocate static binding, scoping and type checking, whereas database systems and operating systems rely heavily on dynamic mechanisms. For example, the existence of a `CREATE TABLE` and `DROP TABLE` command in SQL implies that it is impossible to have a static binding between relation names in queries and the stored database relation values. If a database system furthermore allows the change or removal of individual columns from a relation, it becomes impossible to perform static type checking within query expressions since the correctness of a field selection from a tuple of a relation (`SELECT Supplier.Name FROM Supplier`) can be invalidated at run time by a removal of the column `Name`.

It should be noted that there exists a tradeoff between the flexibility that can be achieved through dynamic binding, dynamic scoping and dynamic typing and the security and efficiency supported by static binding, scoping and typing.

Advances in computer language technology allow one to exceed the limits of the traditional static binding and typing mechanisms (e.g. by means of higher-order or polymorphic types) without sacrificing safety, efficiency and understandability attainable with static mechanisms. A presentation of some of these novel language concepts is the subject of section ??.

3 The Database Programming Language DBPL

Based on the description of the general language principles of naming, typing and binding given in the previous section we are now in a position to illustrate these principles refering to a specific language in a relational setting.

3.1 An Overview of DBPL

DBPL [?] is a successor to Pascal/R [?] and addresses the need for a uniform language framework for advanced database programming by extending a well-understood system programming language (namely Modula-2 [?]) by a small set of abstraction mechanisms needed for database application programming [?]. DBPL 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.

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 new database concepts with the given programming language concepts. In particular, DBPL aims to overcome the traditional competence and impedance mismatch [?] between programming languages and database management systems by providing

- 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 name.

Implementation details (e.g., storage layout of records, clustering of data, existence of secondary index structures, query evaluation strategies, concurrency and recovery mechanisms) are deliberately hidden from the DBPL programmer. A key idea in the design and implementation of DBPL is to let the runtime system choose appropriate implementation strategies based on high-level information extracted from the application program or its environment. As it turns out, the widespread use of *access expressions* (i.e. first-order logic abstractions of bulk data access) in typical DBPL programs facilitates such an approach.

3.2 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.

3.2.1 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 provides the following *built-in types*: `INTEGER`, `LONGINT`, `CARDINAL` (natural numbers), `LONGCARD`, `BOOLEAN`, `CHAR`, `REAL`, `LONGREAL`.

```
TYPE Dollar = REAL;
```

Values of these types are denoted by predefined names (e.g., `TRUE`) or literals (e.g., `0`, `3.2E-2`, `"A"`). It is furthermore possible to introduce user-defined names for these values:

```
CONST DollarToDMRatio = 2.71;
```

DBPL supports application-specific extensions of the set of basic types. An *enumeration type* is defined by a list of names:

```
TYPE SupplierStatusType = (unimportant, important, veryImportant);
```

This declaration defines an order on the values of the enumeration type:

```
    unimportant < important < veryImportant
```

Subrange types are derived from either basic or enumeration types. They impose additional restrictions on the range of values described by the subrange type:

```
TYPE PartNumType = [0..99999];
     ImportantStatusType = [important..veryImportant];
```

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.

Strings are treated as composite objects consisting of a sequence of characters. Their type is therefore `ARRAY [1..maxlength] OF CHAR`, where `maxlength` can vary from one string type to another.

```
TYPE LongString = ARRAY [0..999] OF CHAR;
     String = ARRAY [0..29] OF CHAR;
```

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`.

```
TYPE BonusTable = ARRAY SupplierStatusType OF Dollar;
```

A *record type* declaration defines a labeled cartesian product type. The scope of the label names is the record definition itself. These names are also accessible as field designators referring to components of variables of that record type (e.g. `supplier.Name`).

```
TYPE SupplierRecType = RECORD
                         Num     : SupplierNumType;
                         Name    : String;
                         Status  : SupplierStatusType;
                       END;
     MadeFromRecType = RECORD
                         Num     : PartNumType;
                         Quantity: CARDINAL;
                       END;
```

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 SupplierRelType = RELATION Num OF SupplierRecType;
     MadeFromRelType = RELATION Num OF MadeFromRecType;
     PartNumSetType  = RELATION OF PartNumType;
     Point2DSetType  = 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 `SupplierRel` of type `SupplierRelType` 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 SupplierRel (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 SupplierRel : SupplierRelType;
    OldSupplier : SupplierRelType;
    MadeFromRel : MadeFromRelType;
```

In this respect, DBPL is a contrast to traditional relational database systems, in which a `CREATE TABLE` command subsumes the declaration of an (anonymous) relation type and the declaration of a named relation variable.

It should be clear that the above set of naming and typing rules allows one to express (at least) the structures of the relational data model with a fine-grained control over the basic domains.

3.2.2 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 ??). 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.

