Advances in Database Programming: On Concepts, Languages and Methodologies *

Joachim W. Schmidt Florian Matthes

Fachbereich Informatik
Universität Hamburg
Schlüterstraße 70
D-2000 Hamburg 13, FRG
e-mail: schmidt@rz.informatik.uni-hamburg.dbp.de

Abstract

The development of data-intensive applications, such as information systems or design applications in mechanical or software engineering, involves requirements modeling, design specification, and, last but not least, the implementation and maintenance of large database application programs. This paper concentrates on the last issue and presents the state-of-the-art in *Database Programming*.

Over the years, computer science has isolated a number of requirements essential for data modeling and has developed a variety of solutions to those demands. Examples include the various concepts to identify, structure and constrain data, as well as contributions to data visibility, lifetime, abstraction etc.

The paper first characterizes data-intensive applications by their specific demands in data modeling and argues for database programming languages and environments based on highly integrated and consistent sets of data modeling concepts. Next, a classification scheme for Database Programming Languages (DBPLs) is introduced and applied to several DBPLs. The paper concludes with initial results on database programming methodologies and environments that support the engineering of data-intensive applications.

Key words: Data-intensive applications, database programming languages, constraints, persistence, binding, type systems, polymorphism, abstraction mechanisms, methodologies.

1 Introduction and Overview

As a first cut, data-intensive applications may be characterized by their needs to specify and manipulate *heavily constrained* data that are *long-lived* and may be *shared* by a user community. These requirements are direct consequences of the fact that those data serve as (partial) representations of some organizational unit or physical structure that exist in their own context and on their own time-scale independent of any computer system. Due to the size of the target system and the level of detail by which it is represented, those representational data may become extremely voluminous – in current data-intensive applications up to $O(10^9)$ or even higher.

In strong contrast to the need for global management of large amounts of *representational data*, our class of applications also has a strong demand to process small amounts of local *computational data* that implement individual states or state transitions.

In essence, it is this broad spectrum of demands – the difference in purpose, size, lifetime, availability etc. of data – and the need to cope with all these demands within a single application that determines the daily life of a database programmer.

Another characteristic property of data-intensive applications is *evolution*, i.e. the frequent need to accommodate changing target systems, growing user communities, extended application functionality

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

etc. Frequently, such changes have to be done "on the fly", while minimizing the loss of data and preserving the relationship between and identity of existing data.

Section 2 presents and discusses in detail the basic data modeling concepts found in modern programming languages and database models. Based on those findings, the abstraction principles required by data-intensive applications are discussed and related in section 3. A classification scheme for Database Programming Languages is presented in section 4, and selected DBPLs are discussed and compared. The paper concludes by presenting in section 5 first experiences with an extended database programming methodology and an environment for the engineering of high-quality data-intensive applications.

2 Data Modeling Concepts

The purpose of this section is to present and discuss a set of concepts considered essential for high level data modeling. Many of these concepts, such as data identification and data structuring, have to solve similar problems in both applications, programming and database modeling, others have to meet different requirements resulting from substantial variations, e.g. in data lifetime or quantity.

It should be noted, however, that the designer of a particular DBPL has to balance between different design goals, including expressive power, generality, ease of notation, but also efficiency, implementability and understandability. Therefore, each DBPL represents a judicious combination of the language concepts enumerated in this section. Readers with particular interest in such language design issues are referred to the survey paper by Atkinson and Buneman [AB87], which presents several DBPLs in more detail.

Research and development in the area of DBPLs is based on results from – until now – separate research areas, such as

- Data Modeling (semantic data models, abstraction mechanisms, integrity constraints);
- Transaction Management (concurrency and synchronization models, protocols, implementations);
- Query Evaluation (join algorithms, access path selection, cost modeling);
- Memory Management (data placement and replacement, buffering, access structures);
- Data and Transaction Distribution (degrees of replication and transparency, communication protocols);
- Language Theory (type theory, semantics of programming languages);
- Compiler Technology (compilation techniques, data management, garbage collection).

The long-term goal of DBPL research is to establish an integrated technology based on a sound understanding of that multi-dimensional language design space. A first step in this direction is the definition of a common vocabulary as there are already some terms (like "sharing", "persistence" or "object-orientation") that mean different things to different people [KV87]. Therefore, the following subsections will also discuss some of the central notions used in the DB and PL literature.

2.1 Data Lifetime

The lifetime of data objects during program execution is well understood: Static variables exist during the whole program execution, variables local to a scope (e.g. block or procedure in C or Pascal) are automatically allocated on scope entry and de-allocated on scope exit, and heap variables have a lifetime controlled by the programmer. Scoping rules, variable initialization and garbage collection allow programmers to concentrate on data modeling issues, abstracting from low level memory management tasks [Seb89] .

However, when data are required to last longer than the duration of one program execution, the treatment is much less uniform as it is performed "outside" the language, using operating system or database system services with their own naming, scoping and binding mechanisms.

The notion of *persistence* means the ability of data values to exist for an arbitrary length of time – as long as necessary, as briefly as required [ABC*83, AS89].

Persistent Programming Languages (PPL), like PS-Algol, Napier or Amber, [AM88, AM87] try to extend the scope of programming languages to tasks usually accomplished by specialized tools in the programming environment (e.g. linking [AM85]). The rationale of these languages can be summarized as follows [AS89]:

Persistence Independence: Operations on data should be specified in a way that is independent of the persistence of that data. Similarly, the persistence of a value should be independent of the operations applied to it.

Persistence Data Type Orthogonality: The right to persist should be independent of the type of a data value.

Orthogonal Persistence Management: Persistence should not limit the computational model, the control structures or the strength of the type system.

The models of persistence found in implementations of programming languages can be roughly categorized as follows:

Workspaces: They provide the simplest form of persistence found in many interactive systems (like APL [Ive79] or LISP [Tei75]). At the end of a session the whole workspace (containing programs and variables) is saved and can be loaded at the beginning of a new session. There is no way of sharing at a finer granularity than through the entire workspace.

Modules (e.g. in DBPL and Adaplex, or external databases in Pascal/R): They generalize the notion of databases in DBMS by providing a local name space for constant, type and variable declarations. All variables declared (statically) within such a database module are persistent and can be accessed by several applications simultaneously. Programs can import and manipulate persistent variables like all other program variables.

Side-by-Side Address Spaces: In this model there is an ephemeral (main memory) and a persistent (disk) storage space. A value in main memory can be made persistent by an *extern* operation which writes it to persistent memory. Such a value can be recovered by a symmetrical *intern* operation. These operations preserve the structure of data, including sharing of subobjects and circularities [CM88]. There are subtle differences between various implementations of this model [BJW88, Car86a, Har88, AG89a]:

- Is the identity of an object between pairs of *intern* and *extern* operations preserved?
- Which objects can be made persistent (values, types, procedures)?
- What is the type of an object returned by an *intern* operation? What happens with values of abstract data types?
- Is there a possibility for concurrent access?

Single Address Space: In PS-Algol, Napier and OPAL there is a single global *persistent heap* [ACC81, DCBM89, MS87]. All objects (including procedures) accessible transitively from a global root pointer are persistent. This model combines the advantages of modules (no explicit operations for data movement) and side-by-side address spaces (dynamic creation of persistent objects of any type).

Implementations of persistent programming languages demonstrate that it is feasible to have efficient and shared access to such a homogeneous address space.

