Towards an Object-Oriented Query Algebra

Bennet Vance

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-008

January, 1992

Towards an Object-Oriented Query Algebra

Bennet Vance
Oregon Graduate Institute

January 15, 1992

Abstract

As part of the Revelation project in object-oriented query processing, we are seeking an algebra over complex data structures that is both simple and expressive.

The expected uses of object-oriented databases include engineering and scientific data management as well as management of complex business applications. Bulk processing on these databases will include matrix and vector operations in addition to the traditional set and relational operations. There will also be a need for operations that move information between ordered structures such as arrays and unordered structures such as sets and multisets.

We outline an approach to constructing a query algebra capable of expressing these operations. We suggest a data model that includes subtyping and identity, and a core set of very general bulk operators. Our approach is compared with others that have appeared in the literature.

1 Introduction

Revelation [DGK+91] is a project on query processing in object-oriented databases. The facilities to be provided by the database are intended to cover the needs of a variety of applications, but special attention is being given to scientific applications, for which earlier databases provide relatively meager support.

This paper proposes a query algebra for Revelation. Query algebras are of interest for at least two reasons. First, they provide an abstract language in which to reason about the meanings of queries and the expressiveness of user query languages. Second, query algebras have great practical utility in query optimization: a user query, once translated into an algebra expression, can in many cases be transformed through algebraic identities into an equivalent expression that can be evaluated much more rapidly. Algebraic query optimization is an established technique in the implementation of relational databases [Mai83].

However, in object-oriented databases, the algebraic formulation of queries is complicated by *encapsulation* of object behaviors, and by the presence of *multiple bulk data types.* In addition, object-oriented queries range over complex data structures, and mix bulk operations with *general-purpose computation.* The remainder of the introduction elaborates on these complications.

1.1 Breaking Encapsulation

In Smalltalk-80 [Gol84] and other purely object-oriented languages, there is no such notation as $f(x)$ for the application of a function f to an argument x. The closest analogue is the notation $x\ f$, in which f is referred to as a *message*, and x as the *receiver* of the message. The idea behind these differences in notation and terminology is that the responsibility for computing $x\ f$ lies with the object x, rather than with some free-standing implementation for f. Different objects may prescribe different *methods* of computing f, i.e., different definitions of f. We say that the behavior of the object x is *encapsulated* because the meaning of $x\ f$ is determined internally by the implementation of x, and cannot be known by any agent external to x.

Encapsulation is widely viewed as beneficial from a software engineering standpoint, but it is an obstacle to optimization. Consider the following example of optimization in the compilation of a conventional programming language: The expression $5 * f(x) + 5 * g(x)$ can be optimized to $5 * (f(x) + g(x))$ without any knowledge of the definitions of f and g. But the expression can be further optimized if it happens that $f(x) = 3 * x$ and $g(x) = 2 * x$, for in that case, $5 * (f(x) + g(x))$ can be rewritten as $5 * (3 * x + 2 * x)$, and simplified to $25 * x$.

The analogous expression in an object-oriented language is less easily optimized. As before, $5 * (x\ f) + 5 * (x\ g)$ can be rewritten as $5 * (x\ f + x\ g)$, but it is not obvious how to expand $x\ f$ or $x\ g$ into more informative expressions. To do so would appear to require knowing how x was implemented, but since x is a variable, there is no telling what object x might represent. Two successive evaluations of $x\ f$ with different bindings for x might expand to two entirely different algebraic expressions.

However, it is possible to expand $x\ f$ into an algebraic expression that is valid for all bindings x might take on in a given context. How this is done is explained in the body of the paper. Here we note only that in Revelation we take advantage of knowledge of the database schema, and even the database state, to expand messages into algebraic expressions. Hence the correctness of an algebraic expansion of an expression may depend on the current state of the database; the expression may have to be reexpanded and reoptimized as the database changes. Note, though, that there is nothing new about data-dependent optimization in query evaluation. Query optimization traditionally relies on database statistics to estimate the costs of alternative query evaluation strategies.

We refer to expanding $x\ f$ into a more informative expression as *revealing the definition of* f. It is from this concept of revealing that the Revelation project takes its name.

1.2 Bulk Data Types

The term *bulk data type*, or simply *bulk type*, refers to such structures as arrays, sets, and sequences, which can contain arbitrarily many elements. Trees are also considered bulk types because they may contain arbitrarily many nodes. However, tuples are not considered bulk types; it is true that tuples may contain arbitrarily many components, but a *given* tuple type has a fixed number of components. Intuitively, bulk types are those types that make it possible for databases to contain huge amounts of data.[1] In conventional programming languages, it is almost always necessary to use iterative or recursive constructs to process data from bulk types.

Relational query languages can express bulk processing operations without explicit iteration, as can the relational algebra. This expressive capability is convenient for users and is also conducive to high-level optimizations. However, one of the limitations of the relational model is that it provides only one bulk type: the set. That is, a relation is by definition a *set* of tuples. Being unordered, sets are not convenient for modeling such entities as *lists* (or *sequences*) of events. Note also that sets may not contain duplicates; yet many relational database implementations fail to enforce this property, because it is expensive to do so. In those implementations, relations are actually *multisets* (or *bags*) of tuples, not sets of tuples. Thus, the relational model's exclusive reliance on sets has drawbacks both for application modeling and for implementation efficiency.

The bulk types we will support may be divided into three groups: arrays (of one or more dimensions), trees, and what might be called the *flat* bulk types: sets, multisets, and lists. We want our algebra to be able to express a wide variety of useful operations over these bulk types, but we also want the algebra to be as simple and regular as possible. Complexity would be a hindrance not only to abstract reasoning about the algebra, but also to reliable implementation.

Thus, rather than define a new *ad hoc* set of operators for each bulk type, we exploit the similarities between bulk types to define operators that have analogues from one type to the next. Many algebraic identities on these operators are shared among the different bulk types.

There are necessarily some operators and laws that distinguish among the bulk types. Multidimensional arrays stand out as having processing needs unlike those of the other bulk types. We imagine that a set

[1] Devising a precise definition of *bulk type* is annoyingly tricky. An appealing characterization due to Trinder [Tri91] is that the size of a value from a bulk type is independent of the size of the type description. However, by a malicious reading of this characterization, one can conclude that Int is a bulk type—there is no intrinsic reason why all integers should occupy the same amount of storage, though in most systems they do. Furthermore, a tuple whose components were sets would be a bulk type by this characterization. Because of such difficulties, and because precision on this point is not critical to our topic, in this paper we forego a formal definition of *bulk type*.

of array manipulation operators based on those of APL [FI68] would support efficient array processing, and would provide excellent opportunities for high-level algebraic optimization. However, the present work touches only lightly on arrays.

1.3 General-Purpose Computation

The relational algebra cannot express arithmetic operations, conditional constructs, or explicit iteration or recursion—but does express bulk operations concisely. By contrast, conventional programming languages support general-purpose computing, but usually can express bulk operations only through iterative or recursive constructs. We seek to provide much of the best of both worlds.

In the implementation of our algebra, we will provide access to general-purpose computation by departing from convention in two ways. First, the algebra will include operators on scalar types as well as on bulk types; it will also support "small" constructed types—types that are neither scalar nor bulk types, such as tuples and discriminated unions. Second, when expanding query language constructs into algebraic expressions, we will permit subexpressions to remain in unexpanded form if they cannot be expressed algebraically. Such unexpanded subexpressions will not be subject to algebraic optimization, and we will not touch on them further.

We base our treatment of scalar types and small constructed types on the functional programming model [BW88]. A functional program without recursion is essentially an algebraic expression over a many-sorted algebra, and is amenable to optimization through algebraic transformations. Thus, functional programming blends together smoothly with the algebraic formulation of queries. Moreover, by parameterizing our bulk operations with functional arguments, we obtain a synergy between bulk operations and operations on booleans and other small types. With only a handful of bulk and non-bulk operators, we obtain a very expressive algebra.

2 An Extensible Data Model

Our view of data is at heart a functional one. All data will be typed, and we will concern ourselves almost exclusively with immutable data structures. Thus, there will be no such operations as *insert* or *remove* on lists and sets; rather, operators may be applied to lists and sets to obtain new lists and sets.

In the following, we first develop support for arbitrary data structures, without regard to object-oriented issues. Subsequently we consider the role of encapsulation, subtyping, and identity.

2.1 Built-in Data Types

As is customary, we conceive of data types as being freely built up from a collection of built-in types and type constructors. The question is which types should be built in. On the one hand, we want to provide efficient support for the data structures we consider important; on the other hand, if too many types and constructors are built in, we may lose opportunities for optimization, in addition to sacrificing conceptual simplicity, and complicating implementation.

There is not much doubt that we want our built-in types to include Bool, Int, Real, and Char. Also indispensable, in our view, are the small constructed types: product types (i.e., labeled records, as well as pairs, triples, and other ordered tuples), discriminated unions (including recursive discriminated unions, which may be used to build trees), and function types.

Some of the bulk types, such as List, will also be built in. It would certainly be possible to define List in the traditional way, as a recursive discriminated union with Nil and Cons as constructors. However, we can obtain far superior performance on list operations if we represent a list as a (mostly) contiguous sequence of storage locations, much like an array. For this reason, we will make List a built-in type constructor with its own special operators.