```
DATABASE DEFINITION MODULE SupplierPartDB;
TYPE
   PartNumType        = [0..99999];
   SupplierNumType    = [1000..9999];
   SupplierStatusType = (unimportant, important, veryImportant);
   String             = ARRAY [0..29] OF CHAR;
   Dollar             = REAL;
   Kilo               = REAL;
   PartStateType      = (base, comp);
   SupplierRecType = RECORD
              Num     : SupplierNumType;
              Name    : String;
              Status  : SupplierStatusType;
            END;
   MadeFromSubRecType = RECORD
              Num     : PartNumType;
              Quantity: CARDINAL;
            END;
   MadeFromSubRelType = RELATION Num OF MadeFromSubRecType;
   PartRecType = RECORD
              Num     : PartNumType;
              Name    : String;
              CASE State : PartStateType OF
                base :
                  Cost        : Dollar;
                  Mass        : Kilo;
                  SuppliedBy  : SupplierNumType;
              | comp :
                  MadeFrom    : MadeFromSubRelType;
                  AssemblyCost: Dollar;
              END;
            END;
   SupplierRelType = RELATION Num OF SupplierRecType;
   PartRelType = RELATION Num OF PartRecType;
VAR
   SupplierRel: SupplierRelType;
   PartRel    : PartRelType;
END SupplierPartDB;
```

Figure 1: A typed relational database schema in DBPL

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. ??). 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 ??).

3.2.3 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:= SupplierRecType{11, "John", important};
v2:= MadeFromSubRecType{11, 100};
v3:= PartRecType{3, "nut", base, 300.0, 20.3, 11};

v4:= PartNumSet{1, 3, 77};
v5:= Point2DSetType{ {1.0,2.0}, {2.0,1.0} };
v6:= PartRelType{ {4, "bolt", base, 30.2, 11.4, 11} };
```

The right-hand sides of these assignments make use of value constructors for record types (`v1`, `v2`, `v3`) and relation types (`v4`, `v5`, `v6`). Note that the curly brackets `{}` are *overloaded*: depending on the type identifier preceding them they construct records, arrays, or relations. The assignments to variables `v5` and `v6` contain nested value constructors, e.g. `v6` is assigned a relation that contains a single record (of its element type `PartRecType`). Type identifiers for nested value constructors can be omitted.

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 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., `SupplierRel[7]`).

Variable designators of DBPL therefore consist of a name followed by a path of value selectors:

```
SupplierRel[7].Name:= "Peter";
PartRel[3].Mass := 13;
```

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 SupplierRel (Supplier.Name = "John")

ALL Supplier IN SupplierRel (Supplier.Status = important)

ALL Part IN PartRel (Part.State <> base) OR
  SOME Supplier IN SupplierRel
    (Part.SuppliedBy = Supplier.Num)
```

The comparison operators (=, <>, >=, <=, <) for relations are abbreviations for key-based quantified expressions, e.g. `SupplerRel1>=SupplierRel2` is equivalent to

```
ALL s1 IN SupplierRel1 SOME s2 IN SupplierRel2 (s1.key = s2.key)
```

The test for membership (`thisSupplier IN SupplierRel`) is equivalent to

```
SOME s IN SupplierRel (thisSupplier.key = s.key)
```

Selective Access Expressions are rules that select subrelations.

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

selects all elements `Supplier` of the relation variable `SupplierRel` 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 SupplierRel:
                  Supplier.Status = important}
```

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

```
NameRecType{p.Name, s.Name} OF
   EACH p IN PartRel, EACH s IN SupplierRel:
     (p.State = comp) AND (p.SuppliedBy = s.Num)
```

where `NameRecType` is a record of two strings defined as

```
TYPE NameRecType = 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 `PartRel` and `SupplierRel`.

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)`:

```
NameRelType{
  NameRecType{p.Name, s.Name} OF
     EACH p IN PartRel, EACH s IN SupplierRel:
       (p.State = comp) AND (p.SuppliedBy = s.Num)}
```

where the result relation type has to be defined as

```
TYPE NameRelType = RELATION OF NameRecType;
```

Note, access expressions do not denote relations; only in the context of a relation constructor `RelType{...}` 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:

```
PartRel:= PartRelType{};
SupplierRel:- SupplierRelType{EACH s IN SupplierRel: 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 ??. 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.

3.3 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

3.3.1 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 (`IF THEN ELSIF ELSE`, `CASE`, `WHILE`, `REPEAT`, `FOR`, `LOOP EXIT` etc.). 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 ??) 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.

Relation constructors provide an expressive mechanism to build relations from their elements. To support the inverse operation ("de-setting"), there is a strong need for *iterators*. The essence of an iterator is a selective access expression (see ??) denoting a subset of a relation variable to be iterated over. The body of an iteration is an arbitrary statement sequence that can read and update the value of the loop variable:

```
FOR EACH s IN SupplierRel: s.Status = important DO
  InOut.WriteString(s.Name);
  InOut.WriteLn;
END;
```

3.3.2 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 ??) 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.??, 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.

3.3.3 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 ??). 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. ??).

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

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

```
CONSTRUCTOR SuppliersForParts: NameRelType;
BEGIN
  NameRecType{p.Name, s.Name} OF
     EACH p IN PartRel, EACH s IN SupplierRel:
       (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: SupplierRelType): BOOLEAN;
(* returns TRUE on success *)
BEGIN
  IF SOME bp IN PartRel (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;
```

In section ?? we will relate the above abstraction mechanisms by embedding them into a larger context and discuss their relevance for database programming.