For a type-safe handling of persistent data, it is necessary to have persistent meta-data (type, i.e. schema information). The size and longevity of these data can limit the efficiency of traditional type matching algorithms and complicate the modification of existing software systems using persistent data [Zdo87].

As a partial solution to the problem, PPL provide *incremental binding* mechanisms [MAD87]. Traditionally, a binding consists of a name-value pair [Str67] that can be augmented by a type and an indication whether the value is constant or mutable. The binding can take place at compile time (static binding) or during program execution (dynamic binding).

One particular mechanism of dynamic binding is the use of *environments* [Dea89]: They are collections of bindings that can be extended or modified dynamically. Within the scope of an environment type checking is entirely static. Only on entry into the scope is a single dynamic type check necessary to verify that the environment contains type compatible bindings for all identifiers used within the scope. Using environments, it is possible to perform "local" changes to data types and procedures on "live" information systems without the need to recompile the whole system in order to guarantee its type consistency [AB87, DCBM89].

2.2 Data Typing

Type systems are one of the most important issues in DBPL design. 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 [Car89].

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).

  But also in DBPLs, 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*.

  Modern DBPLs make heavy use of type information to increase programmer efficiency and productivity in the construction of large information systems. 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 [BHR82].

- Types are *descriptions*. A type declaration in a DBPL serves similar purposes like 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 the ability to *name* types (e.g. Age, Dollar, Sex) and re-use type descriptions. Furthermore it is desirable that the notation of a type system is able to express more or less directly the well-known abstraction mechanisms of semantic data models [SS77, PM88] like

  - classification (classes, sets),
  - aggregation (objects, entities, records),
  - generalization (superclasses or appropriate subtyping rules), and
  - association (nested sets, multi-valued functions).

- Powerful type systems support type *abstraction* to hide irrelevant program information or to protect "private" information from external access.

  Abstraction supports the ordered evolution of large information systems [Car89, MRS89]. It should be noted, however, that the problem of encapsulation and abstraction conflicts with

some basic assumptions of existing database query languages, which view a database as a single "flat" name space [BCD89].

- 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.

  The basic mechanisms to attain genericity are explained in Section 2.6.

There are many detailed criteria for a comparison of type systems for DBPLs [ADG*89]. The following sections address only the most important language features and discuss their impact on database programming.

2.3 On Data Consistency

The moment when consistency constraints on data are checked can vary over a wide time interval. The earliest possible point in time depends, of course, on the kind of constraint, but also on the technology applied for constraint maintenance.

Database management systems perform checks at the latest possible point in time: at *execution time* of an operation or even later, at transaction commit time, when data become visible to other users. CASE Systems choose the earliest possible point in time: consistency is already enforeced at *application design time*. Language (i.e. type-) sensitive editors perform incremental semantic checks at *programming time*. Traditional language processors verify type constraints at *compilation time*.

In a *strongly* typed programming language all computations are checked for type errors. As pointed out in [BBO89] this definition depends on the notion of "type compatibility" supported by a particular language[1].

A language is *dynamically* type checked if the types of the arguments are checked as the operators are applied. In *static* type checking, type consistency can be *completely* determined before program execution. In "traditional" programming languages, static type checking takes place during compilation (eventually supported by the use of version keys during linkage).

In the presence of dynamic binding (see Section 2.1) things are more complicated as there is a need for some form of delayed type checking. However, the flexibility and adaptability of dynamic typing is counter-balanced by the possibility of run-time type errors during application execution and the need for a time-consuming type checking code [Mat87].

The policy of modern languages can be described as *eager type checking* [AB87]: The language supports as much static type checking as possible and there are special type constructors (e.g. `dynamic` [Car86a] [Car86b], `env` [Dea89] [DCBM89] [AM87], `ANY` [RLW85] [CDG*88], `Auto` [Car89]) that enable the programmer to specify the boundary between the realms of static and dynamic type checking.

There must be a mechanical way to verify that a program respects all its type constraints. Furthermore this process has to be transparent [Car89]: If a typecheck fails, the reason should be apparent[2].

There should be both a consistent theory supporting the type system (e.g. a typed λ-calculus [Rey74] or a theory of constructions [CH85]) and an efficient type check algorithm (e.g. [Mil78]).

The above discussion concentrates on the verification of the very limited set of *type constraints*. Loosely speaking, these constraints require that at any time during program execution the value of a variable is an element of a set that is defined without any reference to other variables (e.g. age $\in$ INT).

[1] For example, computations like $3 + (7 < 8)$ are valid in languages that do not discriminate numbers and boolean values.

[2] This situation should be contrasted with problems that arise in the area of general program verification.

To be able to enforce *general constraints* (invariants of a particular database application), some database programming languages provide additional language support:

- Application dependent constraints are commonly specified as first order *predicates* ranging over set variables (e.g. $\forall$ e $\in$ EMPLOYEES: salary(manager(e)) > salary(e)). In some languages it is possible to attach such a constraint to a particular type declaration (e.g. in Galileo [ACR85] and TAXIS [MW80]), whereas other languages allow the declaration of database wide constraints (e.g. ADABTPL [SSB86]).
- Most transformations of one consistent database state into another have to be performed in current languages by a sequence of operations. Therefore, it is sometimes necessary to have intermediate states that violate the global integrity constraints. *Transactions* allow the programmer to group such sequences into a single *atomar* operation with respect to integrity control. Transactions as language constructs can be found for example in the languages DBPL, TAXIS and Adaplex.
- It should be possible for application programs to "catch" violations of integrity constraints in order to resume execution in an ordered manner. Several DBPLs therefore support *exceptions* that propagate along the dynamic call chain of nested procedure calls, in search of an appropriate exception handler (e.g. Amber, Adaplex, Galileo).

A more general solution to the problem of integrity violations can be found in TAXIS [CRNM88]. The long-term behavior of processes is modeled in TAXIS by means of *scripts* that offer a Petri-net like graphical formalism. Transitions in scripts (i.e. transactions) can be *triggered* by various conditions: Intra-class predicates, temporal conditions (e.g. $now after 7) but also by the violation of a precondition, a postcondition, or of an attribute constraint.

Such a "data-driven" programming style is advocated by the designers of "active database systems" as a new paradigm for constructing time-constrained database applications [MD89].

There are several DBPLs that allow the restriction of the operations on data by the definition of *access constraints*: In Napier it is possible to define elements of arrays and fields of records as constants that must not be altered after their initialization. The same functionality is provided in TAXIS [MW80] by the attribute category *unchanging*. The language DBPL allows the association of access rights (a subset of the set {insert, delete, update, read, assign}) with views on relation variables. These access rights are verified at compile time [MRS89].

2.4 Data Identification

A main task in data modeling is the representation of objects and of relationships between them [Bro84]. Ullman [Ull87] distinguishes between *value-based* and *object-based* models.

In a value-based model (e.g. the relational data model), objects (tuples) are identified using associative identifiers (key values). This way objects and relationships can be represented uniformly. Many (procedural, functional and logic) DBPLs use sets as bulk data types and support (restricted subsets) of first-order logic for data selection, identification and data transformation (e.g. Pascal/R, DBPL, Modulex, Plain, Aldat, Adaptbl, LDL, FQL and LIFE).

Semantic data models and object-oriented languages emphasize the importance of *object identity*, i.e. "the ability to distinguish objects from one another regardless of their content, location, or addressability" [KC86].

