Towards Database Application Systems: Types, Kinds and Other Open Invitations

Florian Matthes Joachim W. Schmidt

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

Abstract

Software development is moving rapidly from the coding of programs to system's modelling utilizing the services provided by open and modular environments. This shift enables service suppliers to gain customers by generalizing the functionality of their products, and it allows service consumers to conveniently buy functionality by simply specializing their needs. It also motivates consumers to construct applications with more than just the most specialized functionality in mind thus contributing to system extensibility and reusability. Finally, the novel view of software construction blurs the formerly quite sharp distinction between the application part and the system part of software development.

In this paper we apply the consequences of the process sketched above to database systems and data-intensive applications. We argue in favor of a more open but yet controlled interaction between database systems and their applications and we discuss the implications on the major abstraction principles to be supported by future database programming languages. Finally, we follow the open invitation issued by novel computer languages to exploit their potent conceptual basis for the benefit of next generation database application systems.

1 Introduction and Motivation

During the past decade, substantial theoretical and technological progress was made towards expressive, efficient, wide and safe interfaces between application programs and database systems. Central to this development is the idea of *database programming languages* [?, ?, ?, ?, ?, ?, ?], offering an integrated language framework for the implementation of a wide range of data-intensive applications.

Recent advances in type systems for programming languages (like ML, Modula-3, Quest, Napier or TRPL [?, ?, ?, ?]) and extensible database systems (like Exodus, Genesis, Postgres, Starburst or Iris [?]) lead us to the understanding that the technology is there to address another fundamental *interface problem* in integrated database enviroments.

Currently, database application programs run against an environment which has its functionality strictly divided into two distinct partitions:

- The database management system constitutes that part of the environment wich provides the specific functionality required for bulk data management, access path utilization, persistence, concurrency, recovery etc. This *Database Environment* is sealed and secured up to rather narrow interfaces for data, query and transaction definition and hides its internal functionality completely.

- The rest of the environment consists of application-specific modules and libraries that solve relevant subtasks of the problem at hand (e.g. statistics packages or some routines for display handling). In contrast to the closed DBMS world much progress has been made in recent years in opening such *Application Environments* and making them easily extensible without compromising on the side of reliability.

Of course, any piece of functionality required by a user program either has to pre-exist in the environment or has to be provided by a newly developed software component. Note that the two partitions differ substantially in their ability (or willingness) to contribute to *functionality extension*:

- Application Environments are designed with the primary goal of extensibility. They support the extraction of repeating patterns of implementations and protocols from individual user programs, the integration of such services into environments, and the re-import into the programs of an entire user community, thus encouraging sharing and re-use of "standardized" functionality. In recent years substantial progress was made in understanding the linguistic and architectural support required to "factor out" and "multiply in" a wide variety of functionality. Modern programming languages provide mechanisms based on typing, parameterization and instantiation to extract *abstract functionality* from sufficiently different contexts and to utilize it for reliable, robust, extensible and economic system construction.

- In contrast, traditional DBMS Environments disallow any substantial modifications of their functionality and disencourage any utilization of their internal functionality. However, it is commonly agreed that next-generation DBMSs have to cope with novel application areas [?] and unpredictable demands. Therefore, future DBMSs have to be prepared for functionality extension on all levels: new data structures, new operations over them, novel transaction mechanisms etc. and this can be achieved only by *controlled modifications* of DBMS functionality.

In this paper we argue that the degree of extensibility required for next-generation Database Environments can not be achieved by implementations based on traditional system programming languages such as C or Modula-2. Such implementations can not be opened sufficiently because of

- *safety deficiencies*: they do not exploit state-of-the-art techniques for system genericity and control by mechanisms like polymorphism, higher-order functions and subtyping;

- *complexity problems*: there is no suitable support for system abstraction and organization, e.g. by taxonomies of collection types or transaction protocols;

- *conceptual mismatches*: traditional system programming languages do not support crucial DBMS demands, such as the need for data independence, query optimization, persistence, or any kind of data distribution control.

We argue that the *same abstraction mechanisms* be used for the construction of the entire environment of a database application program consisting of both, the Database Management System and the Application Environment. As an initial step towards this goal we even want to go one step further and exploit the *same linguistic platform* for the application program and its environment. For the resulting integrated *Database Application System* we expect improvements in the following essentials:

- improved DB system implementations in terms of correctness, robustness, efficiency;
- adequate language support for database system extensions;
- higher productivity for application programmers by factoring out more re-usable code from user programs into the Application Environment;
- evolution of standards based on functionality and protocols rather than on ad-hoc query languages.
- support for DB interoperability (multi-databases), e.g. by access to the protocols of the transaction manager.

In the next section, we identify key requirements of Database Application Systems. Section ?? outlines which of these requirements are already met in an existing language (Quest [?]). To illustrate the improvements to be expected from a single programming language for Database Application Systems, section ?? investigates the definition of iterators and generic collection types in such a generic and extensible language framework. Section ?? sketches the overall architecture we expect from open database application environments. The paper concludes with a list of language issues that still have to be solved in order to make database system programming languages competitive with well-developed database languages, topics that are also subject of our current research.

2 Requirements of Database Application Systems

As argued above, database system implementation and database application programming activities both share a large set of commonalities. We propose to distinguish three main tasks in the construction of integrated Database Application Systems:

Data abstraction to classify, aggregate and generalize data, programs and meta-data;

Localization abstraction to "factor out" repeating or shared patterns of data, program and meta-data;

Implementation abstraction to selectively hide information about structures defined by the previous two abstraction mechanisms.

The following subsections present these three abstractions in turn, relating them to specific activities in database modelling and application programming. A more detailed discussion of the above abstraction principles and their support by existing (database) programming languages can be found in [?, ?].

2.1 Data Abstraction

Research and development in the area of conceptual modelling [?, ?] has isolated three basic *data abstraction principles* to cope with the size and complexity inherent in data-intensive applications. These abstraction principles can be understood as means to capture structural invariants, considered an important subclass of static constraints among data objects: [?, ?]:

Classification / Instantiation is a form of data abstraction in which a collection of objects is considered as a higher-level object. The higher-level object is a characterization of all properties shared by each object within the collection. Classification establishes an *instance-of* relationship and allows to identify, classify and describe (possibly infinite) sets of objects by means of a single object.

Aggregation / Decomposition is a form of data abstraction in which a relationship between n component objects is considered as a single higher-level aggregate object. The *part-of* relationship between the n components and the aggregate object also allows to identify component objects relative to the higher-level object.

