Bulk Types: Built-In or Add-On?

Florian Matthes Joachim W. Schmidt

University of Hamburg
Department of Computer Science
Schlüterstraße 70
D-2000 Hamburg 13
matthes,schmidt@dbis1.informatik.uni-hamburg.de

Abstract

Bulk structures play a central rôle in data-intensive application programming. The issues of *bulk type* definition and implementation as well as their integration into database programming languages are, therefore, key topics in current DBPL research.

In this paper we raise a more general language design and implementation issue by asking whether there should be at all built-in bulk types in DBPLs. Instead, one could argue that bulk types should be realized exclusively as user-definable add-ons to unbiased core languages with appropriate primitives and abstraction facilities.

In searching for an answer we first distinguish two substantially different levels on which bulk types are supported. *Elementary Bulk* essentially copes with persistent storage of mass data, their identification and update. *Advanced Bulk* provides additional support required for data-intensive applications such as optimized associative queries and integrity support under concurrency and failure.

Our long-term experience with bulk types in the DBPL language and system clearly shows the limitation of the built-in approach: built-in *Advanced Bulk*, as elaborate as it may be, frequently does not cover the whole range of demands of a fully-fledged application and often does not provide a decent pay-off for its implementation effort. On the other hand, restriction to built-in *Elementary Bulk* gives too little user-support for most data-intensive applications.

We report our current work on open database application systems which favours DBPLs with bulk types as add-ons, and outline some of the technological requirements for highly reusable implementations of languages with advanced user-provided bulk type definitions.

1 Bulk Data: A Language and System Problem

There is no doubt that bulk types are of central importance in data-intensive application programming. Accordingly, the design and implementation of bulk types is a key topic in DB and DBPL research and development [Sch78, ADG+89, ABW+90, HS89, AB87]. Irrespective of the particular kind of bulk data structures present in a given database programming environment (e.g. lists, sets, relations in traditional DBPLs; class extents in object-oriented databases; base predicates in deductive databases or extensionally defined functions in functional databases), one can distinguish two fundamentally different approaches to bulk type support within a language framework:

Built-In Bulk Types are provided as first-class parameterized type constructors in several programming languages [Sch77, LRV88, OBBT89, NT89], their syntax, type rules, semantics and implementation being hard-wired into the language processor and the run-time system support.

Add-On Bulk Types are defined and implemented utilizing standard built-in language mechanisms (typing, naming, binding, scoping or recursion) of a sufficiently generic general-purpose base language like ML [MTH90], Modula-3 [CDG+88], Napier-88 [DCBM89] or Eiffel [Mey88].

In this paper we analyze the motivations and basic assumptions behind both (extreme) approaches and explore their individual advantages and shortcomings. This investigation is based on the one hand on our substantial experience

[1] This work was supported by the European Commission under ESPRIT BRA contract # 3070 (FIDE).

with the design and implementation of database programming languages incorporating powerful built-in bulk type support [Sch77, KMP+83, MS89]. On the other hand it reports on our current work within the ESPRIT Basic Research Action FIDE (Formally Integrated Database Environment) where we are designing Tycoon[1], a language and system environment with add-on bulk types.

As elaborated in more detail in section 3, the choice between built-in and add-on bulk types is to be regarded more as an architectural decision than a pure language-design problem. Therefore, we will also present our experience gained in the process of implementing built-in and add-on bulk types.

The issue of built-in vs. add-on bulk types has a longer tradition in the Persistent Programming Language community, in the DBPL camp, however, the main goal always consisted in building in advanced bulk types. By reporting on our own past experiences and future expectations we attempt to contribute to a more open and objective discussion on this important DBPL research issue.

In section 2 we report on experience with built-in bulk types by referring essentially to DBPL, a set- and predicate-based procedural database programming language [SEM88]. We illustrate issues in language design, system construction and application programming involving built-in bulk types. Since others have already discussed the general properties and requirements of bulk types in some detail [ART91], section 2.4 only summarizes the gist of built-in and add-on bulk types in terms of their elementary and advanced demands.

In section 3 we point out some of the more severe limitations of the built-in approach that provide a strong incentive to look for systematic add-on approaches to bulk type definition and manipulation in open database application systems. To illustrate this, section 4.1 sketches how typical built-in bulk types can be captured as an add-ons in a state-of-the-art programming language.

Finally, section 5 outlines our current work in the Tycoon environment that aims at defining uniformly and implementing systematically declarative iteration abstractions applicable to a wide variety of bulk structures. Specifically, we explain how the concepts of data independence and query optimization can be generalized to liberally extensible systems. We also identify specific language mechanisms required for the efficient and robust construction of such open and extensible database application systems.

2 Experience with Built-In Bulk Types

Looking back on several generations of models and systems for bulk data management, one realizes that each of them was designed and implemented targeting a rather uniform user community with very similar functional and operational requirements. Based on a thorough understanding of the application domain (e.g. standard banking applications), a high investment in specific though generic built-in structures (e.g. polymorphic relation types) was justified.

We first study the substantial pay-off of such a language investment as exemplified by DBPL, a type-complete database programming language providing generalized (keyed) sets as built-in bulk types. Specifically, we focus on the finer points of DBPL to clarify general subtleties arising in bulk type definition and use. We also relate DBPL design decisions to approaches found in other languages and systems.

In section 2.3, we argue for the necessity to equip bulk types with appropriate high-level abstractions to achieve "declarative" bulk data access and generalized constraint maintenance without impairing their operational qualities (efficient, concurrent, recoverable access).

We conclude the discussion of built-in bulk types by summarizing requirements for bulk types for data-intensive applications in persistent environments (section 2.4).

2.1 Bulk Types as Parameterized Type Operators

DBPL provides a built-in parameterized relation type constructor that is parameterized not only by the *relation element type* but also by an (optional) ordered list of *key components*, which have to be selectors for values of the relation element type:

`relation` <key-components> `of` <element-type>

Here are some examples of simple and non-standard relation type declarations in DBPL:

[1] TYCOON: Typed Computational Objects in Open Environments

```
type
  Persons = relation name of record name:String; age:CARDINAL end;
  Employees = relation persNum of record
      persNum:CARDINAL; salary:REAL; age: CARDINAL; partners :Persons
      case status: (PartTime, FullTime) of
       PartTime: hoursPerMonth: CARDINAL; | FullTime: employedSince: Year;
      end
    end;
  PersNumSet = relation of persNum;
```

A value of a relation type is a homogeneous collection of simple values (*PersNumSet*) or aggregate values (*Employees*) as specified by the relation element type. The number of elements, called the cardinality of the relation, is unconstrained.

The cardinality of relation variables can vary dynamically, i.e. all relation variables in DBPL programs are *mutable* values. Most built-in bulk types (like those of Pascal/R, E or O_2 [Sch77, RC87, LRV88]) and add-on bulk types (collections defined in Smalltalk, Eiffel, Trellis/Owl or Modula-3 [GR83, Mey88, SCB+86, CDG+88]) define mutable entities. Bulk types in functional languages (Machiavelli, FQL, ADAPTBL or TRPL [OBBT89, Nik88, SSS90, SS91]) define pure values and therefore do not offer destructive insertion, deletion or element update operations on variables of bulk types (see also Fig. 1).

