

Rationale and Design of BULK* †

Steve Rozen^{‡§}
steve@soi.com

Dennis Shasha[§]
shasha@cs.nyu.edu

[‡]Software Options, Inc.
22 Hilliard Street
Cambridge MA 02138
USA

[§]Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York NY 10012
USA

Abstract

BULK is a very-high-level persistent programming language and environment for prototyping and implementing database applications. BULK provides sets and sequences as primitive type constructors, provides high-level operations on them, and allows programmers to define application-oriented bulk types, e.g. syntax trees, bond portfolios, or (geographic) maps.

BULK encourages separation of correctness and efficiency concerns by distinguishing logical type from representation. BULK supports a three-step development paradigm consisting of (i) prototyping, (ii) intensive analysis, optimization, and data structure selection by the compiler to achieve efficiency, and (iii) if efficiency is still inadequate, hot-spot refinement [CGK89]. (In hot-spot refinement developers remove performance bottlenecks by providing the compiler with more information, by directing its optimization efforts, or by re-implementation.) Step (i) focuses on correctness, steps (ii) and (iii) on efficiency. Our goal is an implementation that can usually achieve acceptable efficiency by step (ii) and that provides a tractable interface for hot-spot refinement.

1 Introduction

A major service offered by file systems and DBMSs is storage of *persistent* data, data which persists between executions of a program, and which many programs can share. Programmers in conventional programming languages generally rely on a file system or database management system (DBMS) for this. In addition, file systems and DBMSs also provide a *bulk data type*, a data type whose instances contain a dynamically varying and potentially large number of elements. Sets, sequences, graphs, and relations are familiar examples of bulk data types.

Unfortunately, the data types and representations of the programming language and a DBMS or file system rarely coincide, forcing the programmer to invent two representations and write translation code between them. No compile-time support for type checking spans the loose association of programming language and DBMS; types are checked when data is read from or written to the DBMS, if at all. Furthermore, the set-oriented operations in, for example, relational databases, have no counterpart in most programming languages. The discontinuity in type system and computational model between programming language and DBMS is often referred to as an *impedance mismatch* [CM84].

Database programming languages integrate the facilities of a general purpose programming language and a DBMS, thereby doing away with the impedance mismatch. Examples include ADAPLEX [Com83],

*This work was supported with funds provided by ONR grants N00014-90-J-1110 and N00014-91-J-1472, by NSF grant IRI-89-01699, and by DARPA under contract N00014-85-C-0710 with ONR.

†Appeared in *Database Programming Languages: Proceedings of the Third International Workshop* (Nafplion). Morgan Kaufman, Aug. 1991.

DBPL [EEK⁺85], E [RC87, RC88] FAD [DKV87], Galileo [ACO85], Machiavelli [OBBT89], Modula/R [KMP⁺83], Napier [MAD87], OPAL [CM84, MS87], Pascal/R [Sch77], PS-Algol [Atk83], and many others. These examples are imperative or applicative in flavor; there are also many approaches based on logic programming. Both [AB87] and [OSD86] analyze design issues in database programming languages.

This paper introduces a new database programming language called BULK, states the rationale for its existence, and sketches an approach to optimization based on static analysis.¹

2 How BULK Is Different

BULK is designed to address difficulties arising in the development of database applications using current commercial technology: stand-alone DBMSs and programming languages such as C, Pascal, or Cobol. In [RS89] we analyze these difficulties in the context of an industrial development effort. From this analysis emerged the following desiderata:

- a single logical representation for persistent and volatile data,
- set-level operations for volatile as well as persistent data,
- a data encapsulation mechanism with inheritance,
- bulk types extending beyond first-normal-form relations,
- alternative representations of bulk types, and
- a rich set of tuning options to achieve acceptable efficiency.

These desiderata suggest that database application development would be well-served by a persistent programming language and environment with

- a small set of built-in bulk data types (e.g. sets and sequences) with highly general, set-level operations,
- data encapsulation, inheritance, and the ability to define application-oriented bulk data types based on built-in types,
- static type checking for efficiency and safety,
- a strong separation of logical type from representation to encourage separation of correctness from efficiency concerns, and
- support for a spiral application-development paradigm consisting of three steps:
 1. prototyping,
 2. intensive static analysis and profiling, leading to cost-based optimization and data structure selection by the compiler, and
 3. programmer-directed tuning or reimplementing of remaining efficiency bottlenecks.

BULK is our attempt to design such a language and environment.