Multisets are similar to lists, and will be handled similarly. As we have just suggested, one-dimensional arrays are also similar to lists. The only difference, in our treatment, is that array elements will have an index associated with them, and may be accessed randomly. Multidimensional arrays could be defined in terms of one-dimensional arrays, but it is preferable to make them built-ins in their own right: Suppose we were to support only one-dimensional arrays directly, and to require higher-dimensional array operations to be implemented using one-dimensional representations. Then straightforward operations on two-dimensional arrays, such as transposition, while easy enough to implement using loops or recursion, would be extremely awkward to express algebraically. In many instances this awkwardness of expression would translate to inefficiency in evaluation.

On the other hand, there is also a danger in including built-in types that are too high-level. Such types as Matrix (in the linear algebra sense) and Relation will not be supported directly, for reasons that should become clear presently.

2.2 Decomposition of High-Level Operators

The appeal of including high-level operators such as matrix multiplication, or the relational operators, is that they lend themselves to abstract optimizations. Thus, one might algebraically optimize the matrix expression $A*B+A*C$ to $A*(B+C)$, or the relational expression $\sigma_p(R \bowtie S)$ to $(\sigma_p R) \bowtie S$, on the basis of computational complexity alone. In a later optimization phase, one would work out a plan for multiplying, joining, etc., in such a way as to minimize storage access overhead.

However, this idea of a clean separation between the different phases of optimization does not always work out as nicely as one might wish. An example involving matrix multiplication illustrates the problem.

The matrix product $A * B * C$ can be evaluated either as $(A * B) * C$ or as $A * (B * C)$. If we know nothing about the matrices, either evaluation strategy is equally good. But suppose now that we know that the dimensions of A, B, and C, are $n \times m$, $m \times n$, and $n \times m$, respectively, and let us say that $n \gg m$. (See Figure 1(*i*).)

Whether we choose to multiply $A * B$ first or $B * C$ first greatly affects the sizes of our intermediate results (Figure 1(*ii, iii*)). If we make the wrong choice, we suffer both in computing the intermediate result, and again in computing the final product. It is more expensive to compute $(A * B) * C$ than $A * (B * C)$ by a factor of $(n/m)^2$.

This much we can determine if Matrix is an abstract type with a hidden representation. But the costs of the computation can be analyzed more exactly if the storage layouts of A, B, and C are known. Suppose A has row-major organization, and B and C have column-major organization (Figure 1(*iv*)). If both n and m are large, then access patterns that cross rows of A involve consulting secondary storage much more frequently than access patterns within rows of A; and similarly for the *columns* of B and C. Consequently, assuming a naive matrix multiplication algorithm, the storage layouts shown here are more favorable to computing $A * B$ than $B * C$ [GK90]. In a given situation, this consideration may or may not outweigh the fact that in the abstract, $B * C$ is the best first step. For the purposes of estimating evaluation costs, we would do well to decompose matrix multiplication into more primitive operations.

2.3 Quasi-built-in Types

As we have seen, there is a trade-off involved in incorporating high-level, abstract operations into the algebra. Abstractions can increase expressive power and provide high-level optimization opportunities, but by hiding the details of a computation, they can make cost estimation difficult.

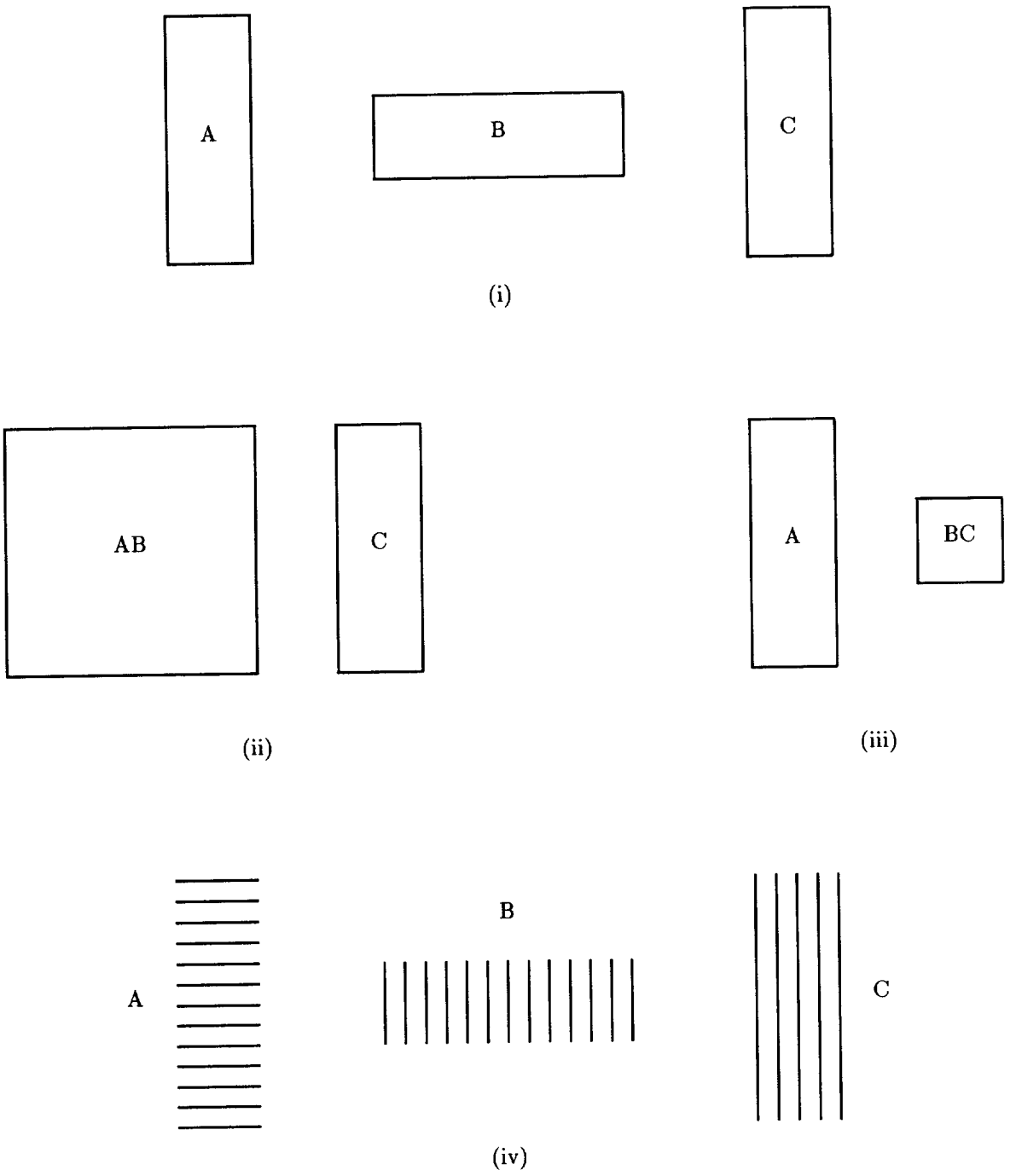

Figure 1: Array Sizes and Storage Layouts

To some extent we can avoid making a trade, if we give special status to types such as Matrix. From a user's viewpoint Matrix might appear to be built in; and, equally important, the Matrix operators could be assumed to obey algebraic laws such as $A * (B * C) = (A * B) * C$ and $(A * B)^T = B^T * A^T$. However, the operators would have no direct implementations, nor would any cost metrics be associated with them. Instead, the laws would be used at optimization time to evolve alternative formulations of a given high-level query; each alternative would then be expanded into an expression built with more primitive operators. The resultant expanded expressions would be separately optimized, and the one with the best cost estimate (after optimization) would be selected for evaluation.

A similar approach is probably appropriate for Set. Sets are a valuable abstraction for users, and they also have useful algebraic properties. However, set operations are often decomposable into more primitive operations. For example, a common implementation of set union computes the multiset union of two multisets, and then eliminates duplicates. By making this decomposition explicit in our algebra, we will open up optimization possibilities that would otherwise be unavailable.

2.4 User-Defined Types and Implementations

In addition to built-in types and quasi-built-in types, we will have user-defined types. To achieve abstraction and encapsulation, we separate the definition of a new type's *interface* (or *protocol*) from the type's *implementation*. The protocol specifies what messages are understood by a given type, and also specifies the argument and result types of those messages. The implementation consists of a representation for a type, together with a collection of message definitions (or methods). To qualify as an implementation for a given type, the implementation must *conform* to the type's protocol [DGK+91]. Subtype relationships between types are also founded on conformance of protocols.

An important aspect of our model is that a given type may be implemented in more than one way. Indeed, a collection of values of some type may simultaneously contain values with different implementations. We may even conceive of having multiple implementations for *built-in* types. Pursuing this idea, we might observe that while there is a clearly a distinction between built-in and user-defined *implementations*, there is no inherent distinction between built-in and user-defined *types*. We could regard all types—even scalar types—as *abstract data types* [Lom76, Lom80] supported by some combination of built-in and user-defined implementations.