The object-oriented language OPAL [CM84] and the initial design of FAD [DV88] postulated that *every* database object (even every string) has a globally unique identity that makes it distinguishable from all other objects at all times.

This should be seen in contrast to many entity-based languages (like Adaplex, Galileo, TAXIS) and modern object-oriented systems (like O_2 [LRV88] or OODAPLEX [Day89]) that distinguish between entities (resp. objects) and values (without identity). The rationale of this distinction is to provide

a "natural" interpretation of equality, "=", a predicate that is involved in virtually every database operation (selection, duplicate elimination, join) [AK89]: For objects it designates a test on "identity" (e.g. person1 = person2) whereas for values it designates "equality" (e.g. person1.name = 'Smith') [SAM89]. A similar solution can be found in DBPLs that support references (called mutable objects in functional programming languages): equality of references means "identity" of objects (e.g. see [OBB89]).

Systems that support object identity also offer two very different rules for the lifetime of an object. The predominant solution is the *reachability* rule: An object implicitly ceases to exist as soon as there are no other objects that reference it. Thus it is only possible to destroy references to an object (e.g. by overwriting with a new value), but there is no way to delete the object itself. This policy avoids dangling references and guarantees *referential integrity*. Other language models allow the explicit *destruction* of objects, thereby invalidating implicitly all references to it (e.g. the language E [RC87] in the EXODUS project [CD87]).

2.5 Data Structuring

Traditionally, database systems only supported a limited and fixed set of built-in types (integer, string, real, eventually date, time, currency etc.) and two powerful type constructors, namely some sort of records and sets (i.e. relations or owner-member or father-children relationships in the network and the hierarchical data model) [Dat81]. The spectrum of data structures was determined by the needs of the database applications, and the kinds of entities and relationships found in *commercial applications* had a big impact on early data models.

Programming languages, on the other hand, provide a variety of type constructors (e.g. array, record, reference) over a set of base types. In general purpose programming languages the device of data structures was influenced by the general notion of *algorithmic completeness*, but limited by the requirements of implementation efficiency. Therefore, most languages do not have built-in (generic) bulk data structures (e.g. relations, multisets, tables, index structures) to support dynamic data structures with a time-varying extent. However, modern languages enable the programmer to define new base types and type constructors using polymorphism and data abstraction (see Section 2.6 and 3).

Many database programming languages incorporate a data model in their type system. The first DBPLs adopted the relational data model and included a type constructor `RELATION` [Sch77] into a procedural programming language. Other languages are based on the functional data model [Shi81] or on object-oriented data models [Alb83, ACR85, MW80, BMW84].

Viewing data models as (specialized) type systems allows one to measure the initial definition of the relational data model [Cod70] against design principles for "good" type systems in programming languages. It also demonstrates how recent DBMS extend these definitions to overcome limitations of the traditional data model:

Type Completeness: In the relational model records must be "flat", i.e. their fields must contain values from the basic domains and relations have to contain records as elements. This should be seen in contrast to a *type-complete* data model where each type constructor can be applied to any base type or composite type. The regularity gained by the principle of type completeness not only enhances the modeling power of a type system [AB87], but also simplifies the understanding of the universe of discourse spanned by the typing system [AM88].

Several research prototypes of DBMS try to overcome limitations of the relational approach by including nested relations to support a wider range of applications, such as CAD/CAM and office information systems [BK85, HR83, JS82, SP82].

Another example of a violation of the type completeness principle in a programming language is the data type *file* found in Pascal [Wir71]: it is not allowed to declare files of files or to store pointers on external files to make dynamic data structures persistent.

User-Defined Data Types: In order to support so-called "non-standard applications", DBMS researchers recognized the need for non-standard, i.e. user-defined base types (e.g. points, line

segments) and type constructors (e.g. arrays, sequences, multisets).

An attractive approach is the inclusion of *abstract data types* (ADTs) into relational DBMS, viewing ADTs as a mechanism to define *new base types* (domains) [RM87, DW88, BB84].

However, the use of ADTs to define *new composite types* (e.g. lists or queues) finds much less attention in the research community. A notable exception is the ATLANT language [Zam88] that utilizes generic data types to construct user-defined data models. A major problem with the ADT approach is query optimization, which needs knowledge about the *semantics* of the operations defined for an ADT [HFLP89] (e.g. selectivity of filters, commutativity of binary operators) [MD86].

There is also some work on new bulk type constructors for database systems and database languages (e.g. lists, multisets [PA86], indexable lists [LRV88, LR89]).

Function Types: An interesting feature of modern languages (like ADA, Modula-2, ML, Prolog) is the ability to define higher-order functions, i.e. functions that take or return other functions. A comparable feature can be found in POSTGRES [RM87] that allows field values containing POSTQUEL procedures. These stored queries can be executed during the evaluation of other queries simulating *is-a* hierarchies, *part-of* hierarchies or non-1NF relations [DW88].

2.6 On Data Compatibility

Essentially, traditional programming languages (such as Pascal or C) take a "black-or-white" view in the sense that data are either of the same type and fully compatible or they are of a different type and, therefore, not compatible at all.

In contrast to those "mathematical" types are the "taxonomical" type systems [Car84] that take a more realistic view of data by emphasizing the commonalities between types rather than their differences.

Polymorphic languages are based on taxonomical type systems and allow a variable or a value to have more than one type. Generally speaking, this ability is important to *avoid overspecifications* of data structures or operations. Overly restrictive type constraints result from the need of *monomorphic languages* (like Pascal or C) to associate a single type with each variable, parameter and value.

On the other hand, with growing application systems there is the experience that for some operations data need to be discriminated by their type (e.g. data of type boolean have to be ruled out for integer multiplication), while for others a whole range of data is acceptable (e.g. push and pop operations on stacks). A more tolerant policy makes it possible to specify, for example, stacks that work for arbitrary element types or to implement an operation `celebrate birthday` that increments the age of a person, an employee or a student.

Following the established classification presented in [CW85], one can distinguish the following kinds of polymorphism:

Ad-hoc Polymorphism is found in many languages (e.g. Ada or Pascal) and can be further classified into:

Overloading: A variable or function name (e.g. `Add`) is used to denote different objects or operations. The context of the application of that name determines the actual object or operation to be used (e.g. `Add` to a stack or to a queue).

Coercion is a semantic operation that converts an argument to the type expected by a function. For example `sin(1)` is converted (by the compiler) into `sin(float(1))` because `sin` expects a floating point number as its argument.

These mechanisms can be understood as pure "syntactic sugar" as they do not add any expressive power to a monomorphic type system.

Universal Polymorphism: On the implementation level a universally polymorphic function can be characterized by its ability to execute the *same* code for arguments of any admissable type.

Universal polymorphism is often considered *true* polymorphism, whereas ad-hoc polymorphism is some kind of *apparent* polymorphism [CW85]. There are two forms of universal polymorphism:

Parametric Polymorphism: A polymorphic function (sometimes called *generic* function) has an explicit or implicit type parameter which determines the type of argument for each application of that function. A typical example is a function `reverse` that reverses lists with arbitrary element types.