We chose SETL [SDDS86] as a starting point for our effort. SETL offers a conjunction of widely understood constructs—an imperative programming language and mathematical (finite) sets, sequences, and relations. SETL formed the basis of a number of approaches to specification and prototyping. For example, SETL

- influenced the design of a knowledge-based software development environment [Rea85],
- has been augmented with fixed-point operators and used as basis for a program-transformation system [PH87], and

¹We use “BULK” to refer to both the programming language and the environment, and distinguish the two only when necessary to avoid ambiguity.

- was used to create a working prototype implementation of a validated Ada compiler [KS84].

BULK inherits SETL’s primitive bulk data type constructors (all describing objects of arbitrary and dynamic size). These are fundamentally sets and sequences, though in addition special syntactic support is extended to binary relations (i.e. sets of sequences of length two) that are functions. BULK offers a highly uniform treatment of sets (and sequences), cited in [Ban88] as “one of the major challenge[s] facing designers of object-oriented database systems.” This uniformity is due to the following:

- Sets and sequences have the same operations, regardless of their implementation. This contrasts with many object-oriented programming languages, where there are a number of collection classes implementing set-like types, all associated with different operations.
- Instances of sets and sequences are values like any other. Thus they may be persistent or volatile, may be subvalues of other values without restriction,² and may be arguments and return values of functions.
- Sets and sequences share similar *former* syntax.³ For example, regardless of whether X is a set or a sequence of integers, the expression

$$\{x * x : x \text{ in } X \text{ st } x > 10\}$$

denotes the set of elements formed by squaring those integers in X that are greater than 10. A similar notation forms sequences, i.e.

$$[x * x : x \text{ in } X \text{ st } x > 10].$$

Using BULK’s data abstraction mechanism (described below) programmers can define additional, application-oriented bulk data types based on sets and sequences.

Like SETL, BULK is value-oriented; there is no explicit heap storage allocation and there are no pointers. Thus, for example, after the instructions

$$a \leftarrow \{1..100\}; b \leftarrow a; a \leftarrow a - \{20\};$$

b ’s value is $\{1..100\}$. Unlike SETL, BULK supports persistent values and data abstraction in the form of classes resembling C++ classes [Str86].

We believe that a strong separation of the notion of logical type from the notion of representation is necessary for a conjunction of simplicity and efficiency in a very-high-level language. For example, BULK’s implementation architecture takes advantage of multiple implementations to statically select low-cost disk-based data structures and query plans. (SETL offers a scheme for automatically or manually choosing among a fixed set of representations for a single logical type [DGLS79, SSS81].) Few database programming languages make this separation, but in DBMSs built-in bulk data types often have multiple implementations, e.g. through the use of various primary and secondary access structures. However, unlike most DBMSs, BULK extends the separation of type and representation to user-defined types, and allows programmers to define new bulk types with multiple implementations. As an example, consider an ordered tree with two operations: pre-order traversal and ordered scan of the leaves. Those tree instances subject to the linear scan might include links from leaf to leaf that would be superfluous in instances subject only to pre-order traversal.

Finally, still building on the separation of logical type from representation, BULK programmers can achieve even greater efficiency by *hot-spot refinement* [CGK89]. Hot-spot refinement starts from the most efficient implementation that the compiler, unaided, can produce. Programmers then identify performance bottlenecks (hot-spots) and specify a more efficient implementation for them. Support for hot-spot refinement is important because developers cannot always afford to throw away and re-code an entire working

²The R²D² [KLW87, Lin88] database system provides a model with very uniform treatment of sets and sequences used as subvalues in an unrestricted way. However, since R²D² is a database system rather than a programming language, there are no volatile sets or sequences.

³Similar constructs have been found useful in functional languages such as Miranda, where they are called ZF expressions or list (or set) comprehensions [Tur82].

prototype.⁴ But if a language provides no support for hot-spot refinement the hot-spots become poorly structured and the system as a whole develops many implementation dependencies, making evolutionary change and future refinement more difficult.

BULK supports hot-spot refinement in two ways.

- The programmer can provide more information to the compiler, in the form of
 - expected execution or data statistics, or
 - directives to guide the optimizer or to use a particular implementation, e.g. to maintain a disk-based set in some particular order.

For this purpose, the BULK language provides an extensible set of pragmas. Thus the programmer can specify the representation to be used if those selected automatically are unsatisfactory.