The choice of the relation element type in DBPL is unconstrained, i.e. the relation type constructor is truly data type complete like the more standard array, sequence or list type operators. Relation types are integrated *orthogonally* into DBPL in the sense that they can be utilized uniformly in (nested) type declarations, variable declarations, parameter positions, module declarations etc. Furthermore, values of relation types adhere to standard naming, typing, binding, scoping and persistence rules.

A closer look at DBPL reveals, however, some complications inherent in all bulk types (relations, maps, etc.) that are to provide *associative element access* [ART91, SDDS86, SFL83]. In DBPL, the key of a relation defines a list of components of the relation element type that *uniquely* determines each relation element. Key components have to define total functions over the domain of possible relation elements. Therefore, they must not be components of variant sections of a variant record type (e.g., the field *hoursPerMonth* is not a valid key component for *Employees*). An empty key component list is a synonym for an enumeration of all components of the element type that are not contained in a variant section of a variant record; in this case a relation is just a *set* of relation elements (*PersNumSet*).

Whereas the above restrictions are direct consequences of the semantics of keyed relations, the decision in the current implementation of DBPL not to allow key components that exceed 4K in size or that involve nested relations – which would require a (possibly nested) relation equality test for every relation update operation – is based on "pure" engineering considerations.

2.2 Elementary Operations on Instances of Bulk Types

A minimum set of operations required for values of bulk types are the means to define an empty bulk structure (*empty set*), to combine two bulk structures of the same type (*set union*) and to access individual elements of a bulk structure (*choose*; for a more rigorous discussion of these issues see [SS91] and [Wad90]). The following subsections present alternative approaches to these tasks.

2.2.1 Generation of Bulk Values

A bulk type has at least to have functions to yield "empty" (*nil*) bulk values for a given element type and that add a single element to a bulk structure (*cons*). Many programming and database languages provide syntactic support to denote concisely values of bulk types by an enumeration of their elements as demonstrated by the following DBPL example:

```
persons:= Persons{}; persons:= Persons{person1,person2}
```

The type name preceding the curly brackets is required in DBPL to determine the key constraints on the constructed relation value.

2.2.2 Operations on Bulk Values

In an orthogonal language framework there should be no need to define special mechanisms to name, bind, scope or parameterize values of bulk types since these should be "inherited" from the basic types or other aggregate types like records or tuples. In particular, the semantics of assignments, parameter passing mechanisms and equality predicates (monomorphic or polymorphic à la ML) require particular attention from the language designer and the application programmer. The following example should give an idea of the variety of possible semantics for set and record assignment and set equality in different systems and languages:

```
person1:= Person{"Peter", 35}; person2:= Person{"Peter", 35}; person3:= person1;
persons:= Persons{person1};
if persons = Persons{person2} then "deep equality"
elsif persons = Persons{person3} then"shallow equality"
elsif persons = persons then "identity"
else "???" end;
```

DBPL, Pascal/R, LDL and Machiavelli have built-in deep equality semantics, whereas O_2, for example, utilizes shallow equality for sets of objects. Languages with add-on bulk data types (like PS-algol, Napier-88, Eiffel or Smalltalk) rely heavily on reference semantics (L-value bindings [AM88], object identity [KC86]). Implementations of collections of composite objects in these languages typically store references to collection elements and often provide a shallow equality test in addition to the built-in identity test. Because of this sharing, an update to a collection element, like

```
person1.age:= 36;
```

in these systems simultaneously affects the value of collection variables (e.g., *persons*) referencing this object.

DBPL – as an extension of Modula-2 – provides strict copy semantics for assignments, value parameters, relation equality and relation construction. Since there is no notion of object identity in DBPL, it may be necessary for database programmers to "implement" object identifiers by explicitly managed unique keys. On the other hand (as argued at length in [MOS91, SM91]) copy semantics give the database user fine control over the locality of and dependencies between data objects in logically partitioned (distributed, heterogeneous, autonomous) object store environments.

The nature of bulk types usually calls for incremental update operations in addition to the plain bulk assignment. Here are some examples of such relation update operations (assignment, insertion, deletion and update) in DBPL:

```
persons:= oldPersons;
persons:+ newPersons;
persons:– oldPersons;
persons:& updatedPersons;
```

These operations on relation values are defined such that the *key integrity constraint* (no duplicate keys) is maintained dynamically at run-time. For example, the above insertion operation (:+) will ignore all elements in *newPersons* with key values that are already present in *personSet*. The standard function *RESTRICTED()* allows the programmer to detect such constraint violations (a posteriori). This deterministic behaviour of the update operators is in strong contrast to the non-deterministic semantics of the relation union operation in DBPL:

```
persons:= Persons{Person{"Peter", 35}, Person{"Peter", 36}}
```

DBPL does not specify which of the two tuples with conflicting key values is chosen to become the member of *persons*. Without going into details, this non-determinism also extends to DBPL's set-oriented query expressions and to its recursive fixed-point queries and was found to be essential for effective query optimization without pertubating the comprehensibility of DBPL programs [SM91, ERMS91].

To our understanding, the definition of useful bulk types (maps, relations, bags, arrays etc.) has to include the possibility to attach *dynamic integrity constraints* on instances of bulk types, e.g. the support for (associative) identification of collection elements. Other examples for such integrity constraints are subset relationships between the extent of a subclass and its superclass(es), referential integrity constraints, cardinality constraints etc.

Some of the subtleties and pragmatic considerations of DBPL (and other systems) are related to the handling of possible *c*onstraint violations. Therefore bulk types are intimately related to other language mechanisms (transactions, exceptions, triggers, evaluation strategies, side-effects) that at first glance seem to be orthogonal to bulk type design.

2.2.3 Element Selection

There are three main approaches to accessing individual elements in a bulk value:

Cursors provide a means to select individual elements from a bulk structure according to a fixed (perhaps undefined) ordering on the collection elements. Cursors can be defined as explicitly managed data structures or they can be manipulated implicitly (via side-effects) as it is done via traditional interfaces between databases and programming languages.

A conceptually rather different approach to navigation through homogeneous bulk values is taken in functional languages: for example, the functions *car* and *cdr* (*head* and *tail* or *choose* and *rest* etc.) allow the access to individual collection elements as well as the book-keeping of the progress of the iteration [SS91]. The following example illustrates how to skip over the first n elements of a stream in a functional framework. The head element of a stream corresponds to the value at a cursor position while the binding to the tail stream provides the ability to "move" the cursor.

```
let rec skip(s:Stream n:Int):Stream =
  if n <= 0 then s
  elsif emptyStream(s) then s
  else skip(tail(s) n-1)
  end
```

DBPL provides – in addition to iterators and high-level query expressions – low-level navigational operations on relation values (for details see [SEM88]). The following DBPL code fragment illustrates a programming situation that requires an explicit navigation over bulk data structures. The *display* and *getChoice* routine embedded in a `while`-loop allow a user to interactively scroll forward and backward through the relation *persons*.

```
LOWEST(persons, p); choice:= forward;
while not EOR() and not (choice = quit) do
  display(p); choice:= getChoice();
  if choice = forward then NEXT(persons, p)
  elsif choice = backward then PRIOR(persons, p)
  end
end
```

Iterators abstract from the details of the element selection process. Procedural languages like DBPL, SETL, Galileo or CO_2 are equipped with loop constructs to execute a statement (sequence) for all elements of a collection in turn:

```
for each p in persons: p.age > 30 do p.age:= p.age + 1 end
```

Higher-order functions like *set-reduce* or *map* [SS91] provide another form of iteration abstraction in a functional framework:

```
let incrementAge = fun (p:Person):Person tuple p.name  p.age + 1 end
let newPersons = map(persons incrementAge)
```

It should be noted that cursors offer more complex element selection patterns than iterators which only enable a static nesting of iterations. Interactive scrolling through a relation or linear-time merging of sorted lists are typical application examples that call for cursors or general-purpose "continuations" [SNR90]. However, for most programming tasks, such explicit control is not required, and iterators, an algebra or a calculus are to be preferred. The expressive power of various iterators is discussed, for example, in [HS89, SS91].