In some languages (e.g. Napier [DCBM89]), the programmer has to supply a type parameter for each application of the function (`reverse[int](ilist), reverse[real](rlist)`) and the compiler verifies that `ilist` is a list of integers and `rlist` is a list of reals. Other languages (e.g. ML [HMT88], Machiavelli [OBB89]) provide special *type inference* rules, i.e. the compiler infers the (implicit) type parameter from the types of the arguments. In the presence of type inference there is no need at all to declare the (polymorphic) type of the argument of `reverse`: the type checker is able to deduce the "most general" polymorphic type of the function argument from the operations applied to the argument within the function body [Mil78].

Sometimes it is necessary to impose additional constraints on the type parameters of generic functions: a function to sort sequences with elements of type α only works for types α for which a comparison function is defined (*bounded parametric polymorphism* [CW85]).

The main use of generic functions in data-intensive applications is the definition of user-defined type constructors (e.g. `tree of` α) with their associated operations (e.g. `insert, delete`). Such polymorphic functions could also be used as vehicles to provide the user-defined semantics for an extensible DBMS (`relation of` α`, index from` α `to` β).

Inclusion Polymorphism: The notion of inclusion polymorphism first appeared in Simula67 [DN66] and can be found in several modern programming languages (e.g. Amber [Car86a], Modula-3 [CDG*88], Oberon [Wir87], Eiffel [Mey88], Quest [Car89]). There is a *subtype* relation between types, similar to the relation of containment between sets. If A is a subtype of B, then any object of A is also an object of B.

The most common subtype relationship is between record types: A record type A (e.g. Student) that has at least all fields of a record type B (e.g. Person) is a subtype of B. In Machiavelli [OBB89] there is also a"lifted" version of this subtyping rule, stating that a set of (relation of) A is a subtype of a set of B iff A is a subtype of B.

The subsequent section will illustrate the use of inclusion polymorphism to directly represent generalization (specialization) hierarchies within the type system of a DBPL.

It should be noted that it is possible to simulate (bounded and unbounded) parametric polymorphism using inclusion polymorphism, but not vice versa, i.e. inclusion polymorphism is more expressive than parametric polymorphism [Mey86]. However, many languages retain both forms of polymorphism supporting a more succinct notation (e.g. Eiffel [Mey88], Trellis/Owl [SCB*86]).

In [Car88], Cardelli proposes a three level structure as a framework for languages with *type operators* for (type-safe) operations on type structures. Intuitively, level 0 consists of the set of all possible values, level 1 contains types (as sets of values) and type operators (mapping types to other types), and level 2 is the level of *kinds*, which are "types" of types and type-operators. As demonstrated in Quest [Car89], such a sophisticated type system can help in adding power and regularity to data-oriented languages.

3 Abstraction Mechanisms for Data-Intensive Applications

Advances in computerized applications are closely correlated with a better understanding and increased utilization of appropriate *abstraction principles*. Abstraction aims at decomposing large problems into smaller tractable ones and at concentrating on selected issues of current interest while

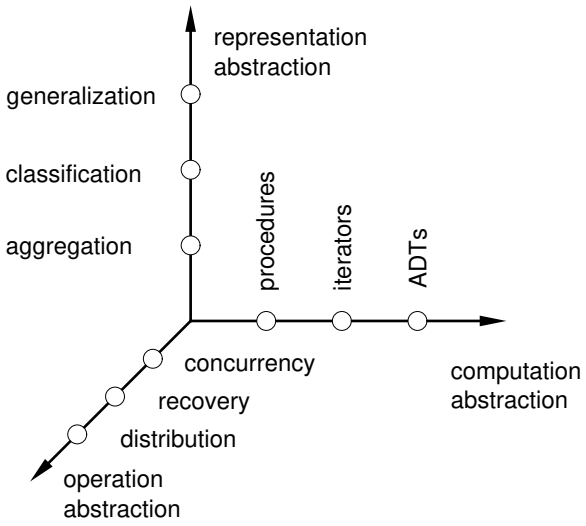

Figure 1: Abstraction Space for Data-Intensive Applications

postponing others considered as irrelevant representational, computational or operational details. Thus, abstraction allows one to identify and delegate subproblems that can be solved independently and can be combined at a higher level to a solution of the original problem.

Following our previous discussion, data-intensive applications require the joint application and close cooperation of abstraction principles from at least two areas:

- representation abstraction as provided by data models;
- computation abstraction as supported by programming languages.

Furthermore, data-intensive applications have a strong demand for certain operational services such as concurrency or distribution management and recovery from failure which, in a traditional setting, are provided through various operating system services.

This abstraction space for data-intensive applications (Fig. 1) will be presented and discussed in the following subsections.

3.1 Abstraction in Data Modeling

The abstraction mechanisms in up-to-date data models are [SS77]

Classification is a form of abstraction in which a collection of objects is considered as a single higher level object, frequently called *class*. Classification is used in conceptual modeling [BMS84] to identify, classify, and describe objects in terms of object classes.

In many DBPLs, classification is represented by two separate mechanisms: A *type* specifies the (time-invariant) common structural properties of objects of a given object class, and a *variable* (e.g. of a set type) serves as a "container" for the (time-varying) extent of that class. The *class* concept of object-oriented languages subsumes both mechanisms.

Aggregation is a form of abstraction in which a relationship between component objects is considered as a higher level aggregate object.

This *part-of* relationship is represented in DBPLs either explicitly by means of record constructors or implicitly in a class definition.

Generalization is a form of abstraction in which a relationship between class objects is considered as a higher level generic object. The abstraction of the three class objects `student`, `programmer` and `lecturer` to the single generalized object `person` may serve as an example.

In semantic data models and object-oriented DBPLs this *is-a* relationship is captured by inheritance hierarchies. Subclasses inherit all attributes of their superclass(es) but they can also add new attributes (e.g. Galileo, Adaplex). TAXIS has separate inheritance hierarchies for data, transaction and exception classes [MBW80, CRNM88]. OPAL has a single inheritance hierarchy, as all operations on data objects have to be attached as *methods* to their class definition. Thereby it is possible for subclasses to selectively *override* method specifications inherited from their superclass.

OPAL and the implementation of Galileo only support *single inheritance*, i.e. a category object can only be generalized to a single generic object. Amber is an example of a language that has *multiple inheritance*. Another unique feature of Amber is the fact that the *is-a* relationship between two object types is inferred by the compiler and needs not to be declared explicitly [Car84].

From the previous discussions it should be clear that the notion of inheritance is close to the (more abstract and technical) definition of inclusion polymorphism (see Section 2.6).

Following [Dij76] and [BR84] these *data abstractions* can guide the top-down design of database transactions by mapping operations on higher-level objects into composite operations (sequencing, case analysis, iteration) on lower-level objects.

3.2 Abstraction in Computation Modeling

The notion of *abstraction* has a somewhat different flavor in computation abstraction as supported by modern programming languages. Here abstraction emphasizes the advantages of *information hiding*, encapsulation, and protection. Liskov and Guttag [LG86] distinguish three kinds of abstractions:

Procedural Abstraction has two aspects: *Abstraction by parameterization* allows one to represent a potentially infinite set of different computations with a single program text that is an abstraction of all of them. *Abstraction by specification* allows one to abstract from the details of a computation given by the body of a procedure. This is accomplished by associating with each procedure a specification, usually in form of first-order predicates for *pre- and postconditions*, or input/output relationship [Hoa69, Dij76, Heh84].

The notion "procedural" abstraction is somewhat misleading as parameterization and specification are fundamental concepts of functional and logic-based languages too.