Alternative implementations for built-ins are likely to have practical value in scientific applications, where a number of representations are used for multidimensional arrays [BP87]. For example, an n-dimensional

array might be represented as a "smaller" n-dimensional array of n-dimensional subarrays; sparse arrays can be compactly represented with structures containing only the nonzero array elements. However, to avoid unnecessary distractions, the remainder of this paper will assume a single implementation for the types we have designated as built-ins.

2.5 Encapsulation and Subtyping

Let us now turn to the problem of breaking encapsulation in algebraic expressions. The mechanism we propose for breaking encapsulation will also give us the means to cope with subtyping in the algebra. This mechanism hinges on the observation that at a given time, a given type has some fixed number of implementations.

We can reveal the expression $x\ f$, even when the value of x is not known, by using type information. Assume the variable x has type T; then we can infer that x is implemented by one of T's existing implementations. Let us call these implementations $I_1, I_2, \ldots, I_n$. Each of these implementations will provide a method (i.e., a function definition) for f (perhaps by inheritance). Let us give the name f_i to the function for f defined by implementation I_i.

Informally, we know that if x has implementation I_i, then the effect of $x\ f$ is to apply f_i to x. To manipulate $x\ f$ in the algebra, we need to formalize our intuition about the meaning of $x\ f$.

Our mechanism for describing the action of messages in the algebra is as follows: We will regard objects whose types have multiple possible implementations as belonging to discriminated unions inside the algebra. This use of discriminated unions is not to be confused with our making discriminated unions available to users as type constructors. To separate the user's view of types from the internal one, we will use the word *sort* for the internal classifications. In many instances, *types* will map directly to *sorts*. Our use of union sorts is intended to be mostly behind the scenes.

Now, instead of saying that x has type T, we may say x has sort $I_1 + I_2 + \cdots + I_n$, and we may define

$$x\ f \equiv \left(\begin{array}{rcl} \text{case } x \text{ of } \text{is}I_1(z) & \Rightarrow & f_1(z) \\ \text{is}I_2(z) & \Rightarrow & f_2(z) \\ & \vdots & \\ \text{is}I_n(z) & \Rightarrow & f_n(z) \end{array} \right)$$

As shorthand for the right-hand side of this definition, let us use the notation $f_T(x)$. In other words, we are interpreting the message application $x\ f$ as the application of a *function* to x—a function that happens to be defined by a case expression.

The advantage of rewriting $x\ f$ as the case expression $f_T(x)$ is that it then becomes susceptible to algebraic manipulation. For example, when messages are cascaded, as in $(x\ f)\ g$, optimization possibilities are created by the general law for case expressions

$$g_\upsilon \left(\begin{array}{rcl} \text{case } x \text{ of is}I_1\ z & \Rightarrow & f_1(z) \\ \text{is}I_2\ z & \Rightarrow & f_2(z) \\ \vdots & & \\ \text{is}I_n\ z & \Rightarrow & f_n(z) \end{array} \right) = \left(\begin{array}{rcl} \text{case } x \text{ of is}I_1\ z & \Rightarrow & (g_\upsilon \circ f_1)(z) \\ \text{is}I_2\ z & \Rightarrow & (g_\upsilon \circ f_2)(z) \\ \vdots & & \\ \text{is}I_n\ z & \Rightarrow & (g_\upsilon \circ f_n)(z) \end{array} \right)$$

where the function g_υ may be taken as an interpretation for the message g. This law transforms cascaded messages into a collection of function compositions of the form $g_\upsilon \circ f_i$, some of which may be optimizable when the definitions of g_υ and f_i are revealed.

Another transformation on case expressions, applicable if the discriminated unions are suitably represented, and $f_1, f_2, \ldots, f_n$ are all the same, is to collapse the entire case expression to $f_1(\mathit{strip\_tag}(x))$, where $\mathit{strip\_tag}(\text{in}I_i\ z) \equiv z$. This important optimization arises frequently as a result of inheritance.

A side benefit of making implementations explicit through the use of union sorts is that in situations where a value is known to have only one possible implementation, it need not have a discrimination tag. This data optimization distinguishes our approach from naive models, in which even atomic objects such as integers must carry the overhead of a type tag alongside the data value.

The union sort mechanism makes it easy to accommodate subtyping in the algebra, because for the purposes of the algebra the effect of subtyping is merely to increase the number of possible implementations for a type: Any implementation for the subtype of a type is also an implementation for the type itself.

2.6 Identity and Updates

Up to this point all the constructors we have mentioned yield pure-value types that do not admit updates. In this paper we will not consider update queries, and for the queries we do consider, the presence of updatable cells in the data model has no effect. Nonetheless, as updates are essential to a database, a few comments about object identity and updatable cells are in order.

Following Ohori [Oho90b], we base the semantics of object identity on the ML *ref* constructor. Each evaluation of the expression *ref* E yields a new updatable cell that is different from any other cell. Thus, *ref* may be used to create objects.

We depart from Ohori in allowing an arbitrary number of *ref* cells in the representation of a single object, rather than asserting a one-to-one correspondence between *ref* cells and objects. For us, an object is an

encapsulated, possibly complex value in which some number of *ref* cells have been embedded. A value with *zero* embedded *ref* cells is still an object, but it is an immutable object and, unlike mutable objects, it is not guaranteed distinct from all other objects.

In this model, every object is a value, and every value is an object. Whether or not an object (or value) is tagged with type information depends on whether it belongs to a union sort.

3 Algebraic Operators and Identities

3.1 Support for Small Types

For the scalar types, the choice of primitive operators and identities is not controversial. For booleans, one will have the usual boolean connectives as operators, commutativity and associativity of $\wedge$, $\vee$, and *exclusive or*, the de Morgan laws, the fact that *false* is a unit for $\vee$ and a zero for $\wedge$, and so forth. For integers, too, the operators and laws are straightforward. Reals are a little more difficult, as associativity and commutativity are not always upheld by floating point operations that nominally have these properties. In the present work, however, we will not delve into these difficulties.

We shall also assume the usual operators and identities on the small constructed types. For example, for function types, the *apply* operator applies a functional value to an argument, and obeys the β- and η-conversion rules of the λ-calculus. For *ref* cells there are creation, dereferencing, and update operators, but in this paper we exclude creation and update, and consequently there are no applicable laws. For pairs, the operators are *fst*, *snd*, and the pair constructor, and we have, among other laws, $(fst(z), snd(z)) = z$; similarly for other product types. The operators for discriminated unions are injections and case expressions; among the laws for the discriminated union $T + U$ are these:

$$\left(\begin{array}{rcl} \text{case } x \text{ of is}T\ z & \Rightarrow & \text{in}T(z) \\ \text{is}U\ z & \Rightarrow & \text{in}U(z) \end{array}\right) = x \tag{1}$$

$$g\left(\begin{array}{rcl} \text{case } x \text{ of is}T\ z & \Rightarrow & f_1(z) \\ \text{is}U\ z & \Rightarrow & f_2(z) \end{array}\right) = \left(\begin{array}{rcl} \text{case } x \text{ of is}T\ z & \Rightarrow & g(f_1(z)) \\ \text{is}U\ z & \Rightarrow & g(f_2(z)) \end{array}\right) \tag{2}$$

$$\left(\begin{array}{rcl} \text{case } x \text{ of is}T\ z & \Rightarrow & e \\ \text{is}U\ z & \Rightarrow & e \end{array}\right) = e \tag{3}$$

As special cases of these laws, viewing Bool as the union $Unit + Unit$, we have

$$\begin{aligned} (\text{if } b \text{ then } \mathit{true} \text{ else } \mathit{false}) &= b \\ g(\text{if } b \text{ then } e_1 \text{ else } e_2) &= (\text{if } b \text{ then } g(e_1) \text{ else } g(e_2)) \end{aligned}$$

$$(\text{if } b \text{ then } e \text{ else } e) \quad = \quad e$$

There are also laws relating *if-then-else* to boolean operators;[2] for example,

$$\text{if } b_1 \wedge b_2 \text{ then } e_1 \text{ else } e_2 = \text{if } b_1 \text{ then } (\text{if } b_2 \text{ then } e_1 \text{ else } e_2) \text{ else } e_2 \tag{4}$$

The importance of these laws will be seen in manipulations on bulk operations in the following.

3.2 A Reductionist Approach to Bulk Operations

An algebra over complex types should be at least as expressive as the relational algebra, and preferably much more so. One way to achieve this expressiveness would be to endow the algebra with a large number of operators as primitives, including the relational operators among them. We have already hinted at our objections to this approach:

- It is aesthetically displeasing, especially if fewer primitives would suffice.
- It raises the question of how easily the algebra can be extended to include new type constructors.
- It makes difficult the discovery and verification of algebraic identities.
- It entails a large implementation effort, and a large maintenance burden in the face of changes to the algebra.

To escape these problems, we will support bulk types with as few operators as possible. In the following, we first illustrate how one bulk reduction operator parameterized by functional arguments can be used to express many familiar bulk operations. We go on to show that this operator, which we call *fold*, obeys powerful algebraic laws that are useful for query optimization. We then observe that *fold* lends itself to efficient implementation; however, we also observe that an implementation cannot be built around *fold* alone.

Initially we introduce our treatment by concentrating on lists. Subsequently, we note the extent to which analogous treatments apply to sets, multisets, one-dimensional arrays, and trees. Finally, we observe that conversions between bulk types can be obtained effortlessly as a side benefit of the algebra's regular structure.