3.3.4 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 Requirements of Post-Relational Data Systems

The relational data model definitely is an important milestone in the process of understanding the needs of data-intensive applications and of developing a sound technological basis for database management systems. However, in the light of the above discussion, we see several severe limitations of the relational model, of which only a few are addressed by currently developed relational extensions [?, ?, ?, ?].

Shortcomings that are more difficult to repair (if at all) result from the exclusively value-oriented approach advocated by relational data modelers: "all information in the database is represented as values in tables" (see foreword by Codd of [?]). Realizing the limitations of such a semantically poor basis this served as a starting point for a more generalized view on data modeling [?, ?] and resulted in a far better understanding of the principles of *data abstraction* [?, ?, ?].

An important structural goal for many data-intensive applications is to extract a maximum of relevant information from individual application programs, localize it in a separate environment, and then make it available to a wider user community. However, by restricting itself to the localization of data, the relational model does not fully exploit the potential of *localization abstraction.*

Database management systems themselves turned out to be data-intensive applications in a similar sense that compilers of algorithmic languages are complex algorithmic programs. However, the interaction between two computational entities in an algorithmic program, say, entering an abstract representation of a conference deadline into a priority queue, differs substantially from passing a user-defined persistent date value to an optimizing join algorithm. Therefore, database applications are expected to have particular demands for *implementation abstraction.*

To draw a first conclusion, we argue for the development of DBPLs that support the increasing demands of data-intensive applications on the basis of a small set of built-in functional primitives that are well-supported by a body of abstraction principles which can be freely combined to ease and encourage the construction of functionally extensible data environments. We see this approach in contrast to the more or less "random" addition of packaged functionality to relational database systems (like subtables, triggers, graphical interfaces, ...).

In this section we want to motivate our approach by characterizing the essential subtasks that arise in the implementation, extension and maintenance of advanced data-intensive application systems. We will concentrate on the principles of

- data abstraction,
- localization abstraction,
- implementation abstraction,

that capture the essence of data modelling, software engineering and system coding. They are intended to provide a sufficiently general and well-understood basis for the database languages of the 90ies.

4.1 Data Abstraction

The data modeling task of database applications can be characterized by the need to describe large collections of entities by *heavily constrained data* with rather simple manipulation operations (create object, remove objects, update object attribute). Constraints can be classified into *static constraints* (like attribute constraints, object constraints, set constraints or dependency constraints) that have to hold in each database state and *dynamic constraints* as, for example, expressed by pre- and postconditions in transaction specifications [?, ?, ?].

Research and development in the area of conceptual modeling [?, ?] has isolated three basic *data abstraction principles* [1] that are sufficient to capture an important subclass of static constraints, namely the structural invariants of data objects [?, ?]:

Classification / Instantiation is a form of data abstraction in which a collection of objects is considered as a higher level object class. An object class is a precise characterization of all properties shared by each object within the collection. Classification represents an *instance-of* relationship between an object in the database and its object class that allows to identify, classify, and describe objects.

Aggregation / Decomposition is a form of data abstraction in which a relationship between component objects is considered as a higher level aggregate object. This is the *part-of* relationship.

Generalization / Specialization is a form of abstraction in which a relationship between object classes is considered as a higher level generic object. This is the *is-a* relationship.

Much of the power of these three abstraction principles comes from their *orthogonality*, i.e. the possibility to apply classification, aggregation and generalization to objects that

[1]The term *data abstraction* is sometimes used in the programming language literature to denote encapsulation (see section ??), a mechanism to attain *localization abstraction*.

already have been abstracted over, e.g. to define metaclasses (classes of classes [?]), nested aggregates (complex objects [?]) or generalized generic objects (multi-level inheritance hierarchies [?]).

Following [?] and [?] these data abstractions can furthermore guide the top-down design of database transactions by mapping operations on composite objects into composite operations (sequencing, case analysis, iteration) on component objects.

Such a *hierachical decomposition* should be supported by appropriate language mechanisms like set-oriented operations (selection, construction), tuple operations (field selection, record construction) and mechanisms to view specialized objects as generic objects or to project generic objects into specialized objects.

The classical relational data model only provides support for classification by means of relations and aggregation of atomic values within individual relation elements. Generalization, repeated aggregation, or repeated classification is beyond the scope of a relational DBMS. A first step to overcome these limitations was the inclusion of the concept of generalization into the set of abstraction mechanisms applicable to data values [?]. The conceptual modeling language TAXIS [?] extends this framework further by allowing one to aggregate, classify and generalize not only data values but also transactions, constraints and exceptions. Finally, object oriented languages rely heavily on the aggregation of data and operations (procedural attachment) as well as the possibility of specializing and generalizing these aggregates (inheritance) [?, ?, ?].

4.2 Localization Abstraction

Due to the evolutionary nature of database applications (arising from changes in the application domain itself or from add-on extensions to existing application programs), the initial implementation of a database program already has to take future incremental modifications and extensions into account.

The main quality criteria to be met by data-intensive applications are therefore:

Controlability of data access. Sharing and persistence in a database environment have to be controlled by appropriate authorization and protection mechanisms. They should provide a fine-grained control over the set of data objects and operations available to individual users and application programs.

Database systems traditionally make heavy use of dynamic views and dynamic sets of access rights (capabilities) whereas programming languages usually attain controlability by means of static scoping and encapsulation mechanisms [?].