Data Abstraction is achieved by bundling a set of objects and a set of operations that characterize the behavior of these objects. The objects can only be created, modified or inspected using a given set of operations (constructors, mutators and observers). *Abstract data types* hide irrelevant information of the type representation and protect internal invariants from external access.

Iteration Abstraction abstracts from details of loops iterating over elements of composite objects (e.g. iteration order, or effects of updates during iteration). For example, CLU [LAB*89] and Trellis [SCB*86] provide built-in language constructs to support user-defined iterators (e.g. over sets, lists, trees).

Many DBPLs (e.g. Pascal/R, DBPL, Adaplex, OPAL) not only have specialized loop statements that apply to variables of their built-in bulk data types, but also support quantified expressions for set selection and set construction. Other approaches to iteration abstraction can be found in Aldat [Mer77] or Machiavelli [OBB89] both of which generalize the operators of the relational algebra (selection, projection, join).

A main advantage of built-in iterators, set-valued expressions or set-oriented operators is the ability to specify intentionally the set of objects that are to be processed and to delegate the task of choosing an "optimal" execution strategy to the compiler or the run-time system. Based on the knowledge of existing access paths, they can speed up the execution of bulk data operations considerably.

This "declarative" style of programming is further emphasized in logic-based languages (e.g. LDL [NT89], FAD [BBKV87]). In these languages, the programmer utilizes (mutually recursive) rules to select and construct sets of objects. The semantics (e.g. fixed point semantics) of these languages allow one to abstract from iteration sequences as well as from termination conditions that arise in recursive query processing.

To integrate procedural elements like updates or sequencing of operations into such a declarative formalism, LDL uses the notion of *stratification* [Naq89]. DBPL, a procedural database programming language, takes an inverse approach: fixed point semantics are only assigned to a distinguished set of "functional" query expressions (called constructors) that can be procedurally abstracted (i.e. named and parameterized [JLS85].

To give a first summary, abstraction in data modeling provides support for describing the *statics* of an application (i.e. its structures and invariants), whereas the abstraction mechanisms found in programming languages are designed to help in defining the *dynamics* of a system (i.e. its short- and long-term behavior).

Modern design methodologies and languages emphasize the importance of an integrated view on the statics and dynamics in system design and implementation. They utilize the various forms of data abstractions (e.g. ADTs, classes) to encapsulate state, behavior and invariants, and (re-)use such capsules as adaptable building blocks for large information systems [Mey88].

3.3 Operational Support for DBPL Applications

Up to now only a few DBPLs provide the operational support considered necessary for realistic data-intensive applications. The key concept by which operation abstraction is achieved is that of a *transaction* [Gra81, Gra78]

In the DBPL system [RS81, MRS84] transactions are first-class language constructs that provide, in addition to procedural abstraction, concurrency and recovery abstraction based on serializability and "undo/redo" semantics [BHG87].

Again, the relational approach provides an excellent framework for integrating that kind of operational support into a DBPL. By representing data classes through (disjoint) sets and abstraction computations through (first-order) predicates the notion of disjointness and sharing, locking and validation can be transformed into a logic framework that provides sound and "efficient" solutions [KR81, BJS86].

Additional work has been done in high level support for data-intensive applications in distributed environments. References can be found in [Lis84, JGL*88, LEE*87].

4 Integrated Database Programming Languages

In the following figures, DBPLs are grouped into four broad categories: Relational DBPLs support bulk data definition and manipulation using set (relation) types and quantified predicates (query expressions). Persistent programming languages provide persistence for all of their data values and include language mechanisms for static as well as dynamic binding and type checking. Object oriented DBPLs emphasize the notion of object identity using classes to organize objects and operations on them. The remaining DBPLs in Fig. 4 are all excelled by their elaborate type systems supporting a direct representation of the data modeling concepts outlined in Section 3.

Figures 2 through 5 characterize the conceptual foundations of several DBPLs. Syntactic or implementation details of particular languages are de-emphasized in this section. The following criteria are applied:

Model of Persistence: How is data lifetime modeled within the language (see Section 2.1)?

Binding: When are names bound to constants, variables, procedures or types (see Section 2.1)?

	Pascal/R	DBPL	Aldat	LDL
References	[Sch77, SM80]	[SEM88, MS89]	[Mer77, Mer84]	[NT89, TZ86]
Model of Persistence	database = record of relations	Modules, orth. persistence	workspace	workspace (?)
Binding	static & dynamic	static	?	static
Typing	static	static	?	static
Identification	value based	value based	value based	value based
Bulk Data Structures	relations not type-complete	sets type-complete	relations not type-complete	sets type-complete
Bulk Data Manipulation	iterators, quantified expressions	(recursive) quantified expressions, iterators	relational algebra, recursive function equations	(recursive) rules
Recovery	no	Transactions	no	no
Concurrency	no	Transactions	?	no
Data Abstraction	no	ADTs, modules	no	modules
Polymorphism	no	no	no	no
Exceptions	no	no	no	no
Algor. Kernel	Pascal	Modula-2	—	—

Figure 2: Relational DBPLs

	PS-Algol	Napier88	Amber
References	[ACC81, ABC*83, Gro87b]	[DCBM89, AM88, AM87]	[Car86a, Car86b]
Model of Persistence	single address space	single address space	side-by-side address spaces (orthogonal)
Binding	static & dynamic	static & dynamic	static & dynamic
Typing	static & dynamic	static & dynamic	static & dynamic
Identification	references	references	references
Bulk Data Structures	—	—	—
Bulk Data Manipulation	—	—	—
Recovery	no	no	no
Concurrency	limited	no	no
Data Abstraction	no	ADTs	modules, ADTs
Polymorphism	no	parametric polym.	multiple inher.
Exceptions	no	no	yes
Algor. Kernel	S-Algol	—	—

Figure 3: Persistent programming languages

	OPAL	O_2	O++	E
References	[CM84, MS87] [MJAP86]	[LRV88, LR89] [BCD89]	[AG89a, AG89b]	[RC87, CD87]
Model of Persistence	single address space	single address space	side by side address spaces	side by side address spaces
Binding	dynamic	static & dynamic	static	static
Typing	dynamic	static & dynamic	static & dynamic	static & dynamic
Identification	object based	object based	pointers	pointers
Bulk Data Structures	sets (and user-defined collections)	sets, lists	clusters not type-complete	files, insert-able arrays
Bulk Data Manipulation	iterators, optimized set-oriented operators	iterators, "embedded query language"	iterators, fixed-point semantics for recursion	iterators
Recovery	yes (shadows)	?	no	yes
Concurrency	yes (Transactions)	?	?	yes (Transactions)
Data Abstraction	yes (methods)	yes (methods)	yes (methods)	yes (methods)
Polymorphism	single inher.	multiple inher.	single inher.	single inher.
Exceptions	yes	no	no	no
Algor. Kernel	Smalltalk	(C/Basic)	C++	C++

Figure 4: Object-oriented DBPLs

	Adaplex	Galileo	TAXIS	Machiavelli
References	[SFL83]	[ACR85, AGOO88] [AGO89]	[MW80, BMW84] [CRNM88, MBW80]	[OBB89, Oho88]
Model of Persistence	modules, persistence only for entities	modules (work-space implem.)	no transient data	modules (?)
Binding	static	static	static	static
Typing	static	static	static (?)	static
Identification	object based	object based	object based	value & object b.
Bulk Data Structures	entity sets not type-complete	classes, sequences	class extents	sets type-complete
Bulk Data Manipulation	iterators, quant-ified predicates	iterators, quant-ified predicates	class iterators	generalized rel-ational algebra
Recovery	yes	?	no	no
Concurrency	yes (Transactions)	?	yes (Transactions)	no
Data Abstraction	ADTs	(semi) ADTs	no	ADTs
Polymorphism	generic modules, single inher.	single inher.	multiple inher.	parametric polym., multiple inher.
Exceptions	yes	yes	yes	yes
Algor. Kernel	Ada	(early ML)	—	Standard ML

Figure 5: DBPLs with advanced type systems

Typing: When do type checks take place (see Section 2.3)?