We shall assume that lists, multisets, and one-dimensional arrays come equipped with constructors to create empty and singleton collections, and with the fundamental primitive operator *append*, denoted $+\!\!+$. For lists, $+\!\!+_{List}$ is concatenation; $+\!\!+_{Multiset}$ is multiset union; $+\!\!+_{Arr}$ is array concatenation. In the next subsection, we abbreviate $+\!\!+_{List}$ as $+\!\!+$.

[2]Note that if booleans are viewed as unions, and the boolean connectives are defined in terms of *if-then-else* and the constant injections *true* and *false*, then all the identities of boolean algebra, as well as the laws shown here, are direct consequences of general laws for discriminated unions.

3.3 The *fold* Operator

Assuming ++ as a given, the most powerful and useful bulk primitive we know of is a reduction operator we shall call *fold*, which we introduce by way of example:

$$\begin{aligned} [2,3,4]\, fold(0, square, +) &= square(2) + square(3) + square(4) = 4 + 9 + 16 = 29 \\ [2,3,4]\, fold(1, id, *) &= 2 * 3 * 4 = 24 \\ [\,]\, fold(1, id, *) &= 1 \end{aligned}$$

Schematically, if $f : \alpha \rightarrow \beta$ is a function on the elements of an α List, $\oplus : \beta \times \beta \rightarrow \beta$ is any associative dyadic operator, and $u : \beta$ is a left and right unit (identity) for $\oplus$, then we have

$$\begin{aligned} [x_1, x_2, \ldots, x_n]\, fold(u, f, \oplus) &= f(x_1) \oplus f(x_2) \oplus \cdots \oplus f(x_n) \\ [\,]\, fold(u, f, \oplus) &= u \end{aligned}$$

At first glance it is surprising that from *fold* one can derive all the traditional relational operators and more, but in fact one can. In our examples above, the list elements are numbers, and $\oplus$ is bound to arithmetic operators. However, the following uses of *fold* are also legitimate (here u,v,w,x,y, and z are variables all of the same type; "one", "two", and "five" are text strings that obey a lexicographic ordering):

$$\begin{aligned} [[u,v,w],[x],[y,z]]\, fold([\,], id, ++) &= [u,v,w]{+}{+}[x]{+}{+}[y,z] = [u,v,w,x,y,z] \\ [\text{"one"}, \text{"five"}, \text{"two"}]\, fold([\,], f, ++) & \\ \text{where } f(s) = \text{if } s \geq \text{"one"} \text{ then } [s] \text{ else } [\,] &= [\text{"one"}]{+}{+}[\,]{+}{+}[\text{"two"}] = [\text{"one"}, \text{"two"}] \end{aligned}$$

These examples generalize to the following operators derived from *fold*:

$$\begin{aligned} xs\ flattenmap(f) &= xs\ fold([\,], f, ++) \\ xs\ flatten &= xs\ flattenmap(id) \\ xs\ filtermap(p, f) &= xs\ flattenmap(\lambda x.\text{if } p(x) \text{ then } [f(x)] \text{ else } [\,]) \\ xs\ select(p) &= xs\ filtermap(p, id) \\ xs\ map(f) &= xs\ filtermap((\lambda x.true), f) \\ xs\ project(a_1, \ldots, a_n) &= xs\ map(\lambda\{a_1 = x_1, \ldots, a_n = x_n, \ldots\}.\{a_1 = x_1, \ldots, a_n = x_n\}) \end{aligned}$$

The derived operators implement a variety of useful and familiar functions. Thus, *flatten* transforms a list of lists into a simple list:

$$[[u,v,w],[x],[y,z]]\ flatten = [u,v,w,x,y,z]$$

The *select* shown is precisely the relational σ applied to lists, and *project* is the relational π applied to lists of records; *map*, also known as *mapcar*, *apply_to_all*, and *collect*, appears in functional languages, and in other object-oriented query algebras; *flattenmap* and *filtermap* are variants on *map*. For example,

$$\begin{aligned} [2,3,4]\ map(square) &= [4,9,16] \\ [2,3,4]\ select(is\_even) &= [2,4] \end{aligned}$$

where *is_even* stands for $\lambda n.n \bmod 2 = 0$.

Familiar identities on these operators, such as

$$(xs\ select(p))\ select(q) = xs\ select(\lambda y.p(y) \wedge q(y)) \tag{5}$$

$$(xs\ map(f))\ map(g) = xs\ map(g \circ f) \tag{6}$$

can be proved by simple inductive arguments from the following identities on *fold*, which may be regarded as definitional:

$$[]\ fold(u, f, \oplus) = u \tag{i}$$

$$[x]\ fold(u, f, \oplus) = f(x) \tag{ii}$$

$$(xs +\!\!+ ys)\ fold(u, f, \oplus) = (xs\ fold(u, f, \oplus)) \oplus (ys\ fold(u, f, \oplus)) \tag{iii}$$

3.4 Algebraic Manipulation of *fold* Expressions

If we allow ourselves to use inductive reasoning, then Equations *i–iii*, in conjunction with the laws for the small types, give us the means to establish numerous identities along the lines of Equations 5 and 6. However, inductive reasoning is more difficult to perform mechanically than the equational reasoning used by existing query optimizers. These optimizers transform expressions by direct application of algebraic identities, substituting equals for equals. For use by such an optimizer, Equations *i–iii* are inadequate. We therefore supplement them with the following laws:

Law 1 *For any list xs,*

$$xs\ fold([\,], (\lambda x.[x]), +\!\!+) = xs$$

Law 2 *Let $h : \beta \to \gamma$ be a monoid homomorphism; thus, there exist operators $\oplus : \beta \times \beta \to \beta$ with identity u, and $\otimes : \gamma \times \gamma \to \gamma$ with identity $h(u)$, such that for all $x_1, x_2 : \beta$,*

$$h(x_1 \oplus x_2) = h(x_1) \otimes h(x_2)$$

Then for any $xs : \alpha List$ *and* $f : \alpha \rightarrow \beta$,

$$h(xs\ fold(u, f, \oplus)) = xs\ fold(h(u), h \circ f, \otimes)$$

Example 1 *Observe that* $\log(x_1 * x_2) = \log x_1 + \log x_2$ *and* $\log 1.0 = 0.0$ *(*1.0 *and* 0.0 *being the identities of real multiplication and addition, respectively). Thus,* log *is a homomorphism as required of* h *in Law 2. Therefore, if nums is a list of real numbers, then by Law 2 we have*

$$\log(nums\ fold(1.0, id, *)) = nums\ fold(0.0, \log, +)$$

That is, the logarithm of the product over a list is equal to the sum of the logarithms of the list elements.

Laws 1 and 2 are consequences of Equations *i–iii*. The proofs rely on induction, but once the laws are established, further identities can be derived from them equationally. Of particular interest is the following corollary of Law 2 based on the observation that Equation *iii* expresses a homomorphic property for *fold*.

Law 2a *Define*

$$h_{v,g,\otimes}(xs) \equiv xs\ fold(v, g, \otimes) \tag{7}$$

Then by Equation iii,

$$h_{v,g,\otimes}(xs{+\!\!+}ys) = h_{v,g,\otimes}(xs) \otimes h_{v,g,\otimes}(ys) \tag{8}$$

It follows that

$$\begin{aligned}
(xs\ fold([\,], f, {+\!\!+}))\ fold(v, g, \otimes) &= h_{v,g,\otimes}(xs\ fold([\,], f, {+\!\!+})) && \{\text{by (7)}\} \\
&= xs\ fold(h_{v,g,\otimes}([\,]), h_{v,g,\otimes} \circ f, \otimes) && \{\text{by Law 2, using (8)}\} \\
&= xs\ fold(v, h_{v,g,\otimes} \circ f, \otimes) && \{\text{by (7), } (i)\}
\end{aligned}$$

Law 2*a* allows us to obtain identities such as Equation 5 without induction, as we shall now demonstrate. By definition of *select*, we may write

$$\begin{aligned}
select(p) &= fold([\,], f, {+\!\!+}) \\
select(q) &= fold([\,], g, {+\!\!+}) \\
select(\lambda y.p(y) \wedge q(y)) &= fold([\,], l, {+\!\!+})
\end{aligned}
\qquad \text{with} \quad
\begin{cases}
f &= \lambda x.\text{if } p(x) \text{ then } [x] \text{ else } [\,] \\
g &= \lambda x.\text{if } q(x) \text{ then } [x] \text{ else } [\,] \\
l &= \lambda x.\text{if } pq(x) \text{ then } [x] \text{ else } [\,] \\
pq &= \lambda y.p(y) \wedge q(y)
\end{cases}$$

and hence we may rewrite the left-hand side of Equation 5 as follows:

$$\begin{aligned}
(xs\ select(p))\ select(q) &= (xs\ fold([\,], f, {+\!\!+}))\ fold(v, g, \otimes) \qquad \text{with} \begin{cases} v = [\,] \\ \otimes = {+\!\!+} \end{cases} \\
&= xs\ fold(v, h_{v,g,\otimes} \circ f, \otimes) && \{\text{by Law 2}a\} \\
&= xs\ fold([\,], h_{[\,],g,{+\!\!+}} \circ f, {+\!\!+})
\end{aligned}$$