Extensibility of data structures and application programs: Extensions range from simple additions of attributes, classes, relationships or inclusion of new queries and update transactions to the weakening or strengthening of database-wide integrity constraints. It should be possible to perform extensions in a way that does not disrupt the overall structure of existing applications and data structures and that minimizes the need for reverification of existing applications.

Reusability of data and operations. One of the main motivations for the use of (centralized or distributed) database systems is the desire to re-use and share information among larger user communities. By now it has become obvious that a mere sharing of "raw data" (like employee records) in some cases contradicts the goals of controlability and correctness as explained above. A first solution to this dilemma is the *encapsulation* of data objects where methods are the only means to interact with an object [?, ?, ?, ?, ?].

Correctness with respect to the (informal) specifications, especially in the presence of incremental changes. For example, integrity constraints on data objects (even those "coded into" application transactions) have to be respected by all update transactions, especially those implemented at a later point in time.

The typical approach to support these quality criteria during the evolution of applications in a language framwork is based on the principle of *localization abstraction*. The principle is based on the idea to localize definitions, i.e. to "factor out" repetitive information from applications and to replace them by a reference to a single, named declaration and to group related information in a local scope.

Classical examples of this principle are the declarations of named constants, types and operations in Pascal-like programming languages. But databases adhere to this principle also by factoring out objects and integrity constraints from multiple application programs and localizing them in a uniform, *centralized database schema*. By forcing application programs to access data objects only through controlled database system services, DBMS can guarantee a certain degree of correctness, controlability, extensibility and reusability.

It is clear that the traditional model of a database (essentially a single, global, unstructured store with a pool of integrity constraints) is an insufficient model to support incremental modifications for large database applications.

A first step towards a more uniform treatment of localization abstraction in a relational environment was presented in section **??** by an extension of the module concept of Modula-2. Section **??** presents a more detailed discussion of new binding and typing mechanisms that allow more flexible interfaces between database systems and their clients.

4.3 Implementation Abstraction

The previous two subsections were mainly concerned with modeling and software engineering aspects of database applications. Because of the size, value and longevity of the data to be dealt with, database applications also have a strong demand for *non-functional support* to be provided by a database system.

The implementation of a lookup table may illustrate this aspect. In a programming language setting, it is sufficient to implement a lookup table by means of a hash table or a search tree. The implementation of a lookup table is therefore adequately described by its operations (insert, remove, lookup) and its time and space requirements. In a database environment implementors of a persistent and shared lookup table have to take care of the storage management on disk, avoid interference between concurrent update operations and guarantee the integrity of the data structure in cases of program failures or system crashes.

These additional non-functional (or operational) requirements of database applications can be summarized as follows:

Persistence Management allows application programmers to abstract from low-level aspects of data representation, e.g. whether objects are held in main memory or on secondary store, how objects are identified (virtual addresses or tuple identifiers), when objects need to be transferred between multiple nodes in a network, etc. [?, ?, ?, ?].

Bulk Data Management allows one to abstract from details of the underlying data structures and access support structures, e.g. how objects are clustered on disc or in main memory, which index structures have to be maintained during update operations, which access paths can be utilized to speed up set-oriented query evaluation, etc. [?, ?].

Support for Atomicity is needed to define higher-level operations that abstract from the possibility of failure of the lower-level operations that are used by their implementation [?, ?].

Support for Concurrency Control is needed to abstract from possible interferences arising from concurrent access to shared data objects by multiple threads of control [?].

It should be noted that, for example, in a relational database management system these requirements are met by a small set of highly interrelated and generic concepts, namely set data structures, set-oriented query languages and the traditional (and highly overloaded) transaction model. The extreme complexity in the implementation of efficient, recoverable and concurrent access to persistent, shared data structures thereby remains completely hidden from database application programmers. The necessity of hiding in particular the heavy non-functional requirements from the clients of an operation demands language support for *implementation abstraction*.

Traditionally, implementation abstraction is achieved in a DBMS by using *generic* mechanisms that perform persistence management, bulk data management, query optimization, concurrency control and recovery for *all possible* data structures and user transactions. This genericity can only be attained based on a priori semantic knowledge of algebraic properties about sets, set operations and transactions.

However, as soon as advanced database languages permit substantial extensions or variations to be performed at the application level, there arises the need to open and extend the up to now fixed and pre-canned functionality of a DBMS at lower levels. Based on additional semantic knowledge of the application domain, a DBMS has to allow application-specific extensions to the persistence management, the bulk data management, the query optimizer etc.

Examples of such extensions are new base types, abstract data types, user-defined collection types, user-defined operations on complex objects, queries or functions as first-class language objects etc. Extensible database systems [?, ?, ?, ?, ?, ?, ?] usually provide well-defined specialized interfaces to lower levels (e.g. to the index manager or to the query evaluator) of the database system to "plug in" user-defined code. This DBMS kernel code written in a standard programming language like C or a specialized database implementation language like E [?, ?] can be called by DBMS extensions that implement, for example, application specific query languages.

5 The Potential of Advanced Language Technology

The descriptions of the abstraction mechanisms for database programming in the previous section remained deliberately abstract and general in order to be able to specialize these concepts to the wide range of languages proposed for the task of database programming. These proposals range from standard query languages, functional query languages, logic-based languages, relational database programming languages, persistent programming languages to object-oriented languages.