Identification: How are objects identified (see Section 2.4)?

Bulk Data Types: Which built-in type constructors are used to define large collections of (persistent) objects? How do these type constructors interact with other types (type-completeness, see Section 2.5)?

Bulk Data Manipulation: Does the language support abstraction from iteration (see Section 3)? Are there predefined iterators for the built-in bulk data types?

Exceptions: Can exception handling be expressed within the language?

Recovery: Is there a notion of *atomic* transactions involving failure recovery?

Concurrency: Is it possible for several applications to access shared data simultaneously?

Data Abstraction: What are the mechanisms of information hiding (see Section 3)?

Polymorphism: Is it possible to write generic code that is statically type checked? Does the type system include subtyping rules (see Section 2.6)?

In addition to these criteria, the tables include specific references (e.g. to language reports) for each of these languages.

5 On Methodologies for Application Design and Implementation

Probably the single most unifying view on database programming is that taken from the position of *constraint maintenance*. As already emphasized the data of interest for our applications serve as (partial) descriptions of structures and processes that exist in their own organizational or physical context independent of any computer system. From that context, data inherit a wide range of qualifications, and the degree to which computerized systems meet those qualifications decides upon their quality.

We will first discuss issues of data qualifications by distinguishing three classes of constraints with increasing generality: type constraints, dependency constraints and semantic constraints. Then we will give a short overview over some major approaches to database programming, each of which concentrates on a particular class of such qualifications.

5.1 Classes of Constraints and Dependencies in Data-Intensive Applications

In many applications one can isolate domains, D, such that, for each element $e \in D$, there exist n partial functions, $f_i : D \rightarrow C_i$, $1 \leq i \leq n$, each of which associates with e an element of some co-domain, C_i. Some basic restrictions on f, D and C may serve as a starting point for constraint and dependency classification.

In the simplest case, co-domains are supposed to be *pre-defined* and *time invariant*. Those static co-domains may be given by enumeration of their elements or by constraining larger, implicitly defined sets. Such constraints can be upper and lower bounds or some other propositional *domain constraint*. For example, take the domain Employees, and the co-domains given by the value sets of the types string, cardinal, Boolean, that function as employee name, salary and marital status.

Frequently, n is fixed (for a long time) and each element, $e \in D$, can best be perceived together with its n associates, $f_i(e)$, as one *structural entity* with n components, identified by appropriate component selectors, such as `e.ci` , `e[i]`, $\ldots (= f_i(e))$. One way or another, all data models do

offer constructs to express and maintain such structural constraints: the class structure in object-oriented models serves that purpose, as well as, for the more classical models, the structure of tuples in relations, records in DBTG-sets, or segments in IMS hierarchies.

For a wide range of domain and structural constraints there exist standard compile-time techniques to map the constrained data automatically and effectively into efficient storage structures and to prove that all operations of an entire application program maintain the constraints. Out of obvious reasons we will use the term *type constraints* for the combination of both, structural constraints and (static) domain constraints.

However, the above co-domains need not be static; they may as well be *dynamic* in the sense that their elements and cardinalities do change over time. In such cases, the qualification of constrained entities depends on the existence of their contributing entities, giving rise to the class of *dependency* constraints. For example, take the domain, `worksOn`, that associates entities from the `Employees` domain and the `Projects` domain under the assumption that projects and employees may come and go.

Depending on the application semantics there may be many more ways to qualify data. Frequently, those *semantic constraints* can be formalized adequately by general first-order invariants on data and by pre- and postconditions on transactions.

5.2 On Methodologies and Environments for Data-Intensive Applications

The decision for a particular design methodology, i.e. for a set of tools, tasks, and techniques considered appropriate for the development of a data-intensive application system, depends heavily on the extent to which the above mentioned classes of constraints and dependencies are prevalent in the application and the degree to which they are supported by the data model at hand.

Early design methodologies, such as SSADM or multiview, concentrated on representing domain constraints and structural constraints, modeling the entities of an application by the type system of the programming language in use.

For data-intensive applications entity models were soon extended to Entity-Relationship models [Che76, Che80, Che85, SSW80, TYF86, Mar88] that can also capture various forms of dependency constraints (1:1, 1:n, n:m). Conceptually, however, the E-R approach can be encountered as a generalized type design methodology: the dependencies modeled by an E-R diagram are mapped into the database schema/type, and the DBMS acts as a global runtime facility that maintains dependencies by standard checks at transaction commit time. An overview of these more traditional approaches can be found in [OSV82, OST83, OSV86].

An essential step forward was the development of integrated methodologies that support both, the design of DB schemata and the programming of DB transactions [ABLV81, BR84, RRU*83].

A remarkable contribution was the LIDAS environment, an interactive design system that includes the database programming language Modula/R [KMP*83] and the design tool GAMBIT [BDRZ83]. Based on an extended E-R model, GAMBIT provides interactive support for the specification of entities and dependency constraints. Furthermore, it performs standard mappings of such specifications into relational data structures and transaction skeletons that consist of guarded database update operations. Such conditional updates maintain dependency constraints and restricted classes of semantic constraints. It is essential to this approach that the underlying integrated database programming language has built-in high level language constructs for data of type relation, such as first-order Boolean and relational query expressions, and supports the notion of transactions [MRS84].

Most current developments in database application support are characterized by that shift of responsibility for global constraint maintenance away from the DBMS and towards integrated database programming environments that care about entire application systems.

The authors and their collaborators are involved in project DAIDA, its goal being to design and build such an advanced software engineering environment for developing and maintaining data-intensive applications [BJM*89, BMSW89, SWBM89]. Essential novel characteristics of the DAIDA approach

are:

- the utilization of a Knowledge Base Management System to represent
  - application-dependent knowledge about the problem domain, collected as part of the requirements specification activity;
  - meta-knowledge about the current system's design decisions and evolutionary history; and
  - application-independent knowledge such as design and implementation strategies for data-intensive software.
- the application of three specific languages for the description of data-intensive software at each stage:
  - the knowledge representation language Telos for domain analysis [BGM89, BKMS89]
  - the semantic data model TDL for conceptual design of system states and transitions [Gro87a, MBW80]; and
  - the imperative database programming language DBPL for the efficient management of typed data including sets/relations [SEM88].

Since a major effort of the DAIDA project concentrates on the production of high-quality database application software, the mapping between the TDL level and the DBPL level is of particular importance. TDL is based on the semantic data model of Taxis [MBW80], but has been refurbished to permit predicative specifications of transactions by general semantic constraints given as invariants and pre- and postconditions. Furthermore, TDL maintains Taxis' object-oriented approach, however, its expression language, unlike Taxis, is based on set theoretic notions.