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

For the purpose of this paper, the above characterization of the three data abstractions should be sufficient, even if there are substantially different technical definitions, e.g., of the term "similarity" in various data models. We only note that much of the power of these three abstraction principles comes from their generality and *orthogonality*, i.e. the possibility to apply classification, aggregation and generalization to objects that 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 [?]).

In the context of Database Application Systems, it is not only necessary to classify, aggregate and generalize *data structures* of a given application, but also to reduce the complexity of the huge number of other entities (functions, modules, types, iterators) found in the application environment by organizing these entities using well-crafted abstractions.

2.2 Localization Abstraction

We already emphasized in the introduction the need to factor out possessions of data and functionality from individual applications into a database application environment and to make them available for shared use by a larger community. The primary goal of this localization process is to increase the consistency between applications and to avoid duplication of effort. Localization becomes indispensable in any environment that is subject to change or evolution.

The basic mechanism to achive localization is *naming*: The binding of a name to an object allows the repeated use of the object denoted by the name in different contexts without duplicating its definition. Section ?? discusses how to substantially enhance the possibiliy for localization by means of *parameterization*, allowing to abstract over partial object descriptions.

As an advanced example for localization, a traditional DBMS allows to factor out data (shared database variables) but also to factor out complex functionality (persistence management, access-path maintenance, crash recovery) from data-intensive applications.

2.3 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 in a Database Application Environment.

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.

The main non-functional (or operational) requirements of database applications can be summarized as support for persistence, atomicity, concurrency and *data independence*. Whereas the first three aspects are also captured by (special-purpose) programming languages [?, ?, ?, ?, ?] the notion of data independence deserves more attention. A first step towards data independence is to provide multiple implementations for a given abstract data structure (e.g. a relation). The choice of a particular implementation (e.g. a B-tree for primary key access plus two secondary indices on non-key attributes) is made dynamically, based on additional global information about relation cardinalities and access patterns.

Another important consequence resulting from the desire to achieve efficiency in the presence of data independence is the need to *optimize* expressions involving multiple abstract data structures The basic idea pursued by query optimizers in database management systems is to transform such an expression into a semantically equivalent, but more efficiently executable form. Again, this transformation is guided by global information about the *operational* properties (e.g. time, space) of the actual implementations available at run-time.

3 Towards a Language for Database Application Systems

In this section we give arguments to support our claim that essential parts of the language technology necessary to fulfill the requirements outlined in the previous section already exist in the programming language research community.

Since our goal is to expoit the same linguistic platform for application programming and for database environment extension, the choice of a uniform underlying programming paradigm becomes crucial. Following the distinction in [?] between computational and semantic models, our language will certainly be based on a computational model. Furthermore, since virtually all practical algorithms for DBMS implementation heavily rely on the concept of state and state transition, a *pure* functional or relational language, like Haskell [?], Machiavelli [?] or Life [?] does not seem to be a suitable starting point.

On the other hand, languages like Modula-3 [?] or Eiffel [?] are too biased towards an "object-oriented" execution model to allow the definition of suitable layers of abstraction, e.g. to emulate the built-in functionality of set- and predicate-oriented languages like DBPL [?].

We expect the general flavor of a language for Database Application System programming to be interactive, compiled, strongly-typed, applicative-order, incorporating higher-order functions and imperative features and a primitive set of built-in mechanisms to achieve persistence, atomicity and concurrency in shared environments.

In order to access pre-existing functionality (e.g. operating system services, window-management services, application libraries) from within the language and to make best use of the hardware resources available, the language implementation should be based on well-developed, optimizing compilation technology.

As an example of a particularly well-developed and consistent language we present Quest [?], a functional language with imperative features, explicit type quantification and subtyping rules inductively defined over all type constructions. Although Quest lacks important operational features required in database programming, e.g. persistence management with incremental loading strategies, primitives for concurrency control and recovery, languages like Quest seem to be the right platform to start with since they support the abstraction mechanisms discussed in the previous section without restricting their application to specific contexts.

3.1 Generalized Data Abstraction

It should be clear that Database Application Systems will be composed of a large number and variety of denotable language objects like values, variables, functions, types, function types, modules, module interfaces, database schemas, databases, views and transactions. Our primary concern is therefore not the set of built-in language primitives, but set of abstraction mechanisms available to cope with the variety and complexity of the language objects te be expected in Database Application Systems.

3.1.1 Classification by Types and Kinds

A specific form of *classification* underlies schema definitions in database systems and type declarations in programming languages: They both classify run-time values according to their statically-known structure expressed by means of type constructors.

For several tasks in database modelling and system programming, traditional, monomorphic classification schemes which only allow to assign a single type to a given variable proved to be too limited. Polymorphic languages, like ML or object-oriented languages, usually have rich structures at the type level, e.g. to describe functions that operate uniformly over values of a *set of types* or to denote *orderings* among types expressed by means of inheritance hierachies. The richness of the type level now calls again for a *classification* of type expressions. As discussed in detail in [?], this can be accomplished by introducing a three-level structure of language entities:

Values and functions inhabit level 0, they are classified by means of

Types and type operators at level 1 which in turn are classified by means of

Kinds at level 2. Kinds can be utilized, for example, to distinguish between "plain" type variables and parameterized type operators.

As will be demonstrated later in section ??, types and type operators as first-class denotable language objects allow to achieve a high degree of uniformity between built-in and user-defined functionality. For example, the typing rules underlying relational algebra or relational calculus can be adequately defined using parameterized type operators.

The idea to resolve the mismatch between the expressiveness of untyped languages (like Smalltalk or Lisp) and the security and descriptive power of strongly typed languages (like Modula-2, Modula-3 or Standard-ML) by including types as denotable and manipulable language objects is not new. However, early languages pursueing this idea [?, ?] had to give up static type checking. Quest also has exceptionally rich structures at the type level, including type variables, subtyping and recursive type operators. In contrast to the above mentioned languages, *static* type checking is preserved by introducing a stratified three level structure and by controlling the interaction between entities of the three levels. In general, higher-level entities act as specifications for lower-level entities and must therefore not depend on the outcome of operations on lower-level entities.

In this text we adopt the conventions of [?] and use lower-case identifiers for level 0 entities, capitalized identifiers for level 1 entities and all caps for level 2 entities.