- If reimplementing is necessary, BULK's alternative class implementation mechanism helps organize the reimplementing. Alternative class implementations can be used by the compiler when it automatically selects data structures, provided the costs associated with operations on the alternative implementation are available. When developers are forced to resort to a lower-level languages for alternative implementations BULK is designed give them a first approximation to the code in a lower level language, and verify type consistency across the interface. Pragmas can provide BULK with the cost properties of the alternative implementation. BULK also accepts information about the foreign function, for example, whether it is pure, whether it is associative or commutative, and what the programmer estimates its cost to be.

The next section is an introduction to BULK through a running example.

3 Example

Our examples are implementation fragments from the bond⁵ information system discussed in [RS89].

User-Defined ADTs Figure 1 shows an abstract data type definition for bonds, in this example the class `bond`. A class is a user-defined type that specifies a representation of instances of the type and an implementation of operations on instances.⁶

- The representation of class instances is specified by the *data members* of the class; in the `bond` class these are `coupon`, `maturity`, and `history-data`.
- Operations on class instances are specified by *member functions*; in `bond` these are `history`, `price-to-yield`, `enter-quote`, `most-recent-quote`, and `average-price`. The member function `price-to-yield` is not implemented in `bond`, but its presence in the definition of `bond` indicates that `price-to-yield` must be defined in subclasses (see below) of `bond`.

The value of a data member can vary from instance to instance, whereas the value of a member function is constant for all instances of a class.

A *member* (data member or member function) that appears prior to the key-word `private` is visible outside the class; such a member is termed *public*. A member after `private` is *private*, and visible only within the member functions of the class. Within a member function, other members are simply referred to by name. For example, the identifier `history-data` in the member function `history` refers to the data member of that name.

⁴In [RS89] we present an example of the use of hot-spot refinement in a real-world application. In this application the developers duplicated persistent data in shared memory for faster access, a difficult job that was unsupported by either programming language or DBMS.

⁵We mean financial bonds, as in "stocks and bonds".

⁶Unlike some other database programming languages, e.g. ADAPLEX [Com83], a class is not associated with an extent; i.e. there is no set automatically containing all instances of a class. The reason is that BULK programs are designed to express an entire application, not just database retrieval. In general we may want to create instances that do not persist or that are invisible to most users. An example would be a system using hypothetical bonds, where most of the hypothetical bonds are thrown away, and only a few are kept for additional work by a restricted set of users. Indexing on all instances of a class was rejected for similar reasons in [MS86].

```

class bond
  coupon : float;
  maturity : date;

  func history() map(date, seq(float, float))
    history-data;
  end history;

  func price-to-yield(float, date) float;

  func enter-quote(day : date, price : float)
    history-data(day) <- [price, price-to-yield(price, day)];
  end enter-quote;

  func most-recent-quote() seq(date, seq(float, float))
    let d be max(domain history-data);
    [d, history-data(d)];
  end most-recent-quote;

  func average-price(d : date) float
    let prices be [x(2)(1) : x in history-data st x(1) >= d];
    reduce(+, 0, prices) / #prices;
  end average-price;

private
  history-data : map(date, seq(float, float));
end bond;

```

Figure 1: bond Class Definition

```

class treasury : subclass of bond
  func price-to-yield(p : float, d : date) float
    foreign [coupon, maturity];
  end price-to-yield;
end treasury

class corporate : subclass of bond
  issuer : string;
  rating : string;
  func price-to-yield(p : float, d : date) float
    foreign [coupon, maturity];
  end price-to-yield;
end corporate;

```

Figure 2: Subclasses of bond

High Level Operations Figure 1 also illustrates operations on *maps* in BULK. In this case a map is a finite function represented extensionally. The member `history-data` in `bond` is such a map. For a functional map \mathcal{M} with domain element x the syntax $\mathcal{M}(x)$ denotes the corresponding range element. In the definition of `most-recent-quote`, the expression

```
max(domain history-data)
```

yields the maximum date in the domain of `history-data`, and

```
history-data(d)
```

yields the quote for that date.

In `most-recent-quote` and `enter-quote` the square brackets `[,]` denote the variadic function returning the sequence composed of its arguments; e.g. `[a, b, c]` denotes the sequence `abc`. Thus, in `enter-quote` the range element of `history-data` associated with `day` is set to the pair

```
[price, price-to-yield(price, day)].
```

Additional set and sequence operators are illustrated below.

Note that, conceptually, the `history` member function returns a copy of `history-data`. However, a BULK implementation can detect applications of `history` where the member `history-data` is not modified while the value returned by `history` is still live.⁷ In this case the implementation can use a reference to `history-data` as `history`'s return value.