Associative Element Selectors provide an elegant solution for programming tasks that require value-based access to an individual collection element. As an example, the following DBPL statement increments the age of *"Peter"* in the set of persons:

```
with persons["Peter"] do age:= age + 1 end
```

In languages without such a facility one cannot adequately exploit the key uniqueness constraint and one has to resort to standard bulk iteration constructs:

```
for each p in persons: p.name = "Peter" do p.age:= p.age + 1 end
```

Associative set and element selection is also a basis for logic- and constraint-based programming languages like LDL [NT89] or Life [AKN89]):

```
person(Name, Age), Age>60.
person("Peter", PetersAge).
```

It should be emphasized that in the above discussion on mutability, copy vs. reference semantics, the handling of constraint violations or basic iteration facilities, we do not argue in favour of a specific solution. On the contrary, we are more interested in identifying the dimensions of the available design space for bulk type definition since a better understanding of this space seems to be a prerequisite to devising language mechanisms supporting a wide range of (possibly user-defined) bulk types and iteration facilities.

2.3 Declarativity and Bulk Types

DBPL provides named and parameterized access expressions as typed first-class language constructs. (Recursive) database queries, bulk iteration, viewing mechanisms and constraint management can thereby be successfully distilled to a single, "declarative" iteration abstraction that can furthermore be integrated rather well into a compiled procedural programming environment.

In addition to the element-oriented operations sketched in the previous section, DBPL provides declarative *query expressions* over values of bulk types. There are three kinds of query expressions, namely boolean expressions, selective and constructive expressions.

Quantified Expressions yield a boolean result and may be nested:

```
some e in employees (e.status = PartTime)
all e in employees (e.status = PartTime)
all e in employees some p in e.partners (p.name = "Peter")
```

Selective Access Expressions are *rules* for the selection of relation elements.

```
each e in employees (e.status = PartTime)
```

denotes all elements *c* of the relation variable *employees* (of type *Employees*) that fulfil the selection predicate *(e.status = PartTime)*. A selective access expression within a relation constructor denotes the subrelation of all selected tuples:

```
Employees{each e in employees (e.status = PartTime)}
```

Another context for selective access expressions are iterators (element-at-a-time processing):

```
for each e in employees (e.status = PartTime) do e.hoursPerMonth:= 40 end
```

Finally, selective access expressions can be named and parameterized (see [MS89]), a process conceptually similar to view definitions in relational DBMS:

```
selector casual on (emp:Employees): Employees;
begin each e in emp (e.status = PartTime) end casual;
```

Selectors differ from relation-valued functions in that they allow updates on the selected subrelations. DBPL emphasizes this analogy to array selectors by the following syntax:

```
employees[casual]:+ Employees{peter}
```

The semantics of such selected bulk updates are discussed in [SM91]. In this particular example, the selector *casual* enforces the constraint that only part time employees can be inserted into the relation *employees*.

Constructive Access Expressions are rules for the construction of relation elements:

```
Pair{e.age, p.age} of
  each e in employees each p in e.partners:
    (e.status = PartTime) and (p.age > 22)
```

where *Pair* is the type `record` *ageEmp, agePartner: CARDINAL* `end`.

Again, constructive access expressions can be named and parameterized. Recursive constructor definitions have fixed point semantics similar to stratified datalog programs [ERMS91]. The use of a constructive access expression in a relation constructor creates a relation that contains the values of all tuples denoted by the construction rule.

2.4 The Gist of Bulk Types

The numerous requirements for bulk types (see, e.g., [ABW+90, ART91]) can be classified into *elementary* and *advanced* requirements which we will summarize below. This discussion is valid for both, built-in as well as add-on bulk types, and, therefore, also establishes a framework for section 4 where we investigate environments that enable their users to add-on bulk types meeting these elementary and advanced requirements.

As a first cut, the distinction between an elementary and an advanced requirement can be based on the kind of technology that is required to provide a particular bulk type support (see Fig. 1). Elementary requirements usually can be met by *local* modifications to language processors (compilers, abstract machines, run-time support libraries) essentially based on *programming language* technology. On the other hand, more advanced requirements (as mentioned already in section 2), for example effective query optimization or stratification analysis of fixed-point queries, involve *global* analysis and modifications as provided by advanced *database* technology.

2.4.1 Elementary Requirements

The basic operations outlined in section 2.2 form a basis for bulk type definition. In order to facilitate the definition of new, type-safe bulk structures, a language has to support at least *parametric polymorphism* and *type abstraction* over generic data structures. In most standard programming languages, the definition of bulk types also requires the use of *higher-order functions*, e.g. to define a specialized monomorphic equality predicate at bulk structure creation time to be used for subsequent associative element retrieval.

Since none of the classical programming languages (C, Pascal, Modula-2, Cobol, Fortran, PL/I) satisfies these elementary requirements, database programming languages derived from these languages (Pascal/R [Sch77], Plain [WSK81], Modula-R [KMP+83], DBPL [MS89], E [RC87], CO_2 [LRV88] etc.) had to be equipped with built-in bulk data types, i.e. all polymorphic code and all higher-order functions are hard-wired into the compiler and the run-time support.

On the other hand, modern programming languages like ML, Eiffel, Napier-88, TRPL or Trellis/Owl [SCB+86], have type systems that may be sufficiently generic to allow programmers to define their own bulk data types (lists, trees, bags, sets, relations or multi-dimensional search structures) from scratch. In fact, much work is devoted to the construction of re-usable generic library code for such bulk types that can be later instantiated with specific type parameters and access mechanisms [Mey90].

Despite this progress towards add-on bulk types there is nevertheless much interest in *type quarks* [ART91], i.e. specialized built-in building blocks to be used in the construction of fully-fledged user-defined bulk types. This interest in the DBPL research community is motivated by additional *operational demands* arising in typical data-intensive applications that may not be adequately supported by the language primitives available in the above-mentioned languages.

Support / Language:	DBPL	Maps	Machiavelli	O2	LDL
elementary					
type completeness	√	√	√	√	–
subtyping	–	–	√	√	–
ordering	–	√	–	√	–
sharing	–	√	–	√	–
update in place	√	√	–	√	–
persistence	√	√	–	√	√
access optimization	√	√	–	√	√
advanced					
key constraints	√	√	–	√	√
first-order predicates	√	–	–	√	√
general constraints	–	–	–	√	√
updatable views	√	–	–	–	–
fixed-point queries	√	–	–	–	√
query optimization	√	√	–	√	√
concurrent access	√	–	–	√	–
recovery	√	√	–	√	–

Figure 1: Examples of Languages with Built-In Bulk Types

For example, built-in specialized index structures (*maps*) [ART91] are capable of capturing the essence of associative access to bulk data. By equipping the abstract machine of persistent languages like PS-algol, Napier-88 or Amber [Car86] with map operations, highly *optimized storage structures*, buffering strategies, clustering policies and garbage collection mechanisms can be employed in the run-time system, exploiting the regularity and size of map structures.