The following three examples declare and bind four value variables, three type variables, and one kind variable:

let *macSerialNum* = *4711*
let *keyboard* = **tuple let** *sn* = *3132* **let** *name* = *"Keyboard"* **end**
let *monitor* = **tuple let** *sn* = *1344* **let** *name* = *"Monitor"* **end**
let *nameOf* = **fun** *(p:Part):String* = *p.name*

Let *SerialNum* = *Int*
Let *Part* = **Tuple** *sn:SerialNum name:String* **end**
Let *PartFunction* = **Fun** *(p:Part):String*

DEF *COLLECTION* = **OPER** *(E::TYPE)::TYPE*

The function *nameOf* of type *PartFunction* exemplifies an important property of the uniform classification schemes we are interested in: Functions introduced by means of the function value constructor **fun** are *first-class* citizens of level 0. In particular, they are classified according to their signature and result type by means of function types (level 1 entities), denoted by the function type constructor **Fun**.

Examples in subsequent sections will demonstrate that the embedding of functions as true level 0 entities leads to a substantial *reduction* of language complexity. Important language concepts like higher-order functions, polymorphic functions, procedural attachment and abstract data types can all be reduced to more primitive concepts, simply by classifying functions as values.

3.1.2 Aggregation by Signatures and Bindings

In programming languages and database models the concept of *aggregation* is captured traditionally by means of labeled records or tuples composed of value components [?, ?]. Relational and deductive databases typically constrain these components to be values of basic (unstructured) types. By loosening this constraint one arrives at more expressive (non-standard) data models based on nesting [?]. Object-oriented models go one step further by allowing functions as attributes of "objects". This procedural attachment is intended to capture the "behavior" of objects.

Quest exemplifies how to liberate the concept of aggregation from the narrow interpretation as simple record definition, namely by introducing the notion of generalized *signatures* and *bindings*. Extending ideas of Landin and Burstall and Lampson [?], Quest introduces a correspondence betweeen declarations, formal parameters and interfaces, all based on common syntax, and also between definitions, actual parameters, and modules.

A *signature* is a (possibly empty) *ordered* association of types to value variables and of kinds to type variables, where all the variables have distinct names.

sn:SerialNum name:String madeFrom:Set(Part)

The above signature declares value variables *sn*, *name* and *madeFrom* of type *SerialNum*, *String*, and *Set(Part)*, respectively. The following signature introduces a type variable *T*, a type operator variable *C* and a value variable *c* of type *C(T)*. Note that signatures introduce variables from left to right and that such variables can be used after their introduction.

T::TYPE C::COLLECTION c:C(T)

Bindings are (possibly empty) ordered associations of values to value variables and types to type variables, where all the variables have distinct names, for example:

let *sn* = *macSerialNum*
let *name* = *"Mac II"*
let *madeFrom* = *set.create* **of** *keyboard monitor* **end**

The binding

Let *T* = *Part* **Let** *C* = *Set* **let** *c* = *set.create* **of** *keyboard* **end**

binds the the type variable *T* to the type *Part* declared in the previous section, the type operator variable *C* to the type constructor *Set* and, finally, the value variable *c* to a singleton set of type *Set(Part)*.

By decoupling aggregation from the tuple construction, we can aggregate also in formal parameter positions and in module interfaces and apply aggregation uniformly to values, functions, types and type operators:

Let *Display* = **Fun** *(T::TYPE C::COLLECTION c:C(T)) Ok*
Let *Collection* = **Tuple** *T::TYPE C::COLLECTION c:C(T)* **end**
Interface *CollectionInterface* **import** ... **export**
  T::TYPE C::COLLECTION c:C(T)
end

There are subtle issues in defining the instance-of relationhip between bindings and signatures, for example, whether the variable names or the order of signature components should matter. The interested reader is referred to [?] for a discussion of these topics and the alternatives supported in Quest.

To give an idea of the power and flexibility gained by the generalized notion of signatures, we show examples of how to develop a refined description of composite parts:

```
let myMac = tuple
    let sn = macSerialNum
    let name = "Mac II"
    let madeFrom = set.create of keyboard monitor end
end
Let CompPart = Tuple sn:SerialNum name:String madeFrom:Set(Part) end
```

The value *myMac* is a simple aggregation of *independent* value variables. Accordingly, the signature of the tuple type *CompPart* is also a simple aggregation of independent type variables. A first example for the need to be able to express *dependencies* between the components of a binding is the aggregation of function values referring to "local" value variables:

```
let myMac1 = tuple
    let sn = macSerialNum
    let name = "Mac II"
    let madeFrom = set.create of keyboard monitor end
    let printMyself = fun():String "composite part " & name & " " & conv.int(sn)
end
Let CompPart = Tuple
    sn:SerialNum  name:String  madeFrom:Set(Part)  printMyself:Fun():String
end
```

A somewhat more elaborated example may illustrate the need for dependencies between components of a signature:

```
Let SerialNum = Tuple
  T::TYPE  first():T  next(sn:T):T  equal(sn1,sn2:T):Bool  asString(sn:T):String
end
```

Values of type *SerialNum* are tuples of five components, namely a type *T* and four operations on values of type *T*. The tuple type *SerialNum* can therefore be understood as the signature of an algebra for values of type *T* capturing the essential properties of serial numbers: Serial numbers can be created (*first*, *next*), there is an equality defined over them, and they are printable entities. A crucial constraint expressed via dependencies in this signature is the requirement that, for example, the result of the application of *first* has to be a legal argument for the application of *next* or *asString*. Note that *SerialNum* is a level 1 entity that classifies a set of level 0 entities, in particular the following implementation of serial numbers by means of integer numbers:

```
let intSerialNum = tuple
  Let T = Int
  let first():T = 0
  let next(sn:T):T = sn + 1
  let equal(sn1,sn2:T):Bool = sn1 is sn2
  let asString(sn:T):String = conv.int(sn)
end
```

3.1.3 Generalization by Subsignatures and Subtyping

Object-oriented models emphasize the need to capture generalization relationships between object classes and to view objects not only as members of their most specialized class, but also as members of more general classes. Typically, this is accomplished by defining static *is-a* relationships between classes and by indicating the most specialized type of an object at object-creation time.

Again, Quest takes a somewhat less biased approach to capture the concept of *generalization*. Instead of bundling classification, generalization and dynamic extent management in the concept of a class, specialization relationships are defined in Quest inductively by means of *subtyping* rules between all types in level 1. The subtyping relation, written <:, is defined such that if *x* has type *A* and *A<:B* then *x* has also type *B*. The syntax for kind terms of level 2 allows to denote the collection of all subtypes of a given type *B* by a kind of the form *POWER(B)*. Then *A::POWER(B)* and *A<:B* have the same meaning. There is also a *subkind* relationship, written <::, in level 2, derived from the subtype relationship by *POWER(A)<::POWER(B)* if *A<:B* and *POWER(A)<::TYPE* for all *A::TYPE*.