The member function `average-price` is defined to return the average price of a bond over all days on or after `d`. The symbol `#` used in `average-price` is a prefix operator that returns the number of elements in its argument. The expression

```
[x(2)(1) : x in history-data st x(1) >= d]
```

denotes the sequence of prices quoted on or after `d`. This is an example of a sequence former, as discussed in Section 2. Here `x(2)` denotes a range element of `history-data`, and since `x(2)` is itself a pair, `x(2)(1)` denotes its first element, a price.

Inheritance In Figure 2, the classes `treasury` and `corporate` are defined as subclasses of `bond`. The class `corporate` has two data members, `issuer` and `rating` (both strings of arbitrary length), not present in `bond`. Both `treasury` and `corporate` implicitly have all the members of `bond`. However, each has a different implementation of `price-to-yield`. The subclass relationship is useful in two ways:

⁷This can be determined from conventional du-chains. More ambitious techniques that move unavoidable copy operations to infrequently executed parts of a program are presented in [Mor91].

```

class bond<cache-most-recent>

  func enter-quote(day : date, price : float)
    if undefined(latest-quote-date) then
      latest-quote-date <- day;
    if day >= latest-quote-date then
      latest-quote-date <- day;
      latest-quote <- [price, price-to-yield(price, day)];
    end if
    history-data(day) <- [price, price-to-yield(price, day)];
  end enter-quote;

  func most-recent-quote() seq(date, seq(float, float))
    [latest-quote-date, latest-quote];
  end most-recent-quote;

private
  history-data      : map(date, seq(float, float));
  latest-quote-date : date;
  latest-quote      : seq(float, float);
end bond;

```

Figure 3: Programmer-Defined Alternative Implementation

- It can represent is-a relationships, as in the example, where each `corporate` is a `bond`.
- It can support the extension of existing software to handle new cases. For example, if a new kind of corporate bond were created that could be exchanged for a specific number of shares of stock, this kind of bond might be created as a subclass of `corporate`.

Integration with Other Languages Also in Figure 2, for both `corporate` and `treasury`, `price-to-yield` is implemented in another programming language, whence the key-word `foreign` in these function definitions. Following `foreign` is a sequence of values to be passed to the foreign function in addition to the explicit parameter. The BULK programming environment must determine that values provided to and received from a foreign function have a representation consistent with those expected by and returned from the foreign function.

Alternative Class Implementations In Figure 3 the syntax `<cache-most-recent>` indicates that this is an alternative implementation for `bond`. It is required that alternative implementations have the same public members. The original implementation (in this case in Figure 1) is called the *canonical implementation*. Here, the `cache-most-recent` implementation of `bond` is understood to have all the public and private members of `bond`, and in addition two other private members, `latest-quote-date` and `latest-quote`. Also, the functions `enter-quote` and `most-recent-quote` have new implementations. The net result is that if `most-recent-quote` is evaluated much more often than `enter-quote`, the `cache-most-recent` implementation of `bond` will be more efficient than the canonical implementation. In Section 4 below we discuss why alternative class implementations are a good idea, and how inheritance and alternative implementations interact.

Persistence Figure 4 illustrates definitions of persistent values. On the first line, the type `bond-id` is defined to be a renaming of the type `atom`. The only operations on `atoms` are equality and the generation of a heretofore unknown `atom` value. Thus, as in this example, atoms often act as object identifiers. The

```

type bond-id atom;

var known-bond : map(bond-id, bond) = {};

var bond-group : map(string, set(bond-id)) = {};

```

Figure 4: Persistent Data Definitions

```

persistent var known-bond;
persistent var bond-group;

let g be read-string();
let d be read-date();
transaction
  print(
    #{known-bond(x) : x in bond-group(g)
      st
        known-bond(x).most-recent-quote()(2)(1) >= 1.1 * known-bond(x).average-price(d)
    });
end transaction;

```

Figure 5: A Program Using Persistent Data (Transaction 1)

next two lines declare `known-bond` to be a map from `bond-ids` to `bonds`, initially the empty map. The last line of Figure 4 declares `bond-group` to be an initially empty map from strings to sets of `bond-ids`.