Without bias towards any of these solutions, one can predict that the current development of language technology will have a significant impact on the quality of the language interfaces for next-generation database systems. The following subsections give references to relevant literature and highlight the most promising achievements of todays language technology.

5.1 The Ubiquitous Data Abstraction

The three data abstractions (aggregation, classification and generalization as described in ??), rooted in the realms of knowledge representation, heavily influenced the design of new data models [?, ?, ?, ?, ?, ?] and are currently being "discovered" by type systems of database programming languages [?, ?].

The intuitive interpretation of a *type* describing a set of values [?] captures an essential aspect of the concept of classification [?, ?]. This static classification based on structural properties of data values and functions is furthermore theoretically well-defined [?, ?, ?, ?] and effectively enforceable by means of static or dynamic type checkers [?, ?].

Imperative and procedural programming languages have a long tradition in representing the concept of aggregation by means of records [?], structures or tuples as found for example in COBOL, Pascal, Ada, Modula-2 or C. On the other hand, functional languages and the theory of programming languages "neglected" labeled cartesian product types and expressed aggregation simply by means of pairs, triples, ... [?, ?, ?, ?].

However, motivated by the strong interest in object-oriented concepts, record types are currently receiving renewed interest not only from programming language implementors [?, ?, ?] but also from type theoreticians [?, ?, ?, ?]. These languages introduce a *subtyping* relationship between record types that allows one to utilize the powerful data abstraction principle generalization in a strongly typed framework.

To summarize, striking parallels exist between the data abstraction principles of conceptual modeling and fundamental concepts of modern type systems. A further analogy is the fact that type systems also obey the principle of *orthogonality*. A second-order λ-calculus [?], for example, allows one to name types, to pass types as parameters, to abstract over type expressions (yielding type operators) etc. Recent development in programming language design [?, ?] and type theory [?] even introduces a three level structure consisting not only of values described by means of types (the traditional two-level structure) but also of a third layer of *kinds* that classify types into simple types, type operators, etc.

Finally, there is now a solid understanding of the mechanism of *type inference* [?, ?, ?], that allows one to omit the (sometimes cumbersome) type annotations from expressions, statements, and functions and to write generic queries and transactions. The compiler is not only able to derive the missing type information from the operations performed on the data values, but it also detects the *most general* polymorphic type of every variable and parameter. Such a typing scheme is therefore also a big step towards the goal of language support for localization abstraction.

5.2 Advances in System Architecture and Implementation

Todays programming languages (like Ada [?], Modula-2 [?], Modula-3 [?], Eiffel [?] or Standard-ML [?, ?]) provide sophisticated naming, typing and parameterization mechanisms to define fine-grained and generic modules that are robust with respect to local changes and that can be bound to different (type-)parameters to be re-used in different contexts.

Localization abstraction therefore requires database as well as language support (see also [?]):

- data centralization as found in database systems including sharing, concurrent access and integrity maintenance;
- functional abstraction and parameterization [?, ?, ?] including higher-order functions and explicit or implicit type parameters [?, ?, ?];
- modularization and encapsulation including separate compilation [?], type abstraction [?, ?], and appropriate incremental binding mechanisms [?].

As explained in section ??, extensions of extensible database management systems are usually coded in a standard programming language like C or a specialized database implementation language.

We argue that the coding of such application-specific extensions should be understood also as part of the global database programming task. Furthermore, it should be clear that for this particular subtask there is a strong demand for the abstraction mechanisms presented in section ?? and ??, like encapsulation, procedural abstraction, strong and polymorphic typing, inheritance etc.

If one adopts this holistic point of view, DBMSs can be viewed essentially as collections of highly generic and extremely reusable functions and abstractions available in a database programming environment. These functions can not only be utilized by building application programs calling them, but they also provide "hooks" for user-specified extensions written in the same language yet at a lower abstraction level.

As soon as one attempts to describe the functionality of a fully-fledged database system in such a mono-linguistic framework one recognizes that there are several places where a DBMS needs to perform *magic*. The most striking example is the process of query optimization. Apparently, a DBMS must have the ability to "reason" about a given query expression prior to its execution, and, as a result of that reasoning, transform a query into a semantically equivalent one that can be executed more efficiently. Clustering of data objects and recovery are two other *magic* mechanisms that are part of the *implementation abstraction* provided by the DBMS.

We claim that the essence of DBMS implementation abstraction can be captured by three distinct language mechanisms:

Core Language Definition: There has to be a small set of basic language primitives (e.g. exception handling, persistence of simply structured data values, lock primitives, atomic operations). These primitives need to be built-in, since their implementation is beyond the scope of a high-level (typed) programming language.

Declaration Correspondence: This principle introduced by Landin [?] heavily influenced the design of several languages [?, ?, ?]. Its initial technical definition states that a programming language obeys the declaration correspondence principle if anything that can be declared inline can also be passed as a parameter. All subsequent additions to this principle try to capture the idea that a language should provide means to abstract from implementations in such a way that a client using this functionality is unable to distinguish it from built-in language constructs. Loosely speaking, this means that all language constructs and values have the same "civil rights".