In this paper we only discuss those subtype (subkind) relationships which are induced by the notion of *subsignatures*. A signature *S* is a subsignature of a signature *S'* if it has the same number of components as *S'*, with the same names and in the same order, and if the component types (or kinds) of *S* are subtypes (or subkinds) of the corresponding components in *S'*. For example, the declarations of the previous sections

Let *Part* = **Tuple** *sn:SerialNum name:String* **end**
Let *CompPart* = **Tuple** *sn:SerialNum name:String madeFrom:Set(Part)* **end**

imply *CompPart<:Part*. It should be noted that in particular the existence of recursive and parameterized type definitions requires non-trivial typing rules for the verification of subtyping relationships [?].

The main conceptual advantage of deriving subtyping relationships based on the generalized notion of signatures is to uniformly capture the concept "partiality" as it occurs in the specification of data, functions and modules. Other important aspects of this approach (e.g. the interaction with type abstraction in order to constrain inferred subtype relationships) are beyound the scope of this paper. As an example for the use of subtyping information, take the function returning the name of a part:

let *nameOf* = **fun**(*p:Part*)*:String p.name*

Since *CompPart<:Part*, the following function applications are both correct:

nameOf(keyboard)
nameOf(myMac)

This particular form of genericity achieved by means of implicit subtyping rules is crucial for the quality of Database Application Systems. It allows to build systems that are capable of operating uniformly over a wide range of specialized data structures, even if the specialization was the result of an *unexpected* extension long after the subsystem exporting the generic function was implemented and delivered.

A formal treatment of Quest-like subtype relationships and a semantics for values, types and kinds in Quest can be found in [?, ?]

3.2 Localization by Parameterization

Until recently, programming languages and database systems focussed on different localization tasks: Programming languages employed lambda abstraction and parameterization mainly to define algorithms and to instantiate them with suitable parameters in different contexts. Database systems on the other hand recognized the need to share (database) variables between multiple applications and to localize the operational support (transaction management, set evaluation) for these database variables within a central DBMS. A logical and physical database schema can then be understood as a very detailed parameterization to be used during the instantiation of the generic data structures offered by a given database (model).

Quest incorporates an important building block to regularize these up to now separated approaches by generalizing the concept of *parameterization* and binding. One can define parameterized function values, parameterized type operators and (indirectly) parameterized kind expressions.

The previous subsection already demonstrated the definition of a parameterized function, *nameOf*, exhibiting subtype polymorphism. A more explicit treatment of polymorphic functions can be achieved by the definition of (kinded) type parameters:

let *pair(A::TYPE x:A):* **Tuple** *first:A second:A* **end** =
    tuple let *first* = *x* **let** *second* = *x* **end**

The polymorphic function *pair* takes a parameter *x* of type *A* and returns a tuple that contains *x* twice. The fact that *x* can be of any type *A* is expressed by defining *A* as an additional type parameter of *pair*. Again, note that the type variable *A* is also used in the definition of the result type of *pair*. The last of the following three applications of *pair* illustrates that actual type parameters can also be *inferred* in Quest – relieving application programmers from supplying (essentially redundant) type parameters:

pair(:Int 3)
pair(:String "ABC")
pair("ABC")

Since both types (*Int* and *String*) are of kind *TYPE*, the above actual parameter substitutions are successfully kind-checked at compile-time.

Set types (tacitly assumed to be pre-defined in the previous sections) can be fully defined within Quest by means of a type expression aggregating a parameterized type operator *Set(E)* and the signatures of the admissible set operations operating on values of this type:

Let *SetInterface* = **Tuple**
 Set(E::TYPE)::TYPE
 create(E::TYPE from:Array(E)):Set(E)

```
  empty(E::TYPE x:Set(E)):Bool
  size(E::TYPE x:Set(E)):Int
  member(E::TYPE e:E x:Set(E)):Bool
  get(E::TYPE x:Set(E)):E
  rest(E::TYPE x:Set(E)):Set(E)
  add(E::TYPE e:E x:Set(E)):Set(E)
  union,difference,intersect(E::TYPE x,y:Set(E)):Set(E)
  equal(E::TYPE x,y:Set(E)):Bool
end
```

Section ?? will provide further details on defining set abstractions that can be instantiated with arbitrary element types *E* of kind *TYPE*. The above definition of *Set* in particular expresses the constraint that the arguments of set functions like *intersection* have to be sets of the same element type and that they return homogeneous sets. Instantiations of the type operator look as follows:

Set(Part)
Set(Set(CompPart))

These are two examples of set operations:

```
let allParts = union(parts baseParts)
if equal(parts allParts) then ... end
```

Other interesting type aspects inherent in traditional database systems can also be expressed by means of higher-order type operators. For example, the traditional quantifiers `EXIST` and `ALL` present in SQL as well as a non-standard `MAJORITY` quantifier proposed for Starburst [?], all share the same signature pattern denoted by the following parameterized type constructor:

```
Let Quantifier(E::TYPE Coll::POWER(Set(E)) =
     Fun(collection:Coll predicate(:E):Bool):Bool
```

A quantifier can be instantiated with arbitrary element types *E*, but only with collection types *Coll* of kind *POWER(Set(E))*, i.e. subtypes of sets of *E*. A very specific instance of such a quantifier can be defined and applied as follows:

```
let rec somePart:Quantifier(Part Set(Part)) =
   fun(collection:Set(Part) predicate(:Part):Bool)
      if empty(collection) then false
      elsif predicate(get(collection)) then true
      else somePart(rest(collection) predicate)
      end
let b:Bool = somePart(parts fun(p:Part):Bool p.sn > 4711)
```

To summarize, this examples illustrates how parameterized types allow to "factor out" common protocols from individual application programs into the database environment, and how they can be utilized to enforce global "standards", e.g. for the definition of iteration facilities. As will be discussed in section ??, this standardization of protocols is important to provide enhanced operational support by a database environment that may choose optimized implementations for a given protocol based on case analysis of the actual protocol parameters (e.g. indexed access based on *p.sn*).

Parameterization is the main tool to achieve localization of functionality in complex systems, more specialized mechanisms, like *exceptions*, are beyond the scope of this paper [?, ?].