The definitions in Figures 1, 2, 3, and 4 would be entered into a *persistent name space* using the BULK environment. A persistent name space is a function from identifiers to types, values, and “programs”, i.e. compilable units of BULK code. BULK code in a persistent name space can reference type and value identifiers in the same name space by prefixing their declarations with the key-word `persistent`, as shown in Figure 5. Therefore BULK object code depends not only on source code, but also on a persistent name space. The source program may have different object code for different persistent name spaces, or even no object code, if the program is erroneous in a particular persistent name space, e.g. because of a type error.

Transactions Operations on persistent data that are bracketed by `transaction` and `end transaction` (which must nest with other lexical constructs) are atomic with respect to recovery and concurrency. In Figure 5 there is a transaction that, given a bond group, `g`, and a date, `d`, prints out the number of bonds in `g` that are worth at least 10% more than their average price since `d`.

4 Alternative Class Implementations

Don't inheritance mechanisms such as those in Smalltalk-80 [GR83] or C++ suffice? For example, an abstract superclass (Smalltalk-80 terminology) or a virtual base class (C++ terminology) can represent a logical type, and subclasses (derived classes in C++) can represent alternative implementations.

However, in practice it seems rare that class definers actually provide alternative implementations; instead they provide classes with similar but different method protocols, and this inhibits simple replacement of one implementation by another. As two examples among the Smalltalk-80 `Collection` classes, consider the task of changing the representation of a `SequenceableCollection` from `OrderedCollection` to `LinkedList`, or of converting a `Dictionary` keyed by integers to an `Array`. Perhaps subclasses-as-alternative-implementations is rarely used because the subclass relationship by itself does not indicate when two classes represent the same ADT.

```

transaction
  for b in range known-bond order by b.maturity do
    let s be [q in b.history() order by q(1),>];
    print(b.maturity, b.coupon, s(1..3));
  end for;
end transaction;

```

Figure 6: Transaction 2

Furthermore, even if used, subclasses-as-alternative-implementations forces the programmer to make a representation decision early, perhaps even before all uses of the data are known, resulting in over-specification. Even in a dynamically-typed language the programmer must make a representation decision, if only what class to send `new` to.

Finally, there is interference between subclasses-as-alternative-implementations and using inheritance to express is-a relationships. For example, suppose a has two subclasses, a' and a'' , as its two implementations, and suppose class b is-a a . To express this with subclasses, one could make b a subclass of a , but then neither a' nor a'' would be a basis for an implementation of b . On the other hand, making b a subclass of e.g. a' means that the implementation a'' is not a basis for an implementation of b .

All-in-all, to use the distinction of [Str88], a subclass mechanism *enables*, but does not *support* alternative implementations.

With explicit alternative implementations, the approximate cost of the (logically) same operation in different implementations can be obtained from pragmas, analysis, or profiling. Once a correct program is produced, compiler and programmer know they can swap alternative implementations, and can focus on efficiency. Additionally, as a testing aid, a compiler could automatically generate test code to perform the same operations on two alternative implementations simultaneously and compare the results.

Alternative implementations mesh well with inheritance, if, as in the case of BULK (and C++), the member functions of a class have no special access to the private members of superclasses. Because alternative implementations are identical in their public members, member functions can operate on any alternative implementation of a superclass.

5 Static Analysis and Optimization

As an example of static analysis and optimization, we sketch BULK's approach to selecting disk-based data structures; ([RS91] explains the approach in detail). The method operates in two phases on a *workload*, i.e. on a collection of transactions and their execution frequencies.

In phase one, for each transaction, a set of rules determines characteristics of a physical design that might be beneficial (compared to some naïve data structure) to the transaction; from among the candidate designs for each transaction, one is selected that yields a low cost estimate for that transaction. These "characteristics of a physical design that might be beneficial" are termed *useful features*. As an example, for Transaction 1 in Figure 5, useful features would include

f_1 : indexing `known-bond` and `bond-group` by their domain elements,

f_2 : maintaining each instance of `bond.history-data` in descending order of its domain and indexing it by domain elements, and

f_3 : clustering `history-data` with the other data members of each `bond` instance.

In another example, consider Transaction 2 (Figure 6). This transaction prints, for each known bond (in order of `maturity`) the `maturity`, `coupon`, and the most recent three elements of `history()`. Useful features for Transaction 2 would include

g_1 : maintaining `known-bond` in order of `maturity`,

```

pragma
  cardinality(known-bond) = 1000;
  cardinality(element(range(known-bond)).history-data) = 3000;
end pragma;

```

Figure 7: Data Statistic Pragmas

g_2 : maintaining, for each bond, the value of the expression

$$[q \text{ in history-data order by } q(1), >](1..3), \quad (1)$$

i.e. the most recent three `history-data` elements, and

g_3 : storing the remainder of `history-data` (i.e. other than the most recent three quotes) separately from `coupon` and the other data members of `known-bond`.