For the mapping between TDL designs and DBPL implementations DAIDA makes the following assumptions:

- for a given TDL-design there may be substantially different DBPL implementations, and it needs human interaction to make and justify decisions that lead to efficient implementations;
- the decisions are too complex to be made all at once — let alone automatically; it needs a series of refinement steps referring to both data and procedural objects;
- not all objects and decisions relevant for the refinement process need to have formally assigned properties;
- these properties should be recorded to support overall system consistency and evolution.

To meet these requirements, DAIDA relies heavily on current work of J.-R. Abrial [AGMS88].

- TDL designs are translated into Abstract Machines with states represented by mathematical objects like functions and sets, and state transitions defined by Generalized Substitutions;
- the formal properties of Machines and Substitutions are assured and organized by Abrial's interactive proof assistant, the B-Tool [AMSV88].

DAIDA's first results support the conclusion that Abrial's methodology of refining Abstract Machines is a promising direction to study the issue of data-intensive application development in a provably correct manner. In this setting, a language like DBPL with a set- and predicate-oriented approach to data modeling, a rich typing system, and abstract support for concurrency, recovery and persistence, turns out to be an appropriate target for the refinement of TDL specifications into database application software.

6 Conclusion

By now it has been recognized that the production of high-quality software for data-intensive applications suffers severely from the lack of *integrated* programming tools, methodologies and environments. Current approaches are based on separately developed technologies like database models and database management systems, programming languages and compilers, communication protocols and operating systems. In current applications, these pieces of technology have to interact through ad hoc and narrow communication links leading to poor performance, difficult maintenance and complex programming.

The classical examples of such ill-fitting components, assembled without change, are the interfaces between database query languages and host languages used for application programming: they lead to the well-known *impedance mismatch* [CM84]:

Databases operate set-at-a-time on many data instances of few built-in data types. They have a declarative style for queries, centralized data and integrity design (by the DBA), and a small set of powerful operators.

Programming languages typically operate one-element-at-a-time on few data instances of a multiplicity of data types. They have an imperative style, separate compilation, pre and postconditions local to procedures and user defined types and operations.

In recent years, research and development in the area of *database programming languages* (DBPLs) has produced interesting results to overcome these mismatches by a "seamless integration" [CM84] of database functionality in programming languages.

To date there exists also a severe *competence mismatch* between the DB and PL area. Over the decades, the field of programming languages has agreed upon a rich set of high-level principles that make the various dimensions of computational technology available in a safe and user-acceptable form. In the past, the area of databases definitely failed to reach a comparable level of representative abstraction and linguistic maturity.

References

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

[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.

[ABLV81] P. Atzeni, C. Batini, M. Lenzerini, and F. Vallanelli. INCOD: A System for Conceptual Design of Data and Transactions in the Entity-Relationship Model. In P.P.S. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis*, pages 379–414, ER Institute, 1981.

[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.

[ACR85] A. Albano, L. Cardelli, and Orsini R. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[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.

[AG89a] R. Agrawal and N.H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *ACM-SIGMOD International Conference on Management of Data*, pages 36–45, Portland, Oregon, June 1989.

[AG89b] R. Agrawal and N.H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[AGMS88] J.-R. Abrial, P. Gardiner, C. Morgan, and M. Spivey. *Abstract Machines, Part 1-4.* Technical Report, 26 Rue des Plantes, Paris 75014, June 1988.

[AGO89] A. Albano, G. Ghelli, and R. Orsini. Types for Databases: The Galileo Experience. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[AGOO88] A. Albano, G. Ghelli, M.E. Occhiuto, and R. Orsini. *Galileo Reference Manual, Version 2.0.* Technical Report, Dipartimento di Informatica, Università di Pisa, February 1988.

[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.

[AM87] M.P. Atkinson and R. Morrison. Polymorphic Names and Iterations. In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, September 1987.

[AM88] M.P. Atkinson and R. Morrision. 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.

[AMSV88] J.-R. Abrial, C. Morgan, M. Spivey, and T.N. Vickers. *The Logic of 'B'.* Technical Report, 26 Rue des Plantes, Paris 75014, September 1988.

[AS89] M.P. Atkinson and J.W. Schmidt. *Tutorium: Datenbanksprachen.* Datenbank Tutorientage, DBT '89, Zürich, Deutsche Informatik Akademie, February 1989.

[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.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Simple and Powerful Database Language. In *Int. Conf. on VLDB*, Brighton, England, September 1987.

[BBO89] 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.

[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.

[BDRZ83] R.P. Brägger, A. Dudler, J. Rebsamen, and C.A. Zehnder. Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions. In C.A. Zehnder, editor, *Database Techniques for Professional Worskstations*, pages 65–96, Institut für Informatik, ETH Zürich, Switzerland, September 1983.

[BGM89] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge Representation as a Basis for Requirements. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems, Springer-Verlag, 1989. (in press).

[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.

[BJM*89] A. Borgida, M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou. The Software Development Environment as a Knowledge Base Management System. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems, Springer-Verlag, 1989. (in press).

[BJS86] S. Böttcher, M. Jarke, and J.W. Schmidt. Adaptive Predicate Managers in Database Systems. In *Proc. of the 12th International Conference on VLDB*, Kyoto, 1986.

[BJW88] A.D. Birell, M.B. Jones, and E.P. Wobber. *A Simple and Efficient Implementation for Small Databases.* Report 24, Digital System Research Center, January 1988.

[BK85] D. Batory and W. Kim. Modelling Concepts for VLSI CAD Objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[BKMS89] A. Borgida, M. Koubarakis, J. Mylopoulos, and M. Stanley. *Telos: A Knowledge Representation Language for Requirements Modeling.* Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, February 1989.

[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.

[BMSW89] A. Borgida, J. Mylopoulos, J.W. Schmidt, and I. Wetzel. Support for Data-Intensive Applications: Conceptual Design and Software Development. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[BMW84] A. Borida, 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*, pages 87–117, Topics in Information Systems, 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.

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

[Car86a] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, Volume 242 of Lecture Notes in Computer Science, Springer-Verlag, 1986.

[Car86b] L. Cardelli. The Amber Machine. In *Combinators and Functional Programming Languages*, Volume 242 of Lecture Notes in Computer Science, Springer-Verlag, 1986.

[Car88] L. Cardelli. Types for Data-Oriented Languages. In *Advances in Database Technology, EDBT '88*, pages 1–15, Volume 303 of Lecture Notes in Computer Science, 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. In M. Carey, editor, *Database Engineering, Special Issue on Extensible Database Systems*, Volume 10, 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.

[CH85] T. Coquand and G. Huet. *Constructions: a higher order proof system for mechanizing mathematics.* Technical Report 401, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, May 1985.

[Che76] P.P.S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Che80] P.P.S. Chen, editor. *Entity-Relationship Approach to System Analysis and Design.* North-Holland, 1980.

[Che85] P.P.S. Chen. *Entity-Relationship Approach: The Use of the ER Concept in Knowledge Representation.* North-Holland, 1985.

[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.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Databanks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CRNM88] 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*, pages 469–487, Volume 303 of Lecture Notes in Computer Science, 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.

[Dat81] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley Publishing Company, 1981.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 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.

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

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

[DN66] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[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.