3.3 Implementation Abstraction by Scoping

The term *implementation abstraction* as introduced in the previous section subsumes the more specific (but also overloaded) term *encapsulation* as used in object-oriented systems. In general, we are interested in defining interfaces to data structures, functions, types etc. that hide particular implementation details (e.g. local declarations, import-export relationships, subtyping relationships) that are only accidental and should be left open for future change without affecting the correctness of clients utilizing the interface. Statically nested procedure scopes, modules, encapsulated class declarations and abstract data types in programming languages, as well as views and the concept of data independence in database environments can all be mapped, more or less directly, to an elementary scoping mechanism.

In Quest, the scope of a value, type and kind identifier is limited to the binding or signature introducing the name. It can be entered via dot notation (field selection). Already in section ??, the notion of a *subsignature* has been introduced. It is also the key to a generalized understanding of implementation abstraction:

let *View* = **Tuple** *a:Int b:String* **end**
let *x:View* = **tuple let** *a* = *3* **let** *b* = *"ABC"* **let** *c* = *true* **end**

By declaring *x* as being of type *View*, with signature *a:Int b:String*, only tuple selections *x.a* and *x.b* are valid on *x*, although the actual binding contains a third value variable binding **let** *c=true*. This example shows a general matching rule between bindings and signatures: A signature *S* is also satisfied by a binding that has an extended subsignature of *S*.

Due to the generalized interpretation of signatures and bindings in Quest (see section ??), this matching rule again does not only apply to record types but also to module interfaces and function declarations.

4 An Exercise in Generic Collections and Iterators

An essential element of DBMS functionality is to provide generic bulk data structures [?] and high-level query languages to efficiently access and manipulate substructures of bulk structures.

In this section we first sketch how to adequately capture this specific externally visible functionality of a DBMS by means of generalized collection types and a standardized protocol expressed via higher-order functions. In contrast to hard-wired services provided by conventional DBMSs and database programming languages, this approach is *intrinsically extensible*. The second subsection demonstrates how well-known specialized implementations of set and relation structures can be introduced systematically as subtypes of the externally visible generalized collection type. In the last subsection we briefly illustrate how important issues like data independence and query optimization can be achieved in such a generalized scenario.

Other authors already investigate the problem of defining generic, reusable data structures [?, ?, ?, ?, ?] or aim at isolating a taxonomy of bulk data types [?, ?]. However, their work does not cover dynamic reconfiguration and global optimization issues, topics we consider vital for reconciling declarative data modelling with efficient data access. Most of the ideas expressed in this and the following section are based on our practical experience in the construction and utilization of set- and predicate- oriented database programming languages (e.g. see [?, ?]).

4.1 Generalized Protocols

An important task in the construction of Database Application Systems (as in any non-trivial system environment) is the specification of protocols that regulate the division of labour between clients and servers of a particular functionality. Analysing existing approaches to interface definition in operating systems, programming languages and database systems, one can distinguish the following three main styles:

Operating system services or other application environment services (e.g. window management routines) are accessible through special-purpose interfaces realized by *parameterized function calls*. Much progress was made (see Unix, Mach) in achieving simplified call interfaces by collapsing traditionally separated functionality (e.g. terminal input and output, local disk access, remote disk access, interprocess communication mechanisms) to a small set of uniform primitives (e.g., open, close, read, write).

Programming languages (or more specialized languages like *Postscript*) can also be viewed as mechanisms to specify a recognized computational functionality. In contrast to the above simple parameterized interfaces, programming language interfaces are extremely generic and allow to express dependencies between individual operations, to exploit contextual information during program interpretation, and even to extend the functionality of a programming language interface by introducing additional declarations.

Database systems traditionally combine the previous two approaches by providing several special-purpose interfaces. Examples are the DBMS facilities for schema definition, physical data description, ad-hoc queries, embedded queries etc.

Much of the power of a relational interface comes from its *over-abstraction*: Relational DBMSs are prepared to accept from their clients predicates in some first-order language together with a specialization that indicates whether it is to serve as a query predicate, a view definition, an invariant or a trigger condition. Other DBMS interfaces, e.g. those for schema declaration or access support, provide their tailor-made functionality at a similar level of abstraction.

According to our understanding of Database Application Systems as expressed in the introduction of this paper, we would like to be able to combine these three approaches in a single, unbiased language framework. As a test case we will examine in this section the definition of a unified, calculus-based "query" interface to values of generalized collection types.

Section ?? already presents a signature *SetInterface*, defining a set type constructor *Set* and algebra operations like *union*, *intersection* over set values. Similar generic interfaces can be defined for other types appropriate for data collection, e.g. lists, (keyed) relations, mappings, views on relations etc. To provide a single, uniform query protocol for such a variety of types, we abstract from individual properties of these collection types and base the protocol definition on an abstract type operator of the following kind:

*Reader::***OPER***(E::TYPE) TYPE*

For the purpose of interface definition, we do not require any additional information about this type operator. As long as we assert that for a given collection type (say *List*) and any element type *E* of kind *TYPE* the property *List(E)<:Reader(E)* holds, we can apply any *Reader* operation also to *Lists*.

Without going into details, *Readers* may be considered as data "sources" since they (lazily) "yield" [?, ?, ?] data elements. Collection types also have the capability of *Writers*, namely acting as "sinks" for data elements. For example, a value *partSequence* of type *Reader(Part)* denotes a homogeneous sequence of values of type *Part*. We do not want to constrain the implementation of this sequence in any way, e.g. a non-materialized view should be a legal *Reader*, too. Therefore, the *Reader* abstraction subsumes other well-known abstractions like lists and list comprehensions in functional programming, sets, relations, multisets and views in database systems, as well as files, streams and pipes in operating systems or channels in the CSP model.

What are the operations we expect to be defined over such abstract *Readers*? *Reader-Operations* should allow to

- map readers to readers of the same or another element type (e.g. *rest, map, select, head*);
- to reduce readers to values of more primitive types (e.g. *empty, size, hom, some, all, majority*);
- to combine readers into a single new reader (e.g. *append, zip, overlay, insert*);
- to perform operations with side-effects over all elements of a reader (e.g. *forEach, for, until, while*);
- to "associatively" select *individual* elements of a reader based on its properties expressed by a predicate analogous to the "de-setting" operation, as found, for example, in Adaplex [?] (*the, theOnly, theFirst, theLast*);

Figure ?? provides a fairly complete definition of such a service protocol given in Quest. The signature *ReaderOperations* defines a list of higher-order function signatures that are intended to provide *iteration abstraction* over homogeneous sequences of data elements. All of these functions are polymorphic in the element type of the sequence (indicated by the type parameters *E* of kind *TYPE*).