We want to show (by deriving $h_{[],g,+\!+} \circ f = l$) that this last expression can be transformed to

$$xs\ fold([], l, +\!+), \qquad \text{i.e.,} \qquad xs\ select(\lambda y.p(y) \wedge q(y)),$$

which is the right-hand side of Equation 5. Thus, it remains only to derive $h_{[],g,+\!+} \circ f = l$:

$$\begin{aligned}
h_{[],g,+\!+} \circ f &= \lambda x.h_{[],g,+\!+}(f(x)) && \{\text{def. of } \circ\} \\
&= \lambda x.h_{[],g,+\!+}(\text{if } p(x) \text{ then } [x] \text{ else } [\,]) && \{\text{def. of } f;\ \beta\text{-rule}\} \\
&= \lambda x.\text{if } p(x) \text{ then } h_{[],g,+\!+}([x]) \text{ else } h_{[],g,+\!+}([\,]) && \{(2)\} \\
&= \lambda x.\text{if } p(x) \text{ then } [x]\ fold([], g, +\!+) \text{ else } [\,]\ fold([], g, +\!+) && \{(7) \text{ twice}\} \\
&= \lambda x.\text{if } p(x) \text{ then } g(x) \text{ else } [\,] && \{(ii), (i)\} \\
&= \lambda x.\text{if } p(x) \text{ then (if } q(x) \text{ then } [x] \text{ else } [\,]) \text{ else } [\,] && \{\text{def. of } g;\ \beta\text{-rule}\} \\
&= \lambda x.\text{if } p(x) \wedge q(x) \text{ then } [x] \text{ else } [\,] && \{(4)\} \\
&= \lambda x.\text{if } pq(x) \text{ then } [x] \text{ else } [\,] && \{\text{def. of } pq;\ \beta\text{-rule}\} \\
&= l && \{\text{def. of } l\}
\end{aligned}$$

This concludes the derivation of Equation 5.

This derivation is straightforward, but long (the derivation of Equation 6 would be a good deal shorter). Preferably an optimizer should not have to have to go through such a large number of steps to carry out a simple transformation. On the other hand, an optimizer able to carry out derivations on the fly might discover optimizations that would be inaccessible to optimizers working from a fixed list of identities for *select*, *map*, and so forth. A compromise between efficiency and flexibility might be achieved by developing a list of frequently applicable identities, and using these to short-cut the derivations whenever possible.

3.5 Evaluation of *fold* Expressions

As we have seen, operators such as *select* and *map* can be expressed in terms of *fold*. One advantage of reducing multiple bulk operators to a single one is that the effort required to build an evaluator for the algebra is thereby reduced as well. However, one might worry that the efficiency of evaluation would suffer in the absence of special-case code for the derived operators.

For the present, we shall assume we are dealing with unindexed collections. Under this assumption, implementing the derived operators in terms of *fold* can be very efficient. (Some low-level optimizations in query execution are necessary to make this true, but the ability to achieve these optimizations is desirable in any case.) In fact, it would be detrimental to efficiency to break *fold* apart and separately implement the different operators it can express. Consider the case where *select* and *project* operations are cascaded,

as they often are. Naively evaluating these operators in sequence, first *select* and then *project*, would entail generating a potentially large intermediate result, and then taking a second pass over that intermediate result to generate the final result. However, a *select* followed by a *project* can always be simplified to a single *fold* by application of Law 2a; the *fold* can then be evaluated with a single pass over the data. Any efficient relational implementation is capable of rolling evaluation of σ and π together into a single pass, and in this sense implicitly performs *fold* all the time, even though *fold* is not in the relational algebra.

Sharing common bulk processing code among the different operators offers the usual advantages of modular design. In particular, gains obtained by tuning the implementation of *fold* immediately accrue to all the operators built on top of it. One way to make *fold* fast is to use parallel evaluation. Since $\oplus$ is assumed associative, the reduction

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

may be written as

$$(x_1 \oplus \cdots \oplus x_{n_1}) \oplus (x_{n_1+1} \oplus \cdots \oplus x_{n_2}) \oplus \cdots \oplus (x_{n_{k-1}+1} \oplus \cdots \oplus x_n)$$

Thus the reduction can be organized as k parallel reductions carried out on k processors and then combined. Note that parallel evaluation opportunities are not generally afforded by list reductions in functional programming languages. For example, the *foldl* and *foldr* functions of Miranda[3] [BW88], while every bit as expressive as our *fold*, must be evaluated serially. We defined *fold* as we did partly for the sake of flexible evaluation order.

3.6 The Complexities of Join

We have not run out of things that can be done with *fold*, but it is not a panacea. Let us turn to an operation for which *fold* is not well suited: relational join. Certainly relational join can be expressed concisely with *fold*, but this yields a brute-force computation. It is probably better to introduce a new primitive *crossfold* with the following semantics:

$$xs\ \mathit{crossfold}(f, g, h)\ ys = xs\ \mathit{flattenmap}(\lambda x.ys\ \mathit{filtermap}((\lambda y.f(x) = g(y)), h(x, y)))$$

The right-hand side computes the cross product of xs and ys, but each pair (x, y) is retained only if $f(x) = g(y)$; the final result is then obtained by applying h to each (x, y) pair that is retained. This computation is a

[3]Miranda is a trademark of Research Software Limited.

slightly generalized form of join that lends itself to implementation by the same kinds of efficient algorithms as relational join.

A difficulty that arises in complex object algebras is that whereas relational join and cross product are associative, their complex-object analogues usually are not. In complex-object algebras, the elements of a bulk type are not necessarily tuples, and there is no general way to combine two elements of unknown structure except to pair them. Since $((x,y),z) \neq (x,(y,z))$, the tactic of pairing yields nonassociative operations.

One way to regain associativity is to invent a complex rule for combining two complex elements. Consider this rule: If neither of the elements is a tuple, then just pair them; if one is an n-tuple but the other is not a tuple, then combine them into an $(n+1)$-tuple in the obvious way; and if both are tuples—an n-tuple and an m-tuple—then combine them into an $(n+m)$-tuple. With this rule, combining (x,y) with z would yield (x,y,z), as would combining x with (y,z). But what if $((x,y),z)$ were actually the desired result? The given rule cannot generate nested tuple structures. Complex-object algebras that treat tuples differently from other objects pay a price in lost simplicity and flexibility.

The parameterization of *crossfold* by functional arguments permits the use of whatever combining rule is suitable in a given context. Although *crossfold* is not associative in general, the following identity allows us to achieve the effect of associativity in a restricted use of *crossfold* corresponding to typical joins in complex-object algebras:

$$\begin{aligned}&(xs\ \mathit{crossfold}(f_1, g_1, \mathit{id})\ ys)\ \mathit{crossfold}(f_2 \circ \mathit{snd}, g_2, (\lambda((x,y),z).(x,y,z)))\ zs \\ =\ & xs\ \mathit{crossfold}(f_1, g_1 \circ \mathit{fst}, (\lambda(x,(y,z)).(x,y,z)))\ (ys\ \mathit{crossfold}(f_2, g_2, \mathit{id})\ zs)\end{aligned}$$

This is not an especially readable identity, so let us state its significance in English: Using *crossfold*, we may implement a 3-way (or, for that matter, an n-way) join yielding a collection of triples (or n-tuples), without being constrained to perform the constituent two-way joins in any particular order.

Another special case of *crossfold* that is associative is the intersection operator:

$$xs \wedge ys = xs\ \mathit{crossfold}(\mathit{id}, \mathit{id}, \mathit{fst})\ ys$$

However, it should be noted that this definition of intersection has some undesirable properties. The most bothersome of these is that $xs \wedge xs$, rather than always yielding xs, may yield a larger list if xs contains duplicates. One might dismiss the problem on the grounds that intersection is not a sensible operation for lists (as opposed to sets). Alternatively, one might question whether the *crossfold* we have defined here is

the right choice of primitives for generalizing join. A further reason to doubt this choice is that this version of *crossfold* appears to be incapable of expressing list difference.

3.7 Bulk Operators for Multisets and Sets

So far we have discussed bulk operators only on lists. If we reinterpret *fold* as $fold_{Multiset}$, $+\!\!+$ as $+\!\!+_{Multiset}$, and $=$ as $=_{Multiset}$, and change the list literals into multiset literals, then the foregoing discussion of *fold* and its uses carries over from lists to multisets. We need only add one proviso: Since multisets are unordered, we require that $\oplus$ be commutative (as well as associative) for $fold_{Multiset}(u, f, \oplus)$ to be well-defined.

Sets present more of a problem. Just as we have done with multisets, it would be possible to support sets with another variant of *fold*, together with $+\!\!+_{Set}$ (which would be set union). However, the set versions of *fold* and $+\!\!+$ do not constitute a good model for efficient computation on sets; moreover, the algebraic properties of sets are somewhat different from those of lists and multisets.[4] We can evade these difficulties by treating Set as a derived type.