[DW88] D.J. De Witt. *Extensible Database Systems: An Overview.* Tutorial n.4, Extending Data Base Technology, CINI foundation, Venezia, March 1988.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems – An Advanced Course*, Volume 60 of Lecture Notes in Computer Science, Springer-Verlag, 1978.

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

[Gro87a] DAIDA Group. *Final Version on TDL Design.* Esprit Project 892, DAIDA Deliverable DES 1.2, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1987.

[Gro87b] Persistent Programming Research Group. *PS-algol Reference Manual.* PPRR 12-87, University of Glasgow, Dept. of Comp. Science, 1987.

[Har88] R. Harper. Modules and Persistence in Standard ML. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems, Springer-Verlag, 1988.

[Heh84] E.C.R. Hehner. *The Logic of Programming.* Prentice-Hall International, 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.

[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.

[Hoa69] C.A.R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12:576–581, 1969.

[HR83] T. Härder and A. Reuter. Database Systems for Non-Standard Applications. In *Proc. Int. Computing Symposium*, Teubner-Verlag, Stuttgart, Erlangen, West Germany, March 1983.

[Ive79] K.E. Iverson. Operators. *ACM Transactions on Programming Languages and Systems*, 1(2):161–176, October 1979.

[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 *Proc. IEEE 4th Int. Conf. on Data Engineering*, Los Angeles, USA, February 1988.

[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.

[JS82] G. Jaeschke and H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proc. ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 124–138, Los Angeles, March 1982.

[KC86] S. Khoshafian and G. Copeland. Object Identity. In *Proc. of 1st Int. Conf. on OOPSLA*, Portland, Oregon, October 1986.

[KMP*83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. *Modula/R Report, Lilith Version.* Technical Report, Institut für Informatik, ETH Zürich, Switzerland, February 1983.

[KR81] C.H. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), 1981.

[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.

[LAB*89] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1989.

[LEE*87] W. Lamersdorf, H. Eckhardt, W. Effelsberg, W. Johannsen, K. Reinhard, and J.W. Schmidt. Database Programming for Distributed Office Systems. In *Proc. IEEE Office Automation Symposium*, Gaithersburg, MD, 1987.

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

[Lis84] B. Liskov. *The ARGUS Language and System.* Programming Methodology Group Memo 40, MIT, Laboratory of Computer Science, 1984.

[LR89] C. Lécluse and P. Richard. *The O_2 Database Programming Language.* Rapport Technique 26-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, January 1989.

[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.

[Mar88] S.T. March, editor. *Proc. of the 6th Entity-Relationship Conference*, North-Holland, 1988.

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

[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.

[MD89] D.R. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In *ACM-SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, June 1989.

[Mer77] T.H. Merret. Relations as Programming Language Elements. *Information Processing Letters*, 6(1):29–33, February 1977.

[Mer84] T.H. Merrett. *Relational Information Systems.* Reston Publishing Co., Reston, Virginia, 1984.

[Mey86] B. Meyer. Genericity versus Inheritance. In *Proc. of 1st Int. Conf. on OOPSLA*, pages 391–405, Portland, Oregon, October 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.

[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.

[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, DAIDA Workpackage IMP 3, Deliverable IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming*, pages 355–392, MIT Press, 1987.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, pages 255–260, June 1989.

[MW80] J. Mylopoulos and H.K.T. Wong. Some features of the Taxis data model. In *6th Intern. Conf. on Very Large Data Bases*, Montreal, Canada, October 1980.

[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.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference. In *ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, 1989.

[Oho88] A. Ohori. Semantics of Types for Database Objects. In *Proc. International Conference on Database Theory*, pages 239–251, Volume 326 of Lecture Notes in Computer Science, August 1988.

[OST83] T.W. Olle, H.G. Sol, and C.J. Tully, editors. *Information Systems Design Methodologies: A Feature Analysis.* North-Holland, 1983.

[OSV82] T.W. Olle, H.G. Sol, and A.A. Verrijin-Stuart, editors. *Information Systems Design Methodologies: A Comparative Review.* North-Holland, 1982.

[OSV86] T.W. Olle, H.G. Sol, and A.A. Verrijin-Stuart, editors. *Information Systems Design Methodologies: Improving the Practice.* North-Holland, 1986.

[PA86] P. Pistor and F. Andersen. Designing a Generalized NF2 Model with a SQL-Type Language Interface. In *Proc. 12 Int. Conf. on Very Large Data Bases, Kyoto*, pages 278–288, August 1986.

[PM88] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[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.

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

[RLW85] P. Rovner, R. Levin, and J. Wick. *On Extending Modula-2 for Building Large, Integrated Systems.* Digital Systems Research Center Reports 3, DEC SRC Palo Alto, January 1985.

[RM87] L. Rowe and Stonebraker M. The POSTGRES Data Model. In *Proc. 13th VLDB, Brighton*, pages 83–96, September 1987.

[RRU*83] J. Rebsamen, M. Reimer, P. Ursprung, C.A. Zehnder, and A. Diener. LIDAS – The Database System for the Personal Computer Lilith. In *Proc. INRIA Workshop on Relational DBMS Design / Implementation / Use on Micro-Computers*, Toulouse, February 1983.

[RS81] M. Reimer and J.W. Schmidt. *Transaction Procedures with Relational Parameters.* Report 45, Institut für Informatik, ETH Zürich, Switzerland, October 1981.

[SAM89] J. Stein, T.L. Anderson, and D. Maier. Mistaking Identity. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[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.

[SFL83] J.M. Smith, S. Fox, and T. Landers. *ADAPLEX: Rationale and Reference Manual (2nd ed.).* Technical Report, Computer Corporation of America, Cambridge, Mass., 1983.

[Shi81] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):134–173, 1981.

[SM80] J.W. Schmidt and M. Mall. *Pascal/R Report.* Bericht 66, Fachbereich Informatik, Universität Hamburg, West Germany, January 1980.

[SP82] H.-J. Schek and P. Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *Proc. 8th Int. Conf. on VLDB, Mexico City*, pages 197–207, September 1982.

[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.

[SSW80] P. Scheuermann, G. Schiffner, and H. Weber. Abstraction Capabilities and Invariant Properties Modelling within the Entity-Relationship Approach. In P.P.S. Chen, editor, *Entity-Relationship Approach to System Analysis and Design*, pages 121–140, North-Holland, 1980.

[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. (in press).

[Tei75] W. Teitelman. *INTERLISP reference manual.* Technical Report, Xerox Palo Alto Research Center, Calif., 1975.

[TYF86] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity relationship model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[TZ86] S. Tsur and C. Zaniolo. LDL: a logic-based data language. In *Proc. 12th Conf. on VLDB*, Kyoto, Japan, August 1986.

[Ull87] J.D. Ullman. Database Theory — Past and Future. In *6th PODS*, pages 1–10, 1987.

[Wir71] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1(1):35–63, 1971.

[Wir87] N. Wirth. *The Programming Language Oberon*. Technical Report, Institut für Informatik, ETH Zürich, Switzerland, 1987.

[Zam88] A.V. Zamulin. Data Base Programming Tools in the ATLANT Language. In *Advances in Database Technology, EDBT '88*, pages 563–566, Volume 303 of Lecture Notes in Computer Science, Springer-Verlag, 1988.

[Zdo87] S.B. Zdonik. Can Objects Change Type? Can Type Objects Change? In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, September 1987.