Reflection is the ability of a system to support its own evolution. This may entail changes to the programs that describe the system and the types used by the system. Reynolds [?] showed (in terms of the λ-calculus) that it is impossible for a system to support itself in the same language it is written in. This means that systems have to resort to a lower level technology to express some part of the reflection. Examples for reflection are the `eval()` function in Lisp (untyped, dynamically bound), the *callable compiler* in the persistent store of PS-algol [?] and Napier [?] (statically typed, dynamically bound) and *generic forms* in ADAPTBL [?] (statically typed and statically bound).

Reflection seems to be a key mechanism in achieving the "magic" involved in the process of query optimization: a query optimizer could use reflection to access the code of a query expression submitted for evaluation, to obtain information about existing access paths within the system and to construct a new code for an equivalent query expression using these access paths.

6 Summary and Conclusion

Looking back on two decades of relational database history one observes contributions on three different levels.

The relational database *model* is, of course, a very important contribution and provides a solid basis for a number of advanced data modeling capabilities:

- reasonable data structures for a variety of commercial applications,
- concepts for very powerful queries and integrity constraints,
- an adequate approach to database updates.

As a second contribution, relational database *technology* has isolated and solved a variety of hard problems. It produced, however, only tailor-made implementations in which solutions of conceptually independent issues were bundled:

- only sets of records over limited domains can be defined and made persistent;
- optimized associative access is provided only for flat relational structures;
- the notion of transaction has highly specialized and overloaded semantics.

The development of relational database *languages* was a third contribution. However, it is now widely agreed that hardly any of them reached the then state-of-the-art in language design. Some of them even hindered the full exploitation of the relational model and led to various mismatches in database application programming [?].

Currently, much effort is invested in the development of new data systems which support typical database functionality and aim for going substantially beyond the relational approach. Examples are the object-oriented systems and the various forms of programming languages with bulk data structures and persistence.

What do we expect from such post-relational data systems? To put it succinctly, we expect three contributions: generalization of the model, unbundling of the technology and, in particular, improvement of the languages.

To draw the comparison with the relational contributions as outlined above, we anticipate post-relational data *models* to be liberated from essential relational restrictions in that:

- new typing schemes will be exploited and data models will become type-complete and orthogonal;
- bulk operators and assertions will be generalized;
- algorithmically complete constructs for data evaluation and manipulation will be provided.

From post-relational data *technology* we require the unbundling of its essential contributions and provision of its capabilities independent of one another:

- persistence will be given to an enlarged or even open set of data types and type constructors;
- efficient implementations of adequate abstraction mechanisms for bulk data access, modularization, interface definition and control etc. are expected;
- support is needed for an appropriate set of base mechanisms for non-functional requirements such as concurrency control, recovery and communication.

Finally, we expect that post-relational data *languages* will utilize modern language technology to bundle (or bind) the above capabilities according to the specific needs of the problem at hand:

- to allow for appropriate naming schemes for object identification and reference;

- to offer binding mechanisms that allow users to choose from a wide range of naming schemes, types, values and mutability restrictions which abstract and protect best an objects relevant properties;
- to capture additional object properties through extra capabilities for flexible scoping, sharing and lifetime definition or an appropriate recovery and migration status.

In conclusion, we are convinced that for the development of next-generation database technology there is a need for a *linguistic guideline*, and we see already a number of projects that are heavily influenced by language design principles (e.g. [?, ?, ?, ?]). On the other hand, next generation database language design (e.g. see [?, ?, ?]) will be driven and controlled by a "systems vision" with high but realistic expectations of the potential of database technology.

References

[A+89] M. Atkinson et al. The Object-Oriented Database System Manifesto. Technical Report 30-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, September 1989.

[AB87] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.

[AB88] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Rapports de Recherche 846, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, May 1988.

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), November 1983.

[ABM88] M.P. Atkinson, P. Buneman, and R. Morrison, editors. *Data Types and Persistence.* Topics in Information Systems. Springer-Verlag, 1988.

[ACC81] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7), July 1981.

[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 *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, pages 203–212, June 1989.

[AFS89] S. Abiteboul, P.C. Fischer, and H.J. Schek. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4), 1987.

[AK89] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *ACM-SIGMOD International Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[Alb83] A. Albano. Type Hierarchies and Semantic Data Models. In *ACM SIGPLAN '83: Symposium on Programming Langauge Issues in Software Systems*, pages 178–186, San Francisco, 1983.

[AM85] M.P. Atkinson and R. Morrison. First class persistent procedures. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[AM88] M.P. Atkinson and R. Morrison. Types, Bindings and Parameters in a Persistent Environment. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

[Ban88] F. Bancilhon. Object-Oriented Database Systems. In *Proc. of the ACM PODS Conf.*, Austin, March 1988.

[Bat86] D.S. Batory. GENESIS: A Project to Develop an Extensible Database Management System. In *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*, pages 207–208, September 1986.

[BB84] D. Batory and A. Buchmann. Molecular Objects, Abstract Data Types, and Data Models: A Framework. In *Proc. of the 10h International Conference on Very Large Data Bases*, 1984.

