Chapter 1.4
Bulk Types and Iterators

Florian Matthes

Technical University Hamburg-Harburg
Harburger Schloßstraße 20
D-21071 Hamburg, Germany

A bulk type describes a homogeneous collection of values. These values are
described by another type which is given as a parameter to the bulk type
constructor. Many bulk types such as sets, lists and relations are already
familiar and widely used in data-intensive applications. Additional bulk types,
like bags and multiset, can be defined either by system providers or by ex-
pert users. Languages with multiple bulk types have a higher potential for
problem-oriented data modelling. In particular, by using bulk types to de-
scribe data that resides in legacy databases one gains access to that data in
a convenient, consistent and safe manner.

The regular structure of bulk type instances can be exploited to pro-
vide high-level iteration abstractions for declarative access to bulk data. An
important topic in database language research is the development of strongly-
typed language constructs like queries, views or iterators that generalize the
successful iteration abstractions of relational algebra, relational calculus or
SQL to other bulk types. Problem-oriented iteration abstractions not only
increase programmer productivity but they can also be exploited to increase
system performance by enabling query optimization and tailored storage al-
location schemes. Furthermore, generic bulk types and iteration abstractions
greatly simplify the construction of user-oriented tools like database browsers
or ad-hoc query interfaces.

Several members of the FIDE consortium explored alternative strategies
for the provision of bulk types. DBPL [14] (synopsis in Chapter 2.1.2), O2
[5] and Fibonacci (presented in Chapter 1.1.2) are all equipped with type-
complete bulk type constructors and a rich built-in query language over com-
plex values of these bulk types. Another strategy to provide bulk types in a
programming language setting is to reduce well-known complex-object query
languages to a small set of primitive operations and iteration abstractions
expressed as first-class abstract data types with polymorphic higher-order
functions [2, 17, 7, 15, 6, 10].

Motivated by the substantial progress in programming language generic-
ity and expressiveness as well as recent advances in object store technology,
members of the FIDE consortium also investigated an extensible approach
to bulk data support [12] (synopsis in Chapter 1.2.2) and [4]. This approach
aims at generalizing the expertise in language design and database system
construction from a fixed set of bulk type constructors to larger families of
collection types and to a broader range of services attached to collection types.
such as integrity checking, data visualization and viewing mechanisms. This add-on approach to bulk type and iterator support is based on the following architectural and conceptual separation of tasks:

- Data-model-independent object stores provide persistence, concurrency control and recovery abstractions over relatively simple, graph-structured data. The object store interface has to find a balance between the need for efficient bulk data access and the desire to support the implementation of a wide range of bulk data structures. These issues are discussed in more detail in Chapter 2.2.1 and 2.2.2.

- Abstract machine languages implemented on top of these object stores are used as well-defined, minimal interfaces for (bulk) data access and data manipulation. These canonical machine languages are generated by language-specific compilers and provide an ideal starting point for integrated program and query optimization as well as multi-platform language implementations (see also Chapter 2.1.3).

- Persistent higher-order programming languages provide a uniform, strongly-typed programming and interactive interface to volatile and persistent, small-scale and large-scale bulk data. As described in Chapters 1.1.1 to 1.1.3, all FIDE languages are based on a common set of design principles like uniform naming, strong typing, flexible R-value and L-value binding, static scoping, orthogonal persistence and type orthogonality that also apply to the design and implementation of bulk types and iteration abstractions.

- Type-safe, generic bulk type libraries implemented in the FIDE languages provide application programmers with prefabricated software components for bulk data storage, iteration abstraction, bulk data input, formatted data display, searching, sorting, integrity maintenance, etc. These bulk type libraries are typically pre-populated by the language designer or constructed by expert programmers. Significant effort has been devoted at both Hamburg and Glasgow Universities to the construction of extensible, but nevertheless easily understandable and efficient bulk type libraries [3, 13].

- Extensible grammars, as developed in [8] allow library designers to provide tailored syntactic forms for library-defined bulk types without compromising the simplicity, optimizability and security of the core language. As described in more detail in Chapter 3.2.3, classical data description and data manipulation languages can then be captured directly via types, type constructors and polymorphic (higher-order) functions exported from bulk type libraries.

The first paper in this Chapter investigates typing issues of bulk types. The semantics of some bulk data models involves user-defined attributes like element equality, ordering or other domain predicates. In this paper, a statically-decidable typechecking scheme is presented where such attributes are part of
the bulk type which makes it possible to reject semantically incorrect bulk operations already at compile-time.

The second paper raises the language and system design question whether there should be built-in bulk types in database programming languages at all. 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. The advantages and disadvantages of both approaches are presented by distinguishing between elementary and advanced bulk type support.

The last paper in this Chapter presents viewing mechanisms on type-complete database objects and compares their semantics with object extension and role manipulation operations. All of these operations are identity-preserving. The ability to provide multiple views on a database object and to modify the structure of an object without invalidating its identity is crucial to the management of bulk class extents and of class relationships [1]. This work is also related to the concept of viewers developed in the FIDE project and presented in [16].

Finally, several other papers in this collection should be referenced that also make contributions to the research topic of bulk types and iterators.

The paper [9] (summarized in Chapter 1.1.3) investigates which subtyping rules should apply to bulk type constructors. For example, under which circumstances is it safe to treat the type List(Student) as a subtype of List(Person), assuming that Student is a subtype of Person?

Chapter 1.2.1 includes an explanation is given on how to implement type-dependent bulk operations like the relational natural join using type-directed reflective programming techniques.

Query optimization techniques for database programming languages where bulk expressions can be nested and mixed freely with application code are presented in Chapter 2.3.4 and Chapter 2.3.5 (synopsis of [11]), respectively.

Finally, the integration of legacy bulk data maintained outside the scope of integrated database languages (in files or commercial databases) as typed bulk data with iteration abstractions is described in Chapter 3.3.1 and 3.3.2.

References