Several set representations are possible, among which the most straightforward is probably a multiset without duplicates. To support this representation, a duplicate-elimination operator is needed. However, it may be just as well to define *dup_elim* as a special case of another operator, the grouping operator that maps $\{a, a, b, c, a, c\}$ to $\{\{a, a, a\}, \{b\}, \{c, c\}\}$. We may conceive of this operator as a new primitive on lists and multisets called *group*. In addition to its list or multiset argument, *group* would take a functional argument to define equivalence classes of the list or multiset elements by mapping each element to a value in some domain of the user's choice. Elements mapping to equal values would be deemed equivalent and would be grouped together. A *map* over the result could be used to extract a representative element from each group of duplicates. It would also be possible to wrap the *group* and *map* operations together by postulating an additional functional argument for *group*.

Note that one operator derivable from *group* would be a *split* operator that would separate a collection over a union sort into collections over the constituent types in the union sort.

It is unclear what is the best way to axiomatize *group* and *dup_elim*. But it is important to recognize, either as an axiom or as a consequence of different axioms, the identity

$$dup\_elim_{\tau}(xs +\!\!+ ys) = dup\_elim_{\tau}(xs) \cup dup\_elim_{\tau}(ys)$$

(where τ is List or Multiset), which asserts that *dup_elim* is a homomorphism. From Law 2 we therefore

[4] For commentary on some of the differences, see Section 4.5.

obtain

$$dup\_elim_\tau\ (xs\ fold([\,], f, +\!+)) = xs\ fold(\{\}_{Set}, dup\_elim_\tau \circ f, \cup)$$

This equation, applied in the reverse direction, can save execution time by replacing $\cup$ with the much cheaper $+\!+$. Applied in the forward direction, it can reduce the size of intermediate results.

3.8 Bulk Operators for Arrays

Again we may adopt the material that applies to lists in the foregoing, and apply it to one-dimensional arrays, so long as we postulate a *fold* for arrays. This is easily done, but in the case of arrays, a more appropriate primitive would probably be a variation of *fold* we will call *foldi*:

$$[a\ b\ c]_{Arr}\ foldi(u, f, \oplus) = f(0, a) \oplus f(1, b) \oplus f(2, c)$$

In other words, *foldi* reduces over (*index, array element*) pairs rather than over array elements alone. Pairing the index values with the elements makes it possible to collect a set of indexes of array values satisfying some criterion, or to find the position of a known element in a sorted array.

We may define *fold* in terms of *foldi* as

$$xs\ fold(u, f, \oplus) = xs\ foldi(u, f \circ snd, \oplus)$$

We also have, among other derived operators,

$$xs\ subarray(low, high) = xs\ foldi([\,]_{Arr}, (\lambda(i, x).\text{if } low \leq i \leq high \text{ then } [x]_{Arr} \text{ else } [\,]_{Arr}), +\!+_{Arr})$$

although it is plain that *subarray* would be more efficiently implemented as a primitive.

These operators, *foldi* and *subarray*, and the immediate derivatives of *fold* cited previously, provide fairly general support for array processing. However, these are all one-dimensional operators. It is possible to express multidimensional array operations in terms of them, but only if the multidimensional arrays involved are represented as one-dimensional arrays of arrays. A suitable choice of operators for direct manipulation of multidimensional arrays is beyond the scope of the present paper.

3.9 Bulk Operators for Tree Structures

In relational systems, tree and graph structures in the user's conceptual schema must be modeled using tuples as nodes and foreign keys as edges. Object-oriented systems permit more direct modeling of these structures by using objects as nodes and object references as edges.

In both models, however, expressing queries that involve tree traversal is problematical. One of the difficulties is that there is no distinction enforceable in these models between a tree and an arbitrary graph. If traversals could be expressed, their complexity would be unbounded unless nodes were marked as they were visited.

Our data model allows for user-defined tree structures as pure values that involve no explicit keys or references. Being trees, these structures are guaranteed acyclic, and the time to traverse them is linear in the number of nodes they contain. Pure-value trees may well be the best representation for many complex structures in engineering, even when those structures have shared subcomponents; note that a tree may be structurally a pure value and still have object references embedded within it.

Trees can be defined with recursive discriminated unions. As an example, consider the following type definition for a possibly empty binary tree:

$$\text{datatype } \alpha \text{ Bintree} = \text{Empty} \mid \text{Leaf of } \alpha \mid \text{Node of } (\alpha \text{ Bintree}) * (\alpha \text{ Bintree})$$

This recursive definition says a tree may be *empty*, or it may consist of a single *leaf* (in which case this leaf is the root), or it may be a *node* with two subtrees obeying the Bintree definition. The type variable α denotes the type of the values associated with the leaves. Note that internal nodes have no values associated with them—they only have subtrees.

Suppose the leaf values of a Bintree were integers and one wished to define a reduction function to sum the squares of all the leaves in a tree. This function would yield 0 for the *empty* tree; it would apply *square* to the value in a tree consisting of just a single *leaf*; and for a tree whose root was a *node*, it would apply + to the results of recursively applying the reduction to the subtrees.

Thus, this particular reduction would be characterized by the triple $(0, square, +)$. Other reductions over type Bintree can also be characterized by triples in an analogous way. We therefore define $fold_{Bintree}$ to carry out the Bintree reduction characterized by a triple given as *fold*'s argument. Summing the squares of the leaves of a Bintree called *tree* would be achieved by

$$tree\ fold(0, square, +)$$

As this example suggests, $fold_{Bintree}$ can be used in similar ways to $fold_{List}$, $fold_{Multiset}$, and so on. It can be used to build up new trees as well, though we will not go into the details here.

Each recursive type induces a different version of *fold*. It happens that $fold_{Bintree}$ closely resembles $fold_{List}$ and $fold_{Multiset}$, but that is only because for the purposes of reduction we have been treating lists

and multisets as though they were binary trees. In our discussion of $fold_{List}$, our assumption of associativity of $\oplus$ was a way of saying that we wanted to be able to view a list as the frontier of *any* binary tree with the right number of leaves—with each leaf corresponding to one element of the list—rather than being confined to a particular tree structure and hence a particular reduction order. In a manner of speaking, we have been using $fold_{Bintree}$ all along, even though the binary trees themselves were not introduced until just now.

The argument to $fold_{Bintree}$ has three components precisely because there are three alternatives in the discriminated union defining the data type Bintree. Reductions over discriminated unions with n alternatives can be characterized by n-tuples, and the generalization of *fold* to a type defined by such a discriminated union would accept an n-tuple as its argument. For more details on reduction operators for recursive types, see the excellent treatment by Pierce et al. [PDM89].

The laws applicable to versions of *fold* induced by recursive types can be generated mechanically. Again, we will not go into the details, but for a flavor of the relationships involved, note that Equations *i–iii* in Section 3.3 correspond to the three alternatives in the type definition for Bintree. Note also that Laws 1 and 2 (Section 3.4) bear a kinship to Equations 1 and 2 for case expressions (Section 3.1)—Equation 1 and Law 1 say that particular instances of *case* and *fold* are the identity function, while Equation 2 and Law 2 allow a function application to be pushed inside a *case* or *fold* expression. These similarities are not coincidental: rather, they arise from the fact that *fold* is shorthand for a recursive case expression over a recursive union. Consequently, all laws for case expressions have analogues for *fold*, and vice versa.

We have not supplied extensive examples to demonstrate the expressiveness of *fold* on user-defined tree structures. But the versatility of the *fold* message illustrated in previous sections is an indication that this single message may well be capable of expressing a wide variety of queries over a wide variety of structures.

3.10 Bulk Conversions and Aggregates

There are a number of operations that require special support in traditional and even in object-oriented databases, but that are handled very naturally by *fold*.

Among these are certain conversions between bulk types, which become almost trivial. For example,

$$fold(\{\}_{Multiset}, (\lambda x.\{x\}_{Multiset}), +\!\!+_{Multiset})$$

converts a list, array, or binary tree into a multiset. Other conversions are similar. However, *fold* conversions from multisets to lists or arrays are technically disallowed because $+\!\!+_{List}$ and $+\!\!+_{Arr}$ are not commutative. Thus, a *sort* operator and probably also a nondeterministic *arbitrary_order* operator would be useful for

carrying out these conversions.

Aggregate computations such as averages are supported in the relational algebra through *ad hoc* extensions, but with our approach, no such extensions are necessary. We have already used summations in our examples. If we use *fold* to accumulate the pair (*partial sum*, *partial count*) rather than just the *partial sum*, the result of the *fold* will be a (*sum*, *count*) pair, from which the average is immediate. Variances, maxima and minima, and other aggregates are all easily computed by reductions.

3.11 Summary

In this section we have described the salient features of our proposed algebra. Our algebra might be characterized as a functional language without recursion, to which a handful of highly versatile operators on bulk types have been added. Multiple bulk types are supported, and to a large extent they are handled in analogous ways.

The algebra obeys laws that can serve as the foundation for a variety of optimizing transformations, including familiar transformations from relational query optimization. Moreover, the operators we have defined are suitable for evaluation in a database implementation.

Much of what we have described revolves around a single family of operators named *fold*. We have argued that the algebra needs several additional bulk operators, but precise definition of these additional operators has been left to the future.