The particular choice of operations and signatures depicted in Figure ?? allows to entirely abstract from the properties of the sequence element type, (e.g. a computable equality predicate is not required) and to "lazily" evaluate expressions formed via reader operations (e.g. quantifier evaluation may not require the computation of all sequence elements). Furthermore, all operations constructing a reader from other readers can be defined in a way such that the constructed reader "naturally" respects the given ordering on the input readers. Last, but not least, reader operations are defined in a purely functional style without relying on the concept of mutability, e.g. by providing destructive insert, update and delete operations on streams.

```
Let ReaderOperations = Tuple
  empty(E::TYPE x:Reader(E)):Bool
  get(E::TYPE x:Reader(E)):E
  rest(E::TYPE x:Reader(E)):Reader(E)

  map(E,F::TYPE x:Reader(E) f(:E):F):Reader(F)
  append,zip(E::TYPE x,with:Reader(E)):Reader(E)
  overlay(E;F;G::TYPE x:Reader(E) with:Reader(F) f(:E :F):G)
          unitE:E, unitF:F):Reader(G)

  forEach(E::TYPE x:Reader(E) body(:E):Ok):Ok
  hom(E,F::TYPE x:Reader(E) f(:E):F g(:F :F):F):F
  hom0(E,F::TYPE x:Reader(E) f(:E):F g(:F :F):F unit:F):F

  select(E::TYPE x:Reader(E) p(:E):Bool):Reader(E)
  for,until,while(E::TYPE x:Reader(E) p(:E):Bool body(:E):Ok):Ok
  the,theOnly,theFirst,theLast(E::TYPE x:Reader(E) p(:E):Bool):E

  some,all,majority(E::TYPE x:Reader(E) p(:E):Bool):Bool
  partition(E::TYPE x:Reader(E) p:Array(Fun(:E):Bool):Array(Reader(E))

  head,tail(E::TYPE x:Reader(E) len:Int):Reader(E)
  selectSub(E::TYPE x:Reader(E) i,len:Int):Reader(E)
  removeSub(E::TYPE x:Reader(E) i,len:Int):Reader(E)
  insert(E::TYPE x,into:Reader(E) after:Int):Reader(E)
  reverse(E::TYPE x:Reader(E)):Reader(E)
end
```

Figure 1: Iteration abstraction over generalized collections

Similar iteration abstractions are discussed, for example, in [?, ?, ?, ?]. The following query applies iteration abstrraction and selects all expensive parts from the set *parts* (of type *Set(Part)<:Reader(Part)*).

let *expensiveParts = select(parts* **fun***(p:Part):Bool p.price > 100.0)*

In this setting, a (read-only) view is nothing else than a non-parameterized function referring to a global set variable (*parts*):

let *partView():Reader(Part) = select(parts* **fun***(p:Part):Bool p.price > 100.0)*

Finally, the scoping and binding rules for function parameters also enable the definition of nested predicates (e.g. for referential integrity constraints):

```
all(compParts fun(cp:CompositePart):Bool
    all(cp.madeFrom fun(p:Part):Bool member(p allParts)))
```

Additional generic operations can be defined by assuming *finite* collections (eventually of bounded maximum size) with a computable equality predicate over their elements (see figure ??). The main complication compared with the *ReaderOperations* of figure ?? arises from the fact that for any operation on a *Collection(E)*, an equality predicate for values of type *E* has to be available. Since this equality predicate also captures primary key constraints on relations, the *CollectionOperations* avoid the well-known problems arising from the subtle interaction between `COUNT, AVERAGE`... operations and (implicit) duplicate elimination.

```
Let CollectionOperations = Tuple
  put(E::TYPE x:Collection(E) e:E):Collection(E)
  size(E::TYPE x:Collection(E)):Int
  full(E::TYPE x:Collection(E)):Bool
  remove(E::TYPE x:Collection(E) e:E):Collection(E)
  member(E::TYPE x:Collection(E) e:E):Bool
  equal(E::TYPE x,y:Collection(E)):Bool
  collect(E::TYPE x:Reader(E) into:Collection(E)): Collection(E)
  union,intersect,difference(E::TYPE x,y:Collection(E)):Collection(E)
  select2(E1,E2,F::TYPE target(:E1 :E2):F into:Collection(F)
        from1:Collection(E1) from2:Collection(E2)
        where(:E1 :E2):Bool)):Collection(F)
  select3(E1,E2,E3,F::TYPE target(:E1 :E2 :E3):F into:Collection(F)
        from1:Collection(E1) from2:Collection(E2) from3:Collection(E3)
        where(:E1 :E2 :E3):Bool)):Collection(F)
end
```

Figure 2: Bulk operations for finite collections with a computable equality predicate

The pragmatic goal behind the definition of such generalized protocol templates is twofold: First, application programmers should be encouraged to use high-level access specifications instead of explicit iterations, thus making programs more readable, putting the burden of correctness proofs (e.g. termination) on the implementors of this general protocol, and giving room to optimized implementations of such protocols (see section ??). Second, designers of new, specialized system services (e.g. graph-traversal algorithms) should tailor their interfaces to comply with the more general ones. Such adherence to global protocols improves the understandability of large systems and the interoperability between separately developed system services.

4.2 Specialized Implementations

In a Database Application System it is not sufficient to concisely define the above protocols and to check the consistent use by protocol clients, but it is equally important to have language support for efficient protocol *implementations* without relying on "magic" predefined external services. Again, we first consider traditional approaches to this task:

Operating system and application libraries make heavy use of standard procedure call mechanisms. The execution model is based on a *substitute and call* semantics: The choice of a particular implementation is typically performed by a simple dynamic case analysis of actual parameter values (e.g. the state of a file descriptor allows to distinguish between local and remote file access).

Programming language processors strive for a static selection of protocol implementations. For example, the implementation of a *FOR* loop statement typically involves a static analysis of the loop variable types, the actual loop bounds and the structure of the loop body. Even advanced optimization techniques employed in state-of-the-art compilers are confined to *local* program text or data flow analysis. Separate compilation techniques required in large-scale and shared systems emphasize the lack of a single global system "picture" to successfully guide global optimization techniques.