The decision to maintain (1) is made not by the programmer, as in the `cache-most-recent` implementation of `bond` of Figure 3, but by the BULK optimizer.

In phase two, the method attempts to find a compromise data structure that minimizes the aggregate frequency-weighted cost of all the transactions in the workload. This entails searching among subsets of the union of all the useful features for the workload; the goal is to find a set of features on which the workload has a low frequency-weighted aggregate cost.

Suppose that there are 1000 bonds, each with an average of 3000 elements in `history-data`. Assuming a workload consisting of the two example transactions with approximately equal frequencies, the data structure selection process would arrive at a representation where

f_1 : `bond-group` and `known-bond` are indexed on their domains,

g_1 : `known-bond` is ordered by `maturity`,

g_2 : expression (1) is maintained,

g_3 : only the three most recent history points are stored with the `coupon` and `maturity` members of `bond` instances, and

f_2 : the remainder of the `history-data` member for each bond is stored separately, but indexed on and sorted in descending order by its domain elements.

Execution plans for the transactions are developed along with the candidate data-structures.

To provide information for optimization in the early stages of development (e.g. during the initial coding and testing), programmers can supply additional, optional, information to BULK using pragmas. Figure 7 shows programmer estimates of the cardinality of `known-bond` and the average cardinality of a bond's `history-data` member.

6 Application Development and Maintenance in BULK

As we envision it, application development in BULK would begin (after requirements analysis and specification) by an initial development using a combination of volatile data and test persistent name spaces with small values. Once the application programs have been debugged, the application would be compiled into fully optimized code using programmer-supplied expected statistics for persistent values and workload.⁸

⁸ O_2 , motivated by similar objectives, takes a related approach [VBD89]. During a development phase O_2 relies on late binding that does limited error checking and little optimization. Then in “execution mode” (i.e. for the production phase) O_2 “deeply compiles” classes by performing additional analysis and replacing dynamic class implementations with statically optimized ones. BULK places more emphasis on static analysis and optimization both in development and production phases, as exemplified by BULK’s static data structure selection and transaction planning. To some extent it may be that this approach is possible for BULK because, as a system with a single language, BULK has complete information about an application. Such complete information might be harder to get in O_2 ’s architecture, which employs separate programming languages and object manager.

```

pragma
  instruction-cost(treasury.price-to-yield) = 50000;
  pure(treasury.price-to-yield);
end pragma;

```

Figure 8: More Informational Pragmas

The prototype would be then tested by users. If its functionality is satisfactory, development would begin to address remaining efficiency concerns. The application would be run for some time, while BULK gathered profile and data statistics, and programmers identified hot spots. Once hot spots are identified, programmers would attempt to refine them into more efficient code.

6.1 Hot-Spot Refinement

Although sometimes it may be necessary to recode part of the application at a logical level—that is, to refine a hot spot without support from BULK—we hope that in most cases programmers can use BULK’s support for hot-spot refinement in one of three ways:

1. By using *informational pragmas*: pragmas that provide more information about data or about the algebraic or cost properties of functions or operations.
2. By using *directive pragmas*: pragmas that direct optimization, for example by directing BULK to use a particular data structure to represent a value, or to use a particular sequence of operations to evaluate an expression.
3. By creating an alternative class implementation.

Informational Pragmas Figure 8 shows a plausible use of informational pragmas for hot-spot refinement. (Figure 7, another example of informational pragmas, is probably characteristic of their use in the initial coding and testing phases of development rather than in hot-spot refinement.) Here `treasury.price-to-yield` refers to the `price-to-yield` function of the class `treasury`, which happens not to be written in BULK. The `instruction-cost` pragma indicates how many instructions a function execution is expected to take. The `pure` pragma indicates that its argument, a function, (i) has no side effects, and (ii) returns a value that depends only on the arguments the function is called with.

Another example of an informational pragma is the following. Suppose a programmer knows (by manual analysis and proof) that in the expression `A union B` (where `A` and `B` are sets), that `A` and `B` are disjoint. The programmer can write

```
pragma A intersect B = {}; end pragma;
```

to allow BULK to use a cheaper implementation of `union`. (Of course, for testing and debugging, BULK should be able to insert run-time code to check the truth of the pragma—this is also the case for the `pure` pragma and any other that affects correctness. Programmers would probably also like to occasionally check the accuracy of pragmas that give data and cost estimates.)