2.4.2 Advanced Requirements

Looking at the historic development of relational database programming languages (from Pascal/R, Plain, RAPP, Modula/R to the current definition of DBPL), one can perceive a trend towards increased built-in language support for advanced requirements of data-intensive applications.

In addition to elementary access primitives (get, insert, delete, update and scan) for bulk structures, many DBPLs recognize the need to provide *iteration abstractions* like iterators [LG86], query expressions [BCD89], set operators [Mer84] or data deduction mechanisms in rule-oriented systems [NT89, ERMS91]. The rationale behind such *declarative access specifications* is to provide concise and optimizable notations for complex data selection and data construction tasks. For example, *set comprehensions* [Bun90, Wad90, ART91] nicely generalize relational calculus queries by exploiting the expressive power of computationally complete optimisable languages in a functional framework.

As exemplified by selectors and constructors in DBPL (see Sec. 2.3), iteration abstraction does not need to be limited to set-oriented bulk read access, but can be uniformly generalized to *bulk updates*, *constraint definition* and (recursive) data derivation [ERMS91].

Query optimization as found in traditional DBMSs and some database programming languages aims at exploiting the algebraic properties of high-level access abstractions to (dynamically) choose cost-effective implementations for these abstractions, minimizing execution time, storage requirements or network traffic. Query optimization has to be regarded as a prerequisite for successful iteration abstraction in data-intensive applications: without effective optimizations, programmers fall back to procedural solutions in order to attain the efficiency required in standard database

applications.

Finally, it should be noted that the use of bulk types in shared, multi-user environments also calls for highly *concurrent access* and optimized *recovery mechanisms* to be implemented for variables of these bulk types. Again, database systems and relational DBMS exploit the "declarativity" of access abstractions and hard-wired knowledge about the representation structures for these purposes. (e.g., see [SM91] for a discussion of the support for these advanced operational requirements in the DBPL system).

3 Limitations of the Built-In Bulk Type Approach

There exist several operational database language implementations that provide built-in linguistic and system support for various kinds of bulk types. However, there are two main categories of arguments against the way bulk types are supported: pragmatic arguments against *limitations of extensibility* and conceptual arguments against a *lack of methodology* in the construction of the support systems for bulk type extensions.

In illustrating these arguments our DBPL experience supports that gained by other well-engineered systems (like O_2, Ingres, Oracle, Iris, Orion, Postgres, LDL) which also address advanced database issues (like query optimization, concurrency control, recovery and client-server architectures) in addition to database programming language issues (like typing, scoping, binding, iteration).

The consequences of built-in bulk can only be fully understood by considering the typical implementation strategy for a database programming language, say DB-X. The compiler (interpreter) for DB-X strongly resembles a compiler (interpreter) for a traditional programming language X, extended by scoping, binding and type compatibility rules for built-in bulk type constructors and their built-in operations (e.g. selection, natural join). To meet database requirements (persistence management, access optimization, concurrency control), the code generation phase maps composite DB-X constructs systematically to lower-level constructs (abstract machine instructions, calls to support routines) that provide the required DBMS functionality.

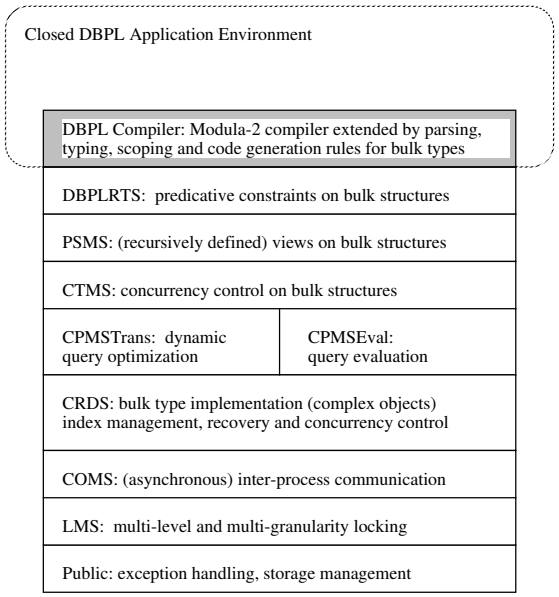

Figure 2: Built-in system support for relational bulk types in the DBPLsystem

As a specific example of this approach, Fig. 2 shows the layers of the DBPL run-time support. Note that DBPL users only access a tiny fraction of the functionality present in the run-time support, namely the topmost (over-abstracted and pre-packaged) interface of the DBPL system (below the bounding box in Fig. 2). In fact, even that functionality is hidden from the user since all calls to (polymorphic) run-time functions are automatically generated by the DBPL compiler, guaranteeing type safety and enforcing concurrency control and storage management protocols

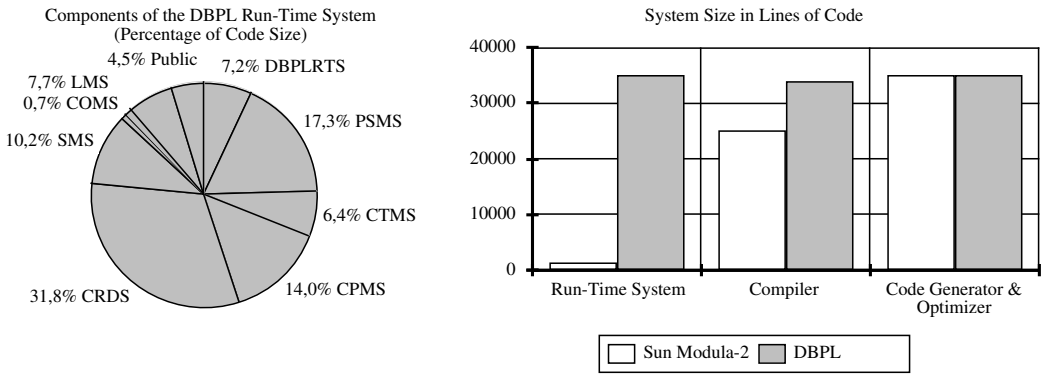

Figure 3: The relative complexity of bulk type implementation in DBPL

required in a persistent multi-user environment. Fig. 3 gives an idea of the complexity of the various tasks involved in the implementation of bulk data types in the DBPL system. A comparison with the (optimizing) Sun Modula-2 system that provides the basis for the DBPL system reveals a significant increase in size of the run-time system.

The built-in approach can be characterized by the following assumptions:

- All language implementation details are hidden from the application programmer (How is a bulk type implemented? How is a query represented, transformed and evaluated? How is serialisability achieved?)
- The efficient implementation of the database language depends crucially on detailed language design time knowledge about permissible language constructs: Which (bulk) types exist? What is the syntax of the query (sub-)language? What are the algebraic properties of the built-in language primitives like equality predicates, comparison operations, aggregate functions?
- Significant optimizations are based on a careful case-analysis within the available language space. Examples are optimizations for special cases like flat relations, unordered collections, read-only transactions or one variable selection queries.

Some of these assumptions are in conflict with the programming language principles of orthogonality and free extensibility. Accordingly, much work in the database community aims at *extending database technology* to support, e.g.

- new aggregate functions in queries,
- new operations within queries (e.g. test for intersection of geometric objects),
- new bulk structures (e.g. ordered sequences),
- new operations in the target list of query expressions,
- new storage structures,
- new join algorithms, etc.