[BB87] F. Bancilhon and P. Buneman, editors. *Proceedings of the 1st Workshop on Database Programming Languages.* Altair, 1987.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[Bee88] C. Beeri. Data Models and Languages for Databases. Technical report, Dept. of Comp. Science, The Hebrew Univeristy, Jerusalem, Israel, 1988.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman, editors. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BHR82] P. Bunemann, J. Hirschberg, and D. Root. A Codasyl Interface to Pascal and Ada. In *Proc. 2nd British National Conference on Databases (BNCOD 2).* Cambridge University Press, 1982.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.

[BL87] P.A. Bernstein and D.B. Lomet. CASE Requirements for Extensible Database Systems. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[BMS84] M.L. Brodie, J. Myopoulos, and J.W. Schmidt, editors. *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, 1984.

[BMW84] A. Borgida, J. Mylopoulos, and H.K.T. Wong. Generalization / Specialization as a Basis for Software Specification. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Topics in Information Systems, pages 87–117. Springer-Verlag, 1984.

[BR84] M.L. Brodie and D. Ridjanovic. On the Design and Specification of Database Transactions. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Topics in Information Systems. Springer-Verlag, 1984.

[Bro84] M.L. Brodie. On the Development of Data Models. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Topics in Information Systems. Springer-Verlag, 1984.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can Object-Oriented Databases be Statically Typed? In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[C$^+$86] M. Carey et al. The Architecture of the EXODUS Extensible DBMS. In *Proc. International Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, Ca., September 1986.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Car88] L. Cardelli. Types for Data-Oriented Languages. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1988.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[CD87] M.J. Carey and D.J. DeWitt. An Overview of the EXODUS Project. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[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, Memlo Park, California, 1988.

[CDMB90] R. Connor, A. Dearle, R. Morrison, and F. Brown. Existentially Quantified Types as a Database Viewing Mechanism. In *Advances in Database Technology, EDBT '90*, volume 416 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1990.

[CL90] L. Cardelli and G. Longo. A semantic basis for Quest. Digital Systems Research Center Reports 55, DEC SRC Palo Alto, March 1990.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *ACM-SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Ma., June 1984.

[CM88] L. Cardelli and D. MacQueen. Persistence and Type Abstraction. In *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

[Cod79] E.F. Codd. Extending the Relational Database Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4), December 1979.

[CRZNM88] L.K. Chung, D. Rios-Zertuche, B. Nixon, and J. Mylopoulos. Process Management and Assertion Enforcement for a Semantic Data Model. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 469–487. Springer-Verlag, 1988.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dat84] C.J. Date. Some Principles of Good Language Design with Special Reference to the Design of Database Languages. *ACM SIGMOD Record*, 14(3):1–7, November 1984.

[Dat89] C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, second edition, 1989.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[DD79] A. Demers and J. Donahue. Revised Report on Russel. TR 79–389, Computer Science Department, Cornell University, 1979.

[Dea89] A. Dearle. Environments: a flexible binding mechanism to support system evolution. In *Proc. HICSS-22, Hawaii*, volume II, pages 46–55, January 1989.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DKA$^+$86] P. Dadam, K. Kuespert, Andersen, et al. A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies. In *ACM-SIGMOD International Conference on Management of Data*, pages 356–367, Washington, DC, 1986.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DV88] S. Danforth and P. Valduriez. The Data Model of FAD, a Database Programming Language, Rev. 1. Technical Report ACA-ST-059-88, MCC, June 1988.

[FH88] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, Workingham, England, 1988.

[GO87] D. Goldhirsch and J.A. Orenstein. Extensibility in the PROBE Database System. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[Gra81] J.N. Gray. The Transaction Concept: Virtues and Limitations. In *Proc. 10th VLDB Conference*, pages 144–154, Cannes, France, September 1981.

[Har84] D.M. Harland. *Polymorphic Programming Languages, Design and Implementation.* Ellis Horwood Limited, a division of John Wiley & Sons, 1984.

[HFLP89] L.M. Haas, J.C. Freytag, G.M. Lohmann, and H. Pirahesh. Extensible Query Processing in Starburst. In *ACM-SIGMOD International Conference on Management of Data*, pages 377–388, Portland, Oregon, 1989.

[HJ87] Schek H.-J. DASDB: A Kernel DBMS and Application Specific Layers. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):351–260, September 1987.

[HMT88] R. Harper, R. Milner, and M. Tofte. The Definition of Standard ML (Version 2). LFCS Report Series ECS-LFCS-88-62, Department of Computer Science, University of Edinburgh, August 1988.

[Hoa68] C.A.R. Hoare. Record Handling. In F. Genuys, editor, *Programming Languages*, pages 291–347. Academic Press, London, 1968.

[HSM89] R. Hull, D. Stemple, and R. Morrison, editors. *Proc. of the 2nd Workshop on Database Programming Languages.* Morgan Kaufmann publishers, 1989.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[I$^+$83] Ichbiah et al. The Programming Language Ada: Reference Manual. Technical Report MIL-STD-1815A-1983, ANSI, 1983.

[JLS85] M. Jarke, V. Linnemann, and J.W. Schmidt. Data Constructors: On the Integration of Rules and Relations. In *11th Intern. Conference on Very Large Data Bases, Stockholm*, August 1985.

[KL89] W. Kim and F.H. Lochowsky. *Object-Oriented Concepts, Databases and Applications.* ACM Press Books, 1989.