Directive Pragmas The following directive pragma forces bonds in `known-bond` to use the `cache-most-recent` implementation:

```

pragma
  use-implementation(element(range(known-bond)), cache-most-recent);
end pragma;

```

(Recall that bonds are elements of the range of `known-bond`.)

To direct BULK to simply *consider* the `cache-most-recent` implementation of `bond`, but still allow BULK to make the implementation choice, one can write

```

pragma
  try(use-implementation(element(range(known-bond)), cache-most-recent));
end pragma;

```

As a final example of a directive pragma, consider

```

pragma memoize(treasury.price-to-yield, 200); end pragma;

```

This directs BULK to “memoize” `treasury.price-to-yield`, that is, to cache up to 200 domain-range pairs for this function. This is an improvement if the function is relatively expensive and often called on the same values repeatedly. Memoization is only possible for pure functions. Ideally we would like BULK to warn when it is unsure whether a directive is correct, as would be the case in this example if the `pure` pragma in Figure 8 were omitted.

Alternative Implementation We have already seen, in Figure 3, an alternative implementation of a class.

As another example, suppose that the `bond` member function `average-price` (see Figure 1) is almost-always called with an argument that is five days before the date of the most recent quote. Then an alternative implementation of `bond` could store the value of `average-price` for that argument. It is unlikely that an automatic optimizer would ever have enough information to discover this optimization.

Once an alternative class implementation is defined, the compiler will consider it as a possible representation for a value, especially if the programmer can provide heuristics for the compiler to determine when the alternative class implementation might be preferable to the canonical class implementation. The programmer, can, of course, simply direct the BULK to use a particular alternative class implementation.

6.2 Evolution of Type and Representation

Two requirements often imposed on database applications seem incompatible with BULK’s reliance on extensive (and expensive) static analysis and optimization. These are:

- The application must run continuously. This requirement is called “high availability”, in the sense that the data and programs must always be available.
- The data and the operations on the data evolve over time.

One cannot shut down the application for hours or days to change the type or representation of the data and re-analyze all the programs.

Dynamically-typed systems seem to have an advantage here, but in practice, it is difficult to correct all the programs that become erroneous because of some change. In a characteristic scenario the newly-erroneous program is discovered three days after the change is made, in the course of some already-late batch run at 3:00 A.M., after the programmer who best understands the change has left for vacation. Therefore BULK must provide continuous availability and evolutionary change without sacrificing static analysis.

Our approach is to accommodate type change in persistent values by temporarily using a union type. For example, to change the logical type of some persistent value, X , from `set(t_1)` to `set(t_2)` one would need first to ensure that all code in the persistent name space that uses X can accommodate a set that contains values of both type t_1 and t_2 . Once that is accomplished the type of X can be changed to a set of the union type of t_1 and t_2 , and the values of type t_1 in X can be replaced by values of type t_2 . Finally the type of X can be changed to `set(t_2)`, and code that uses X can be rewritten based on the new type of X .

A related approach works for changes in representation. For example, suppose it happens, because of workload changes, that a file representing a set, X , and sorted by one attribute might be better sorted by another attribute. We can temporarily represent X in two files (containing disjoint subsets of X) after re-compiling all client programs of X to utilize the bifurcated representation. Once X is completely cut over to the new representation, client programs are compiled again to utilize only the new representation.

7 Summary

We have proposed a very-high-level persistent language to address present-day difficulties in database application development. A central idea is support for the separation of correctness from efficiency concerns. To help developers achieve correctness quickly, BULK offers a convenient notation for coding a prototype at a high level. This notation is characterized by

- uniform treatment of sets and sequences with high-level operators,
- data encapsulation with inheritance,
- independence of type and persistence, and
- static type checking that spans the application and its persistent data.

To address efficiency requirements BULK offers

- built-in alternative representations for built-in bulk types,
- optimization machinery to automatically select a low-cost implementation,
- programmer-defined alternative implementations of bulk types, which can be considered by the compiler during automatic optimization, and
- a set of pragmas to provide additional information to the compiler or directly guide its optimizations.

Though BULK relies on static analysis, we believe it can support data-type and representation evolution and still provide high availability.

Our goal is to realize an implementation that will show that BULK's approach to database application development is feasible and competitive.