Database systems traditionally have to live with both, strong demands on the efficiency of protocol implementations on the one hand side, and very limited information to be derived statically from protocol clients (application programs written in a host language). Therefore, the choice of a particular implementation of a query expression is usually delayed in a DBMS until run-time. Another distinctive property of DBMSs is the availability of several functionally equivalent protocol implementations that only differ in their *operational* characteristics (e.g. access time, storage requirements or locking granularity).

Languages for Database Application Systems have to support all three approaches and therefore require a wide range of binding times and binding alternatives. For example, a "good" implementation of the query expression

let *expensiveParts = select(parts* **fun***(p:Part):Bool p.price > priceLimit)*

should be able to exploit local compile-time information (the structure of the selection predicate *p.price > 100.0* or the type of *p*) as well as information about the actual representation of the set *parts* (e.g. as a sequence ordered on *p.price*), the cardinality of *parts*, the value of the global variable *priceLimit* at run-time or even statistical information and application-domain knowledge about allowable part prices.

Efficient utilization of binding alternatives – the choice of binding time as well as the selection of the specific implementation to be bound to – requires a careful organization and high availability of such alternatives.

As an example for such a taxonomy of protocol service implementations, we investigate implementation alternatives for *Readers*. In Quest, these alternatives can be classified by means of type operators that define subtypes of *Readers*. This enables compile-time checks to assure that only (signature-)correct *Reader* implementations are supplied to the *ReaderOperations* depicted in Figure ??. Values of the most simple kind or *Reader* implementations, *SequentialReaders* only provide the following protocol:

Let *SequentialReader(E::TYPE)::TYPE* = **Rec***(Reader)* **Tuple**
  empty():Bool get():E rest():Reader
end

It should be obvious that (potentially infinite) sequences, like characters typed at a terminal, data records received via a network connection, tokens recognized by a scanner, or arbitrary inductively defined sequences (linear intervals, prime numbers, fibonacci numbers) can be cast into this protocol, for example:

```
let rec countUp(from:Int):SequentialReader(Int) = tuple
   let empty() = false   (* infinite *)
   let get() = from
   let rest() = countUp(from + 1)
end
```

Iterations over finite collections (like lists, stacks, queues, bags, sequential files) independent of their actual implementation (e.g. via linked records or arrays) can provide an additional useful piece of information, namely the total *size* of the sequence.

```
Let FiniteReader(E::TYPE)::TYPE = Rec(Reader) Tuple
  empty():Bool  get():E  rest():Reader  size():Int
end
```

Bulk data structures suitable for database modelling and DBMS implementation are special by providing means to *uniquely identify* collection elements by their *properties*. Their generic protocol

```
Let IndexedReader(E::TYPE)::TYPE = Rec(Reader) Tuple
  empty():Bool  get():E  rest():Reader  size():Int
  member(e:E):Bool  getLike(e:E):E  equality():(:E :E):Bool
end
```

subsumes a function to check for membership of a given element in the sequence (possibly avoiding a full traversal) and for "value-indexed" retrieval using the function *getLike* that returns an element with a given index value. Indexed *Readers* require an equality predicate to be defined over their elements. This equality function is usually passed as a parameter of creation operations. It is also made externally accessible as the result of the higher-order function *equality*. Sets and keyed relations are classical examples of this kind of data structures. For example, the following creation of a *Part* relation defines the attribute *name* to be the primary key attribute of *myPartRelation*:

```
let myPartRelation:IndexedReader(Part) =
    relation.create(fun(p1,p2:Part) p1.name is p2.name)
```

Set, relation and dictionary implementations based on hashing or bit vectors also adhere to the *IndexedReader* protocol. Similarly, ordered data structures like ordered lists, ordered arrays, and various kinds of search trees (B-trees, AVL-trees, R-trees, ...) exploit a total ordering on their data elements that allows the fast determination of all elements *greater*, *greaterEqual*, *less*, or *lessEqual* than a given value of the reader element type *E*.

```
Let OrderedReader(E::TYPE)::TYPE = Rec(Reader) Tuple
  empty():Bool  get():E  rest():Reader  size():Int
  greater,greaterEqual,less,lessEqual(e:E):Reader  ordering():(:E :E):Int
end
```

4.3 Efficienciy by Dynamic Case Discrimination

Query optimization and *data independence* are essential for any database application. A substantial step towards data independence is already achieved by defining a uniform *ReaderOperations* interface to a wide range of data structures. Clients of this uniform interface are not "tempted" to utilize sort orders, indexed access, clusterings etc. available only for specific data structures. This apprroach shields clients from implementation changes which are inevitable in persistent and shared Database Application Environments.

As indicated in the previous seciton, creation operations for *Readers* are more sensitive to implementation changes, since more specialized *Readers* (e.g. hash tables) require additional type-specific information (e.g. a particular comparison or hash-function for part names). This dependency on type representation details may be circumvented by "structure directed operator addition" as proposed in [?]. Another approach found in some object-oriented systems is to equip *every* type with (possibly undefined) "methods" for comparison, hashing, ... and to dynamically inquire the availability of method implementations for a given *Reader* element type.

Query optimization can now be understood as the process of selecting a particular value of the *ReaderOperations* type (e.g. *sequentialReaderOperations* based on *SequentialReaders*) to achieve minimal execution costs for a given instantiation of the reader operation parameters (*parts:IndexedReader(Part)*) like in the function application

select(parts **fun***(p:Part):Bool p.price* > *priceLimit)*

This implies a departure from simple *substitute and call* semantics, as this selection process involves a (dynamic) analysis not only of value parameters (*priceLimit*), but also of

- actual type parameters (is there a total order defined over E = *Part*?),
- actual function parameters (does the predicate p = **fun***(p:Part)* ... produce or rely on side-effects; is it monotone in p?),
- bindings in the client's environment (is there an index defined on *p.name*?),
- bindings in the server's environment (which *Reader* implementations are available?).

We are confident that a small set of comprehensible and simple language mechanisms can be isolated to serve as safe and controlled building blocks for general-purpose (runtime) optimizers. Issues of query interpretation, reflection, transformation and expansion can be treated successfully without sacrificing the merits of static type checking and simple compilation schemes as found in well-developed database programming languages.

5 Open Architectures for Database Application Systems

Figure 3: Open Language Processing in Database Application System

The discussion in the previous section (??) emphasizes the need for a substantially improved flow of information between the various components of a Database Application System, for example, between the application program, the parts and pieces of the compiler, the query optimizer and the shared database schema.