[KV87] S. Khoshafian and P. Valduriez. Sharing, Persistence, and Object Orientation: A Database Perspective. In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, pages 181–195, September 1987.

[L^+77] B. Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. of the 11th ACM Symp. on Operation System Principles, ACM SIGOPS*, pages 111–122, November 1987.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *ACM-SIGMOD International Conference on Management of Data*, pages 424–433, June 1988.

[MAD87] R. Morrison, M.P. Atkinson, and A. Dearle. Flexible Incremental Bindings in a Persistent Object Store. Persistent Programming Research Report 38, Univ. of St. Andrews, Dept. of Comp. Science, June 1987.

[Mat87] D. Matthews. Static and Dynamic Type Checking. In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, pages 43–52, September 1987.

[MB89] J. Mylopoulos and M.L. Brodie, editors. *Readings in artificial intelligence and databases.* Morgan Kaufmann publishers, 1989.

[MBW80] P.A. Mylopoulos, A. Bernstein, and H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[MD86] F. Manola and U. Dayal. PDM: An Object-oriented Data Model. In *Proc. Int. Workshop on Object-oriented Database Systems*, pages 18–25, September 1986.

[Mey88] B. Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Min88] J. Minker. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann publishers, 1988.

[MJAP86] D. Maier, Stein J., Otis A., and A. Purdy. Development of an Object-Oriented DBMS. In *Proc. Int. Conf. on OOPSLA*, Portland, Oregon, October 1986.

[Mos89] J.E.B. Moss. Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, pages 358–374, June 1989.

[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, West Germany, March 1989.

[Naq89] S.A. Naqvi. Stratification as a Design Principle in Logical Query Languages. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[Nik88] R.S. Nikhil. Functional Databases, Functional Languages. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

[NS87] P. Niebergall and J.W. Schmidt. Integrated DAIDA Environment, Part 2: DBPL-Use: A Tool for Language-Sensitive Programming. DAIDA Deliverable WP/IMP-2.c, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1987.

[OB88] A. Ohori and P. Buneman. Type Inference in a Database Programming Language. In *ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, 1988.

[OB89] A. Ohori and P. Buneman. Static Type Inference for Parametric Classes. In *Proc. of ACM OOPSLA Conference*, pages 445–456, New Orleans, L.A., 1989.

[PA86] P. Pistor and F. Andersen. Designing a Generalized NF^2 Model with a SQL-Type Language Interface. In *Proc. 12 Int. Conf. on Very Large Data Bases, Kyoto*, pages 278–288, August 1986.

[RC87] J. Richardson and M. Carey. Programming Constructs for Database System Implementation in EXODUS. In *ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, May 1987.

[Rey72] J.C. Reynolds. Definitional interpreters for higher order programming languages. In *Proc. ACM 25th National Conference*, volume 2, pages 717–740, Boston, 1972.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Colloquium sur la programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[Ric89] J.E. Richardson. E: A Persistent Systems Implementation Language. Technical Report 868, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[RT88a] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System For Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1988.

[RT88b] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, third edition, 1988.

[SAH87] M. Stonebraker, J. Anton, and M Hirohama. Extendability in POSTGRES. *Database Engineering, Special Issue on Extensible Database Systems*, 10(2), June 1987.

[SB83] J.W. Schmidt and M.L. Brodie, editors. *Relational Database Systems.* Springer-Verlag, 1983.

[SBK+88] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. DBPL Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1988.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proc. of 1st Int. Conf. on OOPSLA*, pages 9–16, Portland, Oregon, October 1986.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), September 1977.

[Seb89] R.W. Sebesta. *Concepts of Programming Languages.* Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Inc., 1989.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SM90] J.W. Schmidt and F Matthes. DBPL Language and System Manual. Esprit Project 892 MAP 2.3, Fachbereich Informatik, Universität Hamburg, West Germany, April 1990.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.

[SSB86] D. Stemple, T. Sheard, and B. Bunker. Abstract Data Types in Databases: Specification, Manipulation and Access. In *Proc. of the IEEE 2nd International Conference on Data Engineering*, pages 590–597, Los Angeles, California, February 1986.

[SSS90] L. Stemple, D. Fegaras, T. Sheard, and A. Socorro. Exceeding the Limits of Polymorphism in Database Programming Languages. In *Advances in Database Technology, EDBT '90*, volume 416 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, 1990.

[Sta88] R. Stansifer. Type Inference with Subtypes. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Str67] C. Strachey, editor. *Fundamental concepts in programming languages.* Oxford University Press, Oxford, 1967.

[SWBM89] J.W. Schmidt, I. Wetzel, A. Borgida, and J. Mylopoulos. Database Programming by Formal Refinement of Conceptual Designs. *IEEE – Data Engineering*, September 1989.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, 1985.

[Wan87] M. Wand. Complete Type Inference for Simple Objects. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.

[Wik87] A. Wikström. *Functional Programming using Standard ML.* Prentice Hall, 1987.

[Wir83] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

[WL81] B. Weihl and B. Liskov. Specification and Implementation of Resilient Atomic Data Types. *Proc. ACM SIGPLAN Symp. on Prog. Lang. Issues in Softw. Syst. ACM SIGPLAN Not.*, 16(5), May 1981.