Despite progress made with extensible database systems (for a recent survey see [Sto90]), none of today's database programming language implementations provides adequate support for the above extensions.

Take for example an extension of DBPL by ordered lists. Although this could be achieved by rather limited local changes to an existing database programming language implementation, this can hardly be done by "ordinary" application programmers. The first problem is the lack of polymorphism. It is therefore necessary to write different code for lists of persons, employees, integers, etc. (instead, the type variable *Value* in the following code fragment should be universally quantified):

```
const nil = 0
type
  Value = ... (* element type, e.g. String *)
  ID = CARDINAL;
  Item = record id: ID; val: Value; next:ID end
  Items = relation id of Item;
  List = record items: Items; head, nextIndex: ID end

procedure new(var list: List); begin list:= List{{}, nil, nil} end new;

procedure cons(val: Value; var list: List);
begin
  with list do INC(nextindex); items:+ Item{nextIndex, val, head}; head:= nextindex
  end
end cons;
```

Obviously, relations do not provide the "right" abstraction level at which to build list structures, since the above example exploits relations merely as a typed homogeneous store indexed by integer "addresses". Similarly, relational calculus expressions are not a suitable basis to solve list processing tasks (e.g. to select a sublist of all values satisfying a boolean predicate).

It should be noted that this inadequacy is not a deficiency of DBPL. It can be also found in polymorphic database languages like Machiavelli or Galileo and all commercially available relational database systems, since their bulk expressions do not scale up to user-defined bulk structures. Providing lists as built-in bulk structures (as done in E, O_2 and other object-oriented databases) does not overcome the general lack of extensiblilty, e.g., it may still be impossible to define adequately multisets or to represent geographic data (planes, line segments etc.).

The price for the high level of abstraction and safety in DBPL programs is the inability to re-use, override, reconfigure or adapt the DBPL system code for purposes other than those foreseen at language design time. For example, the implementation of list structures should be performed preferably at the abstraction level of the layer CRDS (see Fig. 2) *within the DBPL system*, utilizing tuple-identifiers and tuples as provided by the layer SMS, inheriting SMS's concurrency and recovery mechanisms without change. Limited modifications of the generic query evaluation and optimization layer CPMS would suffice to generalize access expressions to ordered bulk types. No modifications at all seem to be required to have in addition optimized fixed-point queries over lists by re-using the layer PSMS.

To summarize, the above problems with extensions stem from the fact that traditionally built-in bulk types and query abstractions are only intended to directly support very specific *modelling* requirements and therefore are not designed to be used in the *implementation* of other generic data structures.

In our experience the main arguments against the traditional built-in approach to bulk types can be summarized as follows:

Reusability: It is difficult (often impossible) to safely re-use existing system components of the built-in system environment (buffer manager, polymorphic index structures, wait-for-graph management, clustering algorithms etc.).

Scalability: It is typically not possible to eliminate unnecessary functionality from the built-in system environment (concurrency control or recovery actions, components for recursive query evaluation).

Adaptability: Even if one has access to the implementations of the built-in bulk structures, modifications of these components (e.g. replacing B-Tree index structures by hashed structures or replacing a garbage collection algorithm) have to be performed in a "lower level" programming language with obvious negative consequences on the overall system correctness and long-term system evolution.

The apparent resistance of current DBPL *system implementations* against DBPL *language extensions* gives rise to the following important research question. Can one isolate "generic" language constructs that aid in the systematic construction of flexible DBPL processors and meet more than just a limited set of structures and algorithms foreseen at language design time? The second part of this paper addresses the relationship between DBPL language and system extensibility.

4 From Built-In To Add-On Bulk Types

In the following subsections we illustrate how well-understood bulk structures (sets and relations) can be provided as add-ons without impairing their linguistic quality (syntax), expressiveness (semantics) or optimizability.

As a concrete notation to present the add-on approach, we utilize Quest [Car89], a functional language with imperative features, strict evaluation strategies, 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, Quest turns out to be a promising platform to investigate the relationship between language and system extensibility for bulk types since it incorporates advanced abstraction mechanisms required in bulk type definition and usage.

The main purpose of this section is to introduce some of the more advanced language concepts that play a central rôle in extensible languages and to convince the reader that many implementation and optimization strategies employed for built-in bulk structures can be carried forward to add-on solutions. Important issues of extensibility and reusability are deferred until section 5.

A discussion of the *fundamental language mechanisms* (typing, binding, scoping, polymorphism, higher-order functions, type operators, type inference) employed in the programming examples can be found in [Car89]. A more detailed study of these language mechanisms in the context of database application systems is presented in [MS91].

4.1 Add-On Relation Types

The following examples present an application programmer's view of (add-on) relation types and simple relational operations in Quest:

```
Let Person = Tuple name :String  age :Int end
Let Student = Tuple name :String  age, semester :Int end
let sameName (p1, p2 :Person) :Bool = p1.name == p2.name
let persons = set.new(:Person sameName)
let students = set.new(:Student sameName)
```

The first two lines introduce name – type bindings, defining the types *Person* and *Student* as tuple types; the third line defines the function *sameName* that tests whether two tuples of type *Person* have the same value in their name attribute. This function is then used to define the key constraint for two newly created relation variables *persons* and *students* with elements of type *Person* and *Student*, respectively. Since relations are understood as keyed sets (like in DBPL), all operations on relations are imported from a module *set* and the dot-notation is used to name individual operations from this module. Here are some examples for further set operations:

```
set.insert(persons tuple let name = "Peter" let age = 32 end)
if set.member(students tuple "Peter" 23 5 end) then ... end
```

The Quest type checker enforces the obvious constraints on values of relation types: Only values of subtypes of *Person* can be inserted into a relation of *Persons*; elements of a *Persons* relation have type *Person*; key definitions for relations of *Person*s have to involve *Person* tuples; etc. Since Quest does not have built-in "knowledge" about relation types, there has to be a way to add-on such type constraints in a systematic way. For the module *set* the necessary information is defined by the interface definition *Set* that is discussed below.

The operations *empty*, *get* and *rest* constitute basic mechanisms to access individual elements of a set, one after the other. For a non-empty set, the operation *get* chooses an arbitrary element of the set while *rest* yields the set without the element returned by *get*.

```
if not(set.empty(persons)) then
 let aPerson = set.get(persons)
 let otherPersons = set.rest(persons)
end
```

A more interesting task is to capture bulk operations on sets and relations:

```
set.forEach(students fun(s:Student)
  print("Name = " <> s.name <> " Age = " conv.int(s.age) <> "\n"))

let oldStudents():Students = set.select(students fun(s:Student) s.age>28)

set.join(let select(s:Student p:Person) = tuple p.name s.semester end
  let from1 = students
  let from2 = persons
  let where(s:Student p:Person) = s.name == p.name)

set.forEach(students fun(s:Student)
  set.forEach(persons fun(p:Person)
    if s.name == p.name then print(p.name <> " " <> s.name) end))

set.some(oldStudents() fun(s:Student) {s.semester>5} ∧ {s.name = "Peter"})

let maxSemester = set.reduce(oldStudents() {fun(s:Student) s.semester} int.max)
```

The above programming examples show an iteration over the relation *students*, the definition of a view *oldStudents*, the join between two relations, a nested iteration loop, the application of an existential quantifier and the computation of an aggregate function (highest semester of all old *students*). Note that the scoping rules for the bound variables *s* and *p* support a programming style that is very close to built-in iteration constructs.