Figure ?? gives an idea of the kind of information processed, for example, during the compilation of a modular database programming language. In contrast to traditional implementation techniques for language processors, we envisage an *open* architecture, where the individual components of the compiler are made available to other clients in the DBAS, revealing their well-designed overabstracted interfaces. Compared with the architecture of traditional database programming languages [?] and persistent programming languages [?], the built-in system components are designed to provide enough "genericity" to be instantiated with specific parameters in a wide range of application and system programs. For example, semantic program analysis, basic query optimization and binding optimization in object-oriented languages [?] can all be uniformly supported by formalisms like *attribute grammars* and generic tools for their (incremental) computation [?].

It should be noted that in Figure ??, the traditionally "process-oriented" view of a compiler (mapping from a high-level source language to a low-level target machine) is replaced by a "data-oriented" translation model. A program entity (e.g. a function) may be represented by several co-existing representations (source code, signature, attributed syntax tree, binding) that are tailored to particular information requirements in the DBAS. This shift in perception allows the mutual support and deeper integration of programming language technology: The attribute evaluater and the linker, which both access large, shared and persistent data structures, require *database system support* for integrity control, storage management or access control. Conversely, we already noted that database browsers, query interfaces and query optimizers heavily rely on functionality that is already present in the database language processor.

To summarize, there are strong arguments that database system implementation and database application programming can both be supported by the same DBAS language based on a set of "generic" language features such as typing, kinding, binding, subtyping, subkinding, parameterization and scoping.

6 Research Issues

In addition to the generic, general-purpose language features described in section 3 there seem to emerge arguments for rather specific and novel language concepts that provide direct support for essential interactions in DBAS architectures like those of Figure ??. Examples are languages with reflection [?] that enable programs to analyse their own type and function declarations and to augment their process environment by newly created types and functions derived from this analysis. For example, the linker in Figure ?? supports the (constrained) extension of the process and persistent environment by value and type bindings.

We are currently investigating primitive language mechanisms supporting the following three specific tasks in DBAS programming:

Atomic Updates: Experience with the implementation of database systems and database programming languages indicates that a wide range of transaction models, communication mechanisms and recovery mechanisms can be reduced to a small set of primitive operations on shared, mutable values [?, ?].

Locality Control: The evolution and size of shared persistent data structures requires careful control over the generation of dynamic dependencies between complex data structures and the functions defined over them. In particular, the existence of *higher-order* functions and *references* (object with identity) calls for binding control mechanisms beyond static scoping. As described in [?], traditional polymorphic typing schemes can be extended not only to reason about the structure, but also about the *locality* of data objects in structured object stores [?].

Environment Management: As argued in [?] and [?], purely static binding and type checking mechanisms are insufficient for some of the applications described in the previous section. *Environments* as first-class language objects with specific operations for environment extension and modification may provide the the flexibility required in evolving DBAS.

Acknowledgements

The authors would like to thank Luca Cardelli for providing Quest as an ideal language testbed for many of the ideas expressed in this paper.

References

[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.

[ABW+90] Malcolm Atkinson, Francois Bançilhon, David De Witt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Deductive and Object-oriented Databases.* Elsevier Science Publishers, Amsterdam, Netherlands, 1990.

[AC90] R.M. Amadio and L. Cardelli. Subtyping Recursive Types. Digital Systems Research Center Reports 62, DEC SRC Palo Alto, August 1990.

[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.

[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.

[AKN89] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[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. 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.

[ART90] M. Atkinson, P. Richard, and P. Trinder. Bulk Types for Large Scale Programming. Technical Report 60-90, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, November 1990.

[ART91] M. Atkinson, P. Richard, and P. Trinder. Bulk Types for Large Scale Programming. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology (to appear)*, Lecture Notes in Computer Science, 1991.

[BCC$^+$88] A.L. Brown, R.C.H. Connor, R. Carrick, A. Dearle, and R. Morrison. The Persistent Abstract Machine. PPRR 59-88, Universities of Glasgow and St Andrews, March 1988.

[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.

[BM86] M.L. Brodie and J. Mylopoulos. Knowledge Bases vs. Data Bases. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, Topics in Information Systems. Springer-Verlag, 1986.

[BMS84] M.L. Brodie, J. Mylopoulos, 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.

[Bor88] A. Borgida. Modeling class hierarchies with contradictions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, 1988.

[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.

[Bun90] P. Buneman. Functional Programming and Databases. In D. Turner, editor, *Research Topics in Functional Programming*, pages 155–169. Addison-Wesley, 1990.

[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.

[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.

[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 *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 316–325, June 1984.

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

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[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 *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 377–388, 1989.

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

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

[Ken79] W. Kent. Limitations of Record-Based Information Models. *ACM Transactions on Database Systems*, 4(1):107–131, 1979.

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

[Lar90] J.M. Larchevêque. Incremental compilation in the O_2 database system. (technical note), October 1990.

[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.

[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.

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

[MBCD89] R. Morrison, A.L. Brown, R. Connor, and A. Dearle. The Napier88 Reference Manual. PPRR 77-89, Universities of Glasgow and St Andrews, 1989.

[MBW80] J. Mylopoulos, P. 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.

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

[Mey90] B. Meyer. Lessons from the Design of the Eiffel Libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

[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.

[MOS91] F. Matthes, A. Ohori, and J.W. Schmidt. Typing Schemes for Objects with Locality. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology (to appear)*, Lecture Notes in Computer Science, 1991.

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

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, 1989.

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

[RT88] 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.

[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.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinski, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[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.

[SFL81] J.M. Smith, S. Fox, and T. Landers. Reference Manual for Adaplex. Technical report, Computer Corporation of America, Cambridge, Mass., January 1981.

[SM89] J.W. Schmidt and F. Matthes. Advances in Database Programming: On Concepts, Languages and Methodologies. In *Proc. 16th SOFSEM'89*, Ždiar, High Tatra, ČSSR, December 1989. Available through Hamburg University.

[SM90a] 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.

[SM90b] J.W. Schmidt and F. Matthes. Language Technology for Post-Relational Data Systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.

[SM91] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proc. of the Fourth Int. Workshop on Persistent Object Systems.* Morgan Kaufmann Publishers, January 1991.

[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.

[SS91] D. Stemple and T. Sheard. A Recursive Base for Database Programming Primitives. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology (to appear)*, Lecture Notes in Computer Science, 1991.

[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.

[Sto90] M. Stonebraker. Special Issue on Database Prototype Systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Wad90] P. Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

[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.