4 Related Work

The algebra sketched here has some similarities with other algebras that have been proposed. In discussing related work we will see some of these similarities; we will also see what is novel in the present approach, and the ways in which it improves on the alternatives in expressiveness and amenability to optimization.

4.1 EXCESS/EXTRA

Vandenberg and DeWitt [VD90] describe a complex-object algebra that shares some of the same goals as the present work. We shall refer to their algebra as the EXCESS/EXTRA algebra, after the system it is a part of. This algebra was intentionally designed to be equivalent to the EXTRA query language in expressive power.

The EXCESS/EXTRA algebra assumes a data model in which several general type constructors are provided, and data structures are built through free composition of those constructors. Like ours, the EXCESS/EXTRA algebra defines a small number of powerful bulk operators that allow multisets and arrays to be processed in analogous ways. Because our approach has much in common with theirs, and to some extent builds upon their ideas, here we will focus on some of the differences between the approaches.

Multidimensional arrays are not directly supported by EXCESS/EXTRA. Users may construct them as arrays of arrays or using other representations, but we see the absence of primitive support for them as a potential barrier to efficient implementation of dense arrays.

The EXCESS/EXTRA algebra is many-sorted, but scalar types are all grouped together in the same sort. Thus, integers, booleans, and so forth, are all treated simply as "values," and are not further distinguished; nor are operators provided on "values." As in the relational algebra, boolean expressions appear in the algebra, but only as subscripts to operators over some other sort. Consequently, a boolean expression at the top level of a query cannot be optimized using identities of the algebra. The same applies to boolean subexpressions of a query if they are not attached to other operators, as well as to integer subexpressions, string subexpressions, and so on. The lack of operators and identities for scalar types concerns us because a complex query, especially after methods are revealed, may contain many subexpressions of many sorts. Whenever possible we would like to be able to optimize these queries in their entirety, as we explained at the outset.

In the EXCESS/EXTRA algebra there are general operators called SET_APPLY and ARRAY_APPLY that are equivalent to our *map*. In fact, they are made more powerful through a sleight of hand. The function that is applied to each element of a collection may, for some of those elements, return the special value *dne*, for *does not exist*. These result values are discarded from the result collection, so SET_APPLY or ARRAY_APPLY need not preserve the cardinality of a collection. This makes it possible to implement *filtermap* and *select* with SET_APPLY, but still does not give SET_APPLY the generality of *fold*. One important reduction, flattening an array of arrays, or a multiset of multisets, is addressed with special operators SET_COLLAPSE and ARRAY_COLLAPSE. However, other reductions, like summing the elements of an array, are apparently not provided. It is unclear whether a user could write a general, type-safe *fold* in the data manipulation language.

4.2 ENCORE

Shaw and Zdonik [SZ89a] describe an algebra for the ENCORE object-oriented database system. They characterize all types as abstract data types whose implementations are hidden from the algebra. Axioms on the abstract data types assist the optimizer without revealing implementations. At the same time, Shaw and Zdonik allow for the possibility of making use of "optimization strategies for encapsulated behaviors," citing [GM88].

ENCORE's data model provides only two built-in parameterized types, Tuple and Set. Consequently, bulk data cannot be constructed without sets. Given this limitation, processing of multisets, lists, and arrays is likely to be costly.

A more fundamental difference between the ENCORE data model and our own lies in the treatment of object identity. ENCORE, in the tradition of pure object-oriented languages, views everything as an object with an identity. An opposing viewpoint is that there is a distinction between *objects*, which possess identity, and *values*, which do not [D+91]. For atomic types such as booleans and integers, the distinction is only a matter of terminology, but for compound types the distinction is important. If an ENCORE user creates two separate tuples $x = \langle A : 2, B : true \rangle$ and $y = \langle A : 2, B : true \rangle$, then x and y, though equal in value, will be distinguishable by an identity test. However, in systems that treat tuples as values, x and y would be indistinguishable. For us, too, x and y are indistinguishable since neither one contains any *ref* cells. (See Section 2.6.) Thus, we support structured values without identity.

The ability to generate values without identity is a great asset in query optimization. For example, it is the indistinguishability of multiple instances of the same value that makes it possible to optimize

$$g(f(a), f(a)) \qquad \text{to} \qquad (\lambda z.g(z,z))(f(a))$$

assuming f has no side-effects. But in ENCORE, these two expressions could not be guaranteed to yield the same result in general. In fact, given ENCORE's object semantics, very few algebraic transformations can be applied to a query without changing its meaning.

To deal with this difficulty, Shaw and Zdonik introduce new notions of equivalence of queries [SZ89b]. The roughest form of equivalence they define is *weak equivalence*; it holds on any pair of queries whose results contain the same database objects, glued together in any manner. Thus, if o_1 and o_2 are database objects, then a query returning $\{o_2, \{\{o_1\}, o_2\}\}$ and one returning $\{o_1, o_2\}$ are weakly equivalent. Under this liberal definition of equivalence it again becomes relatively easy to transform a query into a different but equivalent query. However, if the user frames a query so as to structure the result in a particular way, a result with an

entirely different structure will hardly do. Thus, one needs somewhat stronger notions of equivalence that still admit query transformations.

Accordingly, Shaw and Zdonik define a family of relations called *i-equality* or *equality at depth i*. This family of equalities generalizes the familiar object-oriented concepts of identity, shallow equality, and deep equality: Identity is equality at depth 0, shallow equality is equality at depth 1, and deep equality may be thought of as equality at depth ∞. Two queries are *i-equivalent* if their results are always i-equal; they are *id-equivalent at depth i* if, in addition, their results have isomorphic graph structures. Shaw and Zdonik present a number of query transformations that preserve id-equivalence at depth 2. Using such transformations, one can optimize a query without losing much structural information from its result.

One objection to ENCORE's complicated approach to equivalence is that it could be difficult for users to master. A second, more technical objection concerns composition of equivalences. Suppose we have a transformation rule that says query Q is k-equivalent to R for some $k > 0$, and let f be some function. What can we say about the relationship between $f(Q)$ and $f(R)$? In general, we cannot even assure weak equivalence of these two expressions, since f may perform computations sensitive to the structure of its argument. In other words, we may not freely substitute equivalents for equivalents in subqueries. Similarly, if the result of one query may be referred to in subsequent queries, then the initial query may safely be transformed only in ways that preserve id-equivalence at depth 1.

The most interesting contribution of ENCORE is its provision of query transformations that involve inverse relationships between database collections. These transformations make it clear that vast improvements can be achieved on some queries by taking advantage of inverse relationships. Such transformations differ from other algebraic transformations in that they rely not only on the properties of the algebraic operators, but also on integrity constraints on the underlying database.

4.3 An Algebra Based on FP

A general and abstract approach to algebraic manipulation of bulk types is suggested by Beeri and Kornatzky [BK90]. Their treatment of high-level operators is similar to ours, but their framework is somewhat different. Asserting that "[f]unctional languages such as Lisp with higher-order functions do not possess a useful algebra of programs due to the unrestrained use of these higher-order functions," they base their algebra on Backus's FP. To FP's tuple constructor they add constructors for sets, multisets, lists, arrays, and trees, but they treat all of these constructors in a uniform way—at least as far as the bulk operators are concerned. The distinctions they make between the constructors are expressed through axioms. For example,

a *permutability* axiom on sets says that $\{o_1, o_2, \ldots, o_n\} = \{\rho(o_1, o_2, \ldots, o_n)\}$, where ρ is any permutation and {} denotes the language's set constructor. Similarly, sets satisfy a *duplicate elimination* axiom that says $o_i = o_j \Rightarrow \{o_1, \ldots, o_i, \ldots, o_j, \ldots, o_n\} = \{o_1, \ldots, o_i, \ldots, o_{j-1}, o_{j+1}, \ldots, o_n\}$. In this way their model neatly relieves the bulk operator definitions of the need to distinguish between constructor types. On the other hand, it is not clear how such a model might be efficiently implemented.

The bulk operators themselves are familiar: *apply-to-all* (*map*), *product* (a kind of generalized Cartesian product), and *pump* (similar to *fold*). Beeri and Kornatzky achieve selection or *filtering* by using apply-to-all with a function that yields *null* for items to be discarded. A constructor axiom for *null elimination* has the effect of removing the unwanted items. The concept here is of course the same as the use of *does not exist* in EXCESS/EXTRA.

The *null* values are also used to distinguish internal nodes of a tree from its leaves. If a node's subtrees are *null*, then it is a leaf. The lack of explicit discriminated unions to distinguish different node types restricts the kinds of trees that can be described conveniently. This limitation also leads to cumbersome implementations of simple operations, despite the power of the *pump* operator. For example, to sum the leaf values of a binary tree, ignoring internal node values, entails passing *pump* an argument function of surprising messiness.

4.4 Maps

Atkinson et al. [ALR91] take a very different approach to combining minimality with generality. They define a single bulk data type, called a *map*, that can be specialized to any other bulk type one might be interested in: finite functions, arrays, relations, other kinds of sets, and so on.

Although the authors describe relations as just one of many bulk types that maps can represent, one can also turn this view upside down and take relations as a starting point. Then maps can be characterized as relations whose unique-key constraints are enforced, and whose attribute types are arbitrary. For example, a relation can be used as a *function* from its key to its other attributes. If the key is an integer, the function represents an *array*. A *set* is simply a relation whose key includes all its attributes. For efficiency reasons, these relations implementing maps of different kinds are maintained in key-sorted order, and in some operations this order affects semantics.