It goes beyond the scope of this paper to demonstrate that the Quest implementations of these bulk types (underlying the module *set*) could be based on B-trees, hash tables, bit sets or a combination of these data structures which could be even redefined dynamically via a special "DBA" interface.

On the other hand, the following interface *Set* of the module *set* contains all the information required by the Quest compiler to check the syntax and type rules for add-on set types.

```
interface Set
export
 T ::ALL(E ::TYPE) TYPE
 new(E ::TYPE equal(:E :E):Bool) :T(E)
 member(E ::TYPE set :T(E) element :E) :Bool
 insert(E ::TYPE set :T(E) element :E) :Ok

 empty(E ::TYPE set :T(E)) :Bool
 get(E ::TYPE set :T(E)) :E
 rest(E ::TYPE set :T(E)) :T(E)

 forEach(E ::TYPE set :T(E) stmt(:E):Ok) :Ok
 select(E ::TYPE set :T(E) pred(:E):Bool) :T(E)
 join(E1,E2,F ::TYPE  select(:E1 :E2):F
    from1 :T(E1) from2 :T(E2)  where(:E1 :E2):Bool) :T(F)
 some(E ::TYPE set :T(E) pred(:E):Bool) :Bool
end;
```

The first declaration in this interface defines an abstract type constructor *T* (identified as *set.T* from outside of this interface) that can be instantiated with an arbitrary type *E* of kind *TYPE*. For example, a *set.T(Student)* is the type of a set of students, while *set.T(Person)* denotes the type of a set of persons. *Set.new* is the only functions that creates values of type *set.T(E)* from scratch. It takes a boolean equality function *equal* between pairs of elements of type *E* as a parameter. This function defines the primary key constraint that has to be enforced at run-time by the relation implementation.

Since all functions of the interface *Set* work uniformly over sets of any element type *E*, all these polymorphic functions have a formal type parameter *E* (of kind *TYPE*). By using the name of a type parameter in the type definiton of more than one arguments of a function (e.g. *E ::TYPE set :T(E) element :E* for *insert*), one can express correctly the constraint that only values of type *E* can be inserted into a set of type *set.T(E)*. Similar constraints hold for the functions *member, get* and *rest*. Finally, the signatures of the function *forEach* demonstrates how to define functions that correspond to composite statements (e.g. `for each ... do`) in traditional programming languages.

It should be noted that the implementation of bulk types and high-level query abstractions in the add-on approach is performed in the *same language* that is also used for database application programming. This distinguishes the add-on approach from extensible database systems (like EXODUS, Genesis, Starburst or Iris; for a survey see [Sto90]) that have to resort to lower-level languages like C or C++ to implement new data structures. In particular, the implementation of add-on bulk types has to respect all typing rules and language constraints that pertain to application programs. The uniformity of system and application programming is essential for the smooth "factoring-out" of code from application programs into the application environment in case its functionality is found useful for a broader user community. Vice versa, it is often necessary to import generic code from the environment and to specialize or extend it according to application-specific requirements. Finally, due to this uniformity there is no need to define (ad hoc) interface rules between internal and external data representations or internal and external control structures.

4.2 Classifying General Data Structures

Being aware of the structural and operational richness of state-of-the-art programming language technology, an immediate objection to the add-on bulk type approach is the apparent threat of not being able to cope with the implied system add-ons required, e.g., for query optimization. As already mentioned in section 3, optimizations in database systems make heavy use of algebraic or structural peculiarities (e.g., commutativity, associativity, homogeneity, "flatness") of the data structures and operators offered by the underlying data model.

Therefore, a promising research direction towards generic optimizations for generalized data models is to isolate such "useful" properties of generalized data structures and operations and to identify partitions in a generalized bulk type and operation space for which *systematic* implementation and optimization strategies can be devised which exploit these specific properties.

For example, the three operations *empty*, *get* and *rest* in the *Set* interface (on p. 13) seem to capture a characteristic property of a large class of bulk structures, namely the ability to (sequentially) enumerate their elements. Indeed, as discussed in more detail in the next section, it is possible to define a rich declarative query language based on these access primitives.

Other researchers are investigating similar characterizations of bulk data structures based on algebraic properties of their inductively defined constructor functions (Monads [Wad90], Ringads [Tri91]). In a similar spirit, [BTBN91] shows that the expressive power of the relational algebra and of a complex object algebra with or without powerset can also be characterized by the concept of *structural recursion* on sets extended by very few programming language constructs (like tupling or conditionals). Early work on the systematic mapping from recursively defined data structures and their associated operations down to relational structures and operations (in the context of Pascal/R) can be found in [Lam84].

Formal software specification techniques may be another source for the characterization of relevant subspaces amenable to global optimizations. It is interesting to note that there is ongoing work to establish an integrated framework incorporating both, aspects of logic databases, as well as equational specifications of algebraic data types [Bee91].

As outlined in the next section, there is an important difference between the idea of (conceptual) sublanguages in the add-on approach and the traditional notion of built-in bulk type constructors with their associated query notations. This difference results from the fact that there is no artificial syntactic or linguistic barrier between optimizable query notations and non-optimizable general purpose programming constructs. Therefore, there is no need to revise the language definition, the language processor or existing application programs whenever advances in database technology enable the handling or optimization of more general data structures or query constructs.

5 Uniform Iteration Abstractions

This section refines the approach outlined in section 4.2 and captures the commonalities of a large class of bulk types by a small common set of access primitives. We also demonstrate how a very expressive query language can be systematically constructed from these access primitives. Finally, we indicate our approach to methodically achieving efficient implementations for this extended class of bulk structures.

In Tycoon we seek to explore the potential of an approach which starts with a type-safe core language supporting only low-level structures (e.g. arrays or inductively defined data types) and basic mechanisms to control mutability, sharing, clustering and persistence of these structures. Application programmers typically do not utilize this built-in

primitive functionality but make heavy use of problem-oriented bulk types (sets, relations, lists, priority queues etc.) and *standardized* iteration abstractions over these bulk structures already provided by their programming environment.

To motivate the potential of such standardized iteration abstractions, it may be helpful to compare the query that, for example, selects the top-ten managers according to the sales of all suppliers in their company with the conceptually similar query that computes the ten documents in a directory that contain the most mis-spelled words according to a given dictionary of correct words:

```
let companySales(c:Company):Real = real.sumOf(SFW(
   let select(s:Supplier):Real = turnover(s)
   let from = suppliers
   let where(s:Supplier):Bool = worksFor(s c)))
let topTenManagers = selectFirst(10
  sort(managers fun(m:Manager):Real companySales(m.company) {real. >=}))

let errorCount(t:TextFile):Int = size(SFW(
   let select(s:Supplier) = s
   let from = text.wordsOf(t)
   let where(s:String):Bool = not(member(s spellDictionary))))
let topTenFiles = selectFirst(10
  sort(documents errorCount {int. >=}))
```

Without going into details, the remarkable similarity between these two queries is due to the availability of "generic" declarative query constructs. For example, the bulk operations *add*, *count*, *select .. from .. where*, *selectFirst*, *member*, and *sort* are higher-order generic functions (defined in a Tycoon module, see Appendix A) that are uniformly applicable not only to bulk types of different element types (*Strings* or *Suppliers*) but also to a wide range of bulk types (*Relations* or *Dictionaries*).