Acknowledgments Thanks to the referees and to Glenn H. Holloway, Walter G. Morris, and Judy Townley, for reading earlier drafts of this paper, and whose suggestions led to many improvements.

References

- [AB87] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.
- [ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM TODS*, 10(2):230–260, June 1985.
- [Atk83] M. P. Atkinson *et al.* An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.
- [Ban88] François Bancilhon. Object-oriented database systems. In *Proc. Seventh ACM SIGACT-SIGMOD-SIGART Int'l Symp. on Principles of Database Systems*, March 1988.
- [CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. Technical Report ACA-ST-063-89, MCC, 1989.
- [CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *SIGMOD*, pages 316–325, June 1984.
- [Com83] Computer Corporation of America. *ADAPLEX: Rationale and Reference Manual*, May 1983. Technical Report CCA-83-08.

- [DGLS79] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, and Jacob T. Schwartz. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM TOPLAS*, 1(1):27–49, July 1979.
- [DKV87] S. Danforth, S. Khoshafian, and P. Valduriez. FAD, a database programming language, revision 2. Technical Report DB-151-85, Rev. 2, MCC, September 1987.
- [EEK⁺85] H. Eckhardt, H. Edelman, J. Koch, M. Mall, and J. W. Schmidt. Draft report on the database programming language DBPL. Technical Report DBPL-Memo 091-85, Univ. of Frankfurt, Fachbereich Informatik, 1985.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [KLW87] Alfons Kemper, Peter C. Lockemann, and Mechtild Wallrath. An object-oriented database system for engineering applications. In *SIGMOD*, pages 299–310, May 1987.
- [KMP⁺83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J. W. Schmidt, and C. A. Zehnder. Modula/R report, Lilith version. Technical report, Institute für Informatik, ETH, Zurich, 1983.
- [KS84] P. Kruchten and E. Schonberg. The Ada/Ed system: A large-scale experiment in software prototyping using SETL. *Technology and Science of Informatics*, 3(3), 1984.
- [Lin88] V. Linnemann *et al.* Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the 14th International Conference on Very Large Databases*, pages 294–304, 1988.
- [MAD87] R. Morrison, M. P. Atkinson, and A. Dearle. Flexible incremental bindings in a persistent object store. Technical Report Persistent Programming Research Report 38, Univ. of Glasgow, Department of Computing Science, June 1987.
- [Mor91] Walter G. Morris. CCG: A prototype coagulating code generator. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.
- [MS86] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In *Proc. Int'l Workshop on Object-Oriented Database Systems*, pages 171–182. IEEE Computer Society Press, September 1986.
- [MS87] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
- [OBBT89] Atsushi Otori, Peter Buneman, and Val Breazu-Tannen. Database programming in Machiavelli — a polymorphic language with static type inference. In *SIGMOD*, pages 46–57, 1989.
- [OSD86] Jack A. Orenstein, Sunil K. Sarin, and Umeshwar Dayal. Managing persistent objects in Ada: Final technical report. Technical Report CCA-86-03, Computer Corporation of America, May 1986.
- [PH87] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code — A case study. *J. of Symbolic Computation*, 7(4):207–232, 1987.
- [RC87] J. E. Richardson and M. J. Carey. Programming constructs for database system implementation in EXODUS. In *SIGMOD*, 1987.
- [RC88] J. E. Richardson and M. J. Carey. Persistence in the E language: Issues and implementation. Technical Report #791, Univ. of Wisconsin, September 1988.
- [Rea85] Reasoning Systems, Inc., Palo Alto CA. *REFINE User's Guide*, 1985.

- [RS89] Steve Rozen and Dennis Shasha. Using a relational system on Wall Street: The good, the bad, the ugly, and the ideal. *CACM*, 32(8):988–994, August 1989.
- [RS91] Steve Rozen and Dennis Shasha. A framework for automating physical database design. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 401–411, September 1991.
- [Sch77] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM TODS*, 2(3):247–261, September 1977.
- [SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets*. Springer-Verlag, 1986.
- [SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM TOPLAS*, 3(2):126–143, April 1981.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Str88] Bjarne Stroustrup. What is object-oriented programming. *IEEE Software*, pages 10–20, May 1988.
- [Tur82] David A. Turner. Recursion equations as a programming language. In Darlington *et al.*, editor, *Functional Programming and Its Applications*. Cambridge University Press, 1982.
- [VBD89] Fernando Velez, Guy Bernard, and Vineeta Darnis. The O₂ object manager: An overview. Technical Report 27-89, GIP Altaïr, February 1989.