Some of the bulk types utilized in the examples are so common (like text files, sequences of words, ordered lists, sets and relations) that they are already made available in the standard Tycoon environment. New bulk structures can be either defined from scratch (i.e. by utilizing the primitive built-in Tycoon types like tuples, variants, functions and fixed-sized arrays), or their implementation can be based on existing bulk types like lists or hashed dictionaries.

Whereas the definition of bulk types and iteration abstractions is to be regarded as a classical system programming task, application programmers typically only need to instantiate or specialize these services by appropriate type-specific information, e.g. to define application-specific bulk variables like *Suppliers*, *Managers*, *Documents* and bulk functions or associations between them like *turnover*, *worksFor* or *wordsOf* and *errorCount*.

As should be clear from the examples above, our main goal is to factor-out repeating program control patterns from specific applications into the language environment. The feasibility of this approach depends crucially on the ability to isolate highly generic access abstractions (captured by higher-order functions) that can be applied uniformly even to highly *specialized bulk structures* (represented by abstract types or abstract parameterized type constructors).

5.1 An Elementary Bulk Type Interface

As explained in section 4.2, it is highly desirable to find commonalities shared by all bulk structures that can provide the basis for substantial system support required in a database scenario. The following type definition abstractly defines a basic *protocol* for a *sequential enumeration* of the elements of an arbitrary homogeneous bulk structure:

```
Let Rec IterRep(E ::TYPE) ::TYPE = Tuple
  empty() :Bool
  get() :E
  rest() :IterRep(E)
end
```

The recursively defined type constructor *IterRep* is parameterized by the element type *E* of the iteration. For example, *IterRep(Int)* is the type of iterations over integer numbers while *IterRep(Person)* is the type of iterations over tuples of type *Person*. An *IterRep* for a given element type *E* is defined as a tuple with the three named components *empty*, *get* and *rest* which are parameterless functions. The function *empty* yields *true* if the iteration contains zero elements. The other functions are only defined for non-empty iterations and split an iteration into two parts: *get()*

is a value of the iteration element type and *rest()* is again an iteration of type *IterRep(E)*, however, without the value returned by *get()*.

In other words, any structure for which a value of type *IterRep* can be constructed, has "bulk" character. Intuitively, this can be understood by the fact that if *x* is of type *IterRep(E)* then the three function components (*x.empty*, *x.get*, *x.rest*) provide a mapping from *x* to (possibly infinite) homogeneous lists:

[x.get() x.rest().get() x.rest().rest().get() ...]

This can be exemplified by the polymorphic function *elements* that maps an array *arr* of an arbitrary element type *E* to a recursively defined value of type *IterRep(E)*, i.e. arrays are bulk types because one can sequentially enumerate their elements;

```
let elements(E ::TYPE arr :Array(E)) :IterRep(E) =
 begin
  let rec elementsFrom(index :Int) :IterRep(E) =
     tuple
      let empty() :Bool = index > size(arr)
      let get() :E = arr [index]
      let rest() :IterRep(E) = elementsFrom(index+1)
     end
  elementsFrom(1)
 end
```

The definition of type *IterRep* does not only characterize "stored" bulk types like lists, arrays, sets, relations, dictionaries or files, but it also captures related abstractions like predicatively defined subcollections (views), derived collections (e.g. grandparents or ancestors derived from parents), streams, pipes or channels in operating systems or simple enumerations or intervals (e.g. "all numbers in the range from 1 to 999").

5.2 Generalized Iteration Abstractions

Based on elementary bulk access defined by the standardized protocol of abstract sequential access, we can now define higher-level, declarative *iteration abstractions*. These abstractions can be either algebra-like operations (union, difference, selection, cartesian product or join) or calculus-like expressions involving (nested) existentially or universally quantified expressions. Technically speaking, iteration abstractions do not work directly on bulk structures but on values of type *IterRep* derived from bulk structures:

```
let persons :Array(Person) = ...
let oldPersons :Array(Person) = ...
let allPersons = iter.append(elements(persons) elements(oldPersons))
iter.join(
  let select(p1,p2:Person) = tuple p1.name p1.age p2.age end
  let from1 = elements(persons)
  let from2 = elements(oldPersons)
  let where(p1,p2:Person) = {p1.name == p2.name} ∧ {p1.age > p2.age})
```

What are the operations we expect to be defined over general bulk structures? These operations should allow (see also Appendix A)

- the mapping of iterations to iterations of the same or another element type (e.g. *map, select, select .. from .. where*);
- the quantification over bulk structures by reducing iterations to values of the primitive types (e.g. *size, reduce, fold, some, all*);
- the combination of iterations into a single new iteration (e.g. *append*);
- the iteration with side-effects over each element of an iteration (e.g. *forEach, forDo, untilDo, whileDo*);

- the "associative" selection of *individual* elements of a iterations based on its properties expressed by a predicate analogous to the "de-setting" operation, as found, for example, in Adaplex [SFL83] or DBPL (*the, theOnly, theFirst, theLast*).

Similar iteration abstractions for specialized bulk types are discussed, for example, in [OBBT89, AM87, SDDS86]. Note that the module *iter* in the Tycoon environment provides a very general and highly reusable functionality. For example, suppose an application has to operate on stacks of rectangles and lists of line segments, the operations in the module *iter* provide a building block for a declarative geometric "query language", e.g. to formulate a query to find the topmost rectangle on a stack that has the maximum number of intersections with a given set of line segments.

Even if the underlying data structures only provide simple sequential access and no type-specific information, the nesting of iteration abstractions and the existence of binary operations on bulk structures gives rise to *alternative* evaluation strategies for a given "query". In the spirit of database query optimizers (or loop optimization techniques in programming languages) important optimizations can be applied to these general data structures (filter promotion, quantifier exchange, constant propagation, memorizing of repeating subqueries etc.). At the same time it is possible to provide optimized implementations for join operations (e.g. by sort and merge) without perturbating the declarative "query" style in application programs.

To our understanding, the main reason why none of the existing programming languages provides these optimizations for add-on iteration abstractions lies in the lack of an adequate *implementation technology* for such *global*, run-time optimizations: most optimizations rely on information about a complete query expression (including selection predicates and user-defined functions) or even on parameter values that are only determined at run-time. Therefore, traditional (local, incremental) compilation technologies fall short for this particular class of problems.

5.3 Refined Bulk Type Interfaces

The sequential bulk type interface presented in section 5.1 can be used as a good starting point to study refined bulk type interfaces with particular support for bulk optimizations. Our experience with the query optimization problem in Pascal/R, Modula/R and DBPL indicates that the *IterRep* protocol definition needs to be extended by three kinds of information to enable optimizations that are competitive with today's commercially available systems.

- Information about alternative *access support* (sequential access, positionally indexed access, value indexed access, ordered access (for range queries), hierarchical access)[2]. This implies that an individual bulk structure may possess *multiple* access paths with substantially different operational behaviour and that even for a single declarative iteration abstraction a choice between these implementations becomes possible.

- Cost functions for access mechanisms (computing time, number of data elements visited) to guide the query transformation process.

- Statistical information on the bulk structure cardinality, the distribution of attribute values or the correlation between attribute values. Some of this information is (conceptually) derivable from knowledge about the application domain (e.g. the maximum number of students in a lecture, the set of possible *age* attribute values, data dependencies between zip codes and city names), others have to be computed at run-time.

Technically, this information could be cast into an extended *IterRep* type specification that has to be met by implementors of add-on bulk types. Since query optimization is an "optional" feature, it is appropriate to allow "do not know" or default values for these additional iterator attributes (as it is done in extensible query optimizers). System experimentation is underway to understand how to distribute this information systematically over the Tycoon environment while maintaining modularity and extensibility.

We do not want to leave this discussion without mentioning the problems arising from destructive updates (e.g., within *forDo* loops or as side-effects of user-defined functions defining selection expressions). In [MOS91] we propose extensions of polymorphic type systems to capture not only structural aspects of data values, but also information about the *locality* of (mutable) data objects with identity. This allows one to utilize type checking and type inference technology to reason about the scope of side-effects and to identify referentially transparent ("purely functional") subexpressions that permit effective query rewriting techniques.

[2] In [MS91], these access mechanisms are characterized by type constructors like it was done for the sequential case in section 5.1. This approach also allows to capture the commonalities between access mechanisms by an (inferred) subtype relationship.

6 Research Issues In Generic Bulk Data Support

The main purpose of this paper is to discuss the rather complex interaction between language and system considerations involved in bulk type definition and support. We identify *elementary* and *advanced* requirements for bulk types, each of which can be satisfied either by specialized built-in language mechanisms or generalized open system architectures.

Recent advances towards expressive type systems and highly effective compilation schemes may allow the *elementary* requirements for bulk data storage (including persistence management) to be satisfied without resorting to the built-in approach. Given state-of-the-art language technology, a high investment in built-in bulk types (as exemplified by DBPL) can only be justified by the support it provides for *advanced* requirements like generalized associative queries or integrity constraint management in multi-user environments. We finally reported on our current work that aims to satisfy even these advanced requirements by user-definable add-ons to generic core languages focussing on linguistic and architectural aspects, since important component technology seems to be already available in extensible database systems.

Clearly there remain many open research issues for further theoretical and experimental work. What is a suitable formal framework for data models supporting generalized bulk types? Is it necessary to equip DBPLs with specific *syntactic* support for a concise notation of bulk data manipulations (e.g. list comprehensions)? Is there a taxonomy for general optimization strategies (algebraic vs. operational, static vs. dynamic, deterministic vs. probabilistic)? Where should the meta-information required for optimizations come from (static program analysis, program annotations via pragmas, access to program specifications, compile-time or run-time reflection)? Can standardized iteration abstractions be exploited for program verification tasks?

Finally, one could speculate that the experience gained in supporting add-on bulk types may be relevant also for other kinds of substantial language extensions such as the support of "declarative" constraint management in the presence of destructive updates.

Acknowledgements

The authors would like to thank Malcolm Atkinson, Catrieel Beeri and Peter Buneman for stimulating comments on this work and the referees for their constructive remarks on an earlier version of this paper that helped in streamlining the flow of argumentation.

References

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

[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.

[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.

[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.

[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*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An Extensible Database Management System. *ACM Transactions on Database Systems*, 14(11):1711–1729, November 1988.

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

[Bee91] C. Beeri. A Logical Framework for Object-Oriented Databases. (private communication), June 1991.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991.

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

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[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.

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

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data Construction with Recursive Set Expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

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

[HS89] R. Hull and J. Su. On Bulk Data Type Constructors and Manipulation Primitives: A Framework for Analyzing Expressive Power and Complexity. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 396–410, June 1989.

[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, Departement Informatik, ETH Zürich, Switzerland, February 1983.

[Lam84] W. Lamersdorf. Recursive Data Models for Non-Conventional Database Applications. In *Proc. Intern. IEEE Conf. on Data Engineering*, Los Angeles, April 1984.

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

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

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

[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.

[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*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

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

[MS91] F. Matthes and J.W. Schmidt. Towards Database Application Systems: Types, Kinds and Other Open Invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

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

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

[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.

[Sch78] J.W. Schmidt. Type Concepts for Database Definition. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*. Academic Press, 1978.

[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 112-88, 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.

[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.

[SNR90] M. Santo, L. Nigro, and W. Russo. Programmer-Defined Control Abstractions in Modula-2. *Computer Languages*, 15(3), October 1990.

[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*, volume 504 of *Lecture Notes in Computer Science*, April 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.

[Tri91] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991.

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

[WSK81] A.L.. Wasserman, D.D. Sheretz, and M.L. Kerstin. Revised Report on the Programming Language PLAIN. *ACM SIGPLAN Notices*, 16(5):59–80, May 1981.

A Iteration Abstractions for Add-On Bulk Types

The following fragment taken from a prototypical Tycoon iterator interface gives some examples of useful iteration abstractions that are uniformly applicable to instances of a wide range of bulk types (see Sec. 5).

```
interface Iter
import :IterRep
export
 Def T = IterRep_T
 error :Exception(Ok)
  (* Raised if any of the following operations cannot be carried out. *)

 select(E ::TYPE iter :T(E) p(:E):Bool) :T(E)
  (* Return an iteration over the elements in iter that fulfil p. *)
 append(E ::TYPE first, second :T(E)) :T(E)
  (* Return an iteration over each e in first and then each e in second. *)
 map(E, F ::TYPE iter :T(E) f(:E):F) :T(F)
  (* Return an iteration over f(e) for each e in iter. *)
 size(E ::TYPE iter :T(E)) :Int
  (* Return the number of elements of an iteration. *)
 fold(E, F ::TYPE iter :T(E) f(:E):F g(:F :F):F) :F
  (* Reduce f(e) for each e in iter by g. Raise error if iter is empty. *)
 reduce(E ::TYPE iter :T(E) g(:E :E):E) :E
  (* Reduce iter by g. Raise error if empty. *)
 flatten(E ::TYPE iter :T(T(E))) :T(E)
  (* Return an iteration over each e2 in each e1 in iter. *)
 unnest(E, F ::TYPE iter :T(E) f(:E) :T(F)) :T(F)
  (* flatten(map(iter f)). *)
 some, all(E ::TYPE iter :T(E) p(:E):Bool) :Bool
  (* Does p(e) hold for some element (resp. all elements) in iter? *)
 join(E1,E2,F ::TYPE  select(:E1 :E2):F
     from1 :T(E1) from2 :T(E2)  where(:E1 :E2):Bool) :T(F)
  (* Join from1 with from2. *)
 any(E ::TYPE iter :T(E) p(:E):Bool) :E
  (* Choose any element in iter that fulfils p.
     Raise error if no such element exists. *)
 selectFirst(E ::TYPE n :Int iter :T(E)) :T(E)
  (* Return first min{n,size(iter)} elements of iter. *)
 forEach(E ::TYPE iter :T(E) statement(:E):Ok) :Ok
  (* Perform statement for each e in iter. *)
 forDo, untilDo, whileDo
    (E ::TYPE iter :T(E) p(:E):Bool statement(:E):Ok) :Ok
  (* Equivalent to
     forEach(select(iter p))
     forEach(selectUpto(iter p))
     forEach(selectBefore(iter fun(e:E) not(p(e))))  *)
 selectBefore, selectAfter, selectUpto, selectStartingWith
    (E ::TYPE iter :T(E) p(:E):Bool) :T(E)
  (* Let i be the index of the first element in iter that fulfils p.
     Return an iteration over all elements with an index j such that
     j<i, j>i, j<=i, j>=i *)
 sort(E ::TYPE iter :T(E) order(:E :E):Int) :T(E)
  (* Order has to define a linear order on values of type E. *)
end
```