Maps come with both imperative and algebraic operations. Among the latter are *union*, *intersection*, and *difference*, which are almost the same as their relational counterparts. However, *union* and *intersection* are not commutative if the operand maps, viewed as relations, have keys in common. There is a *generation*

construct that achieves selection, projection, and so forth, using a syntax akin to that of list comprehensions. We comment briefly on comprehensions in the next section.

4.5 Abstract Bulk Types

4.5.1 Comprehensions, Quads, and Ringads

List comprehension notation, found in several functional languages, allows lists to be generated in a concise, declarative manner that resembles mathematical set former notation. Wadler [Wad90] has given an algebraic characterization of comprehensions that can be motivated by category theory. Trinder and others have explored the use of comprehensions as a database query notation [Tri91, HN91]. Trinder emphasizes the possibilities for query optimization offered by the algebraic structure underlying comprehensions.

In Trinder's treatment, the possible manipulations on a bulk data type are distilled to a quadruple of functions. Such a quadruple, provided it is associated with a type constructor and that its components satisfy eight algebraic laws, is called a *quad*. As an example, the four functions for the List constructor would be *map*, $\lambda x.[x]$, *flatten*, and $\lambda x.[]$. Evaluation of a List comprehension involves translating the comprehension into an algebraic expression built from these four functions.

Quadric descriptions also exist for sets, multisets, and some trees. Each such description includes four functions, which we will refer to generically as *map*, *single*, *flatten*, and *empty*. A comprehension for a given bulk type can always be translated into a quadric expression; thus, comprehensions provide nothing in expressiveness that cannot be obtained by other means. (What comprehensions do provide is notational convenience—an important consideration in the design of user languages, but less important to the construction of an algebra.)

Of concern to us here is the expressiveness of the quads themselves. Clearly they can express *map* and *flatten*, since these functions are built into the quad definition. They can also express selection, as

$$select\ p = flatten \circ map(\lambda x.(\text{if } p\ x \text{ then } single \text{ else } empty)\ x)$$

but they cannot express general reductions or conversions. More fundamentally, there is no way, using the quad functions, to express $+\!\!+$; indeed, there is no way to create any bulk type instance with cardinality greater than 1.[5] For the quadric description of a bulk type to be useful, one needs some mechanism external to the quad to build up instances of that bulk type. Trinder assumes the existence of $+\!\!+$ but does not

[5] One could define $a +\!\!+_\tau b = flatten_\tau(map_\tau(\lambda x.\text{if } x \text{ then } a \text{ else } b)[true, false]_\tau)$, provided one assumed the availability of the constant $[true, false]_\tau$; but the construction of any such constant from the quad components alone is impossible.

relate it to the quad components by any algebraic laws. One has no assurance that the $+\!\!+$ operators for different bulk types will have similar algebraic properties. Thus, the quad structure by itself is inadequate to characterize bulk types in an abstract, encapsulated way.

In recent collaborative work of Watt and Trinder [WT91], the quad is supplanted by a richer algebraic structure, the *ringad*. Before we comment on ringads, however, it is worth noting that sections of the Watt and Trinder paper overlap rather closely with the present work. Here we shall point out some of the differences. First, Watt and Trinder treat both *flattenmap* (which they call *iter*) and *fold* (which they call *reduce*) as primitive collection operators, rather than deriving *flattenmap* from *fold*. Second, they treat sets, multisets, and lists (and more) in a uniform manner; in our treatment, sets are problematical. Third, Watt and Trinder offer no laws as general as our Laws 2 and 2*a*.

These differences are actually related to one another. To see the connections, let us review our thinking about sets. Part of our reason for separating sets from multisets and lists is that set operations tend to be costly to evaluate; by decomposing set operations into multiset operations and duplicate eliminations, we give ourselves optimization opportunities that are unavailable if set operations are treated as atomic. But from a theoretical standpoint as well, it is difficult to reconcile the behavior of sets with that of multisets and lists. For example, taking $+\!\!+_{Set}$ as set union, and defining $fold_{Set}$ by Equations *i*–*iii*, we find (using the fact $A \cup A = A$) that

$$2 = \{2\}fold(0, id, +) = (\{2\} \cup \{2\})fold(0, id, +) = (\{2\}fold(0, id, +)) + (\{2\}fold(0, id, +)) = 2 + 2 = 4$$

In general, $fold_{Set}$ is ill-behaved. However, $fold_{Set}(\bot, f, \sqcup)$ is well-behaved when $\sqcup$ is the *join* operation of some lattice, and $\bot$ is its bottom. (Dually, $fold_{Set}(\top, f, \sqcap)$ is also well-behaved.) In particular, $flattenmap_{Set}(f) = fold_{Set}(\emptyset, f, \cup)$ is always well-behaved, since set union is the join operation in the lattice of sets over a given type.

Watt and Trinder avoid difficulties with sets by basing most of their discussion on *flattenmap* rather than *fold*. Since *flatten*, *map*, and *select* can all be expressed in terms of *flattenmap*, Watt and Trinder achieve considerable generality in this way. They recognize the utility of *fold* as well, and provide it as another operation, but make no promises about its algebraic properties. Thus, *fold* plays a much less important role in their treatment than in ours. It is then natural that our Laws 2 and 2*a*, which are laws about *fold*, should be absent from their treatment.

The algebraic structure used by Watt and Trinder, as we have noted, is called a ringad. The components of a ringad are $\langle \tau, single_\tau, flattenmap_\tau, +\!\!+_\tau, [\,]_\tau \rangle$, where τ represents some bulk type. A rich set of laws

relates these components and assures that they interact in a way that most of us would consider reasonable. Note, though, that *fold* is not included in the ringad, nor can it be constructed from the functions that are. Just as ++ operated externally to the quad abstraction, *fold* operates externally to the ringad abstraction. Thus, the ringad, while a step forward, still falls short of providing a complete abstract model of bulk types.

4.5.2 Adjunctions

As we mentioned, Wadler's algebraic characterization of comprehensions drew on ideas from category theory [Wad90]. In fact, many algebraic structures in computer science have descriptions in terms of category theory. The existence of such descriptions can be reassuring: seemingly complicated or arbitrary laws, rewritten as categorical abstractions, are seen to be instances of simple, regular patterns (e.g., [Spi89]). It is therefore worth asking whether the equations and laws on *fold* presented in this paper admit a categorical description.

The answer is that they do. The applicable categorical notion is the *adjunction* [Spi90]. Adjunctions do not characterize structures so much as the relationships between structures. Through the adjunction framework we see bulk types as transformations from potentially unstructured element types to structured algebraic entities. For example, since every list is a monoid, regardless of the type of its elements, lists may be said to constitute a transformation from types to monoids. The structure embedded in that transformation is described by an adjunction.

Let $\mathcal{X}$ be TYPE, the category of types (its arrows are functions), and let $\mathcal{A}$ be MONOID, the category of monoids (its arrows are monoid homomorphisms). Then we may define an adjunction $\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \rightharpoonup \mathcal{A}$ as follows:

$$
\begin{aligned}
F &: \mathcal{X} \to \mathcal{A} \\
F &\text{ is } \begin{cases} \tau \mapsto \langle \tau List, +\!\!+, [\,] \rangle & \text{on objects} \\ map & \text{on arrows} \end{cases} \\
U &: \mathcal{A} \to \mathcal{X} \\
U &\text{ is } \begin{cases} \langle \tau, \oplus, u \rangle \mapsto \tau & \text{on objects} \\ id & \text{on arrows} \end{cases} \\
\eta &: Id_{\mathcal{X}} \dot{\to} UF \\
\eta_\tau &= \lambda x : \tau . [x]_{List} \\
\varepsilon &: FU \dot{\to} Id_{\mathcal{A}} \\
\varepsilon_{\langle \tau, \oplus, u \rangle} &= fold(u, id, \oplus)
\end{aligned}
$$

We may define a slightly different adjunction by taking $\mathcal{A}$ to be the category of *commutative* monoids, and by replacing each occurrence of List by Multiset.

To show that the foregoing equations do indeed define an adjunction, we must verify that F and U are functors, and that η and ε are natural transformations. We must also verify these adjunction laws:

$$\begin{aligned} U\varepsilon \cdot \eta U &= Id_U \\ \varepsilon F \cdot F\eta &= Id_F \end{aligned}$$

For the interested reader, we sketch what is involved in carrying out these verifications. That F is a functor follows from Law 2; U is trivially a functor. The naturality of η comes from Equation *ii*; that of ε, from Law 2. The law $U\varepsilon \cdot \eta U = Id_U$ is obtained as a special case of Equation *ii*, and $\varepsilon F \cdot F\eta = Id_F$ is a restatement of Law 1.

We have shown that from the type constructor for List (or Multiset), together with *fold* and ++ operations obeying a handful of laws, we can construct an adjunction. Equally important, although we will not demonstrate it, is a converse of sorts: given that $\langle F, U, \eta, \varepsilon \rangle$ is an adjunction from TYPE to MONOID, we could have extracted ++ from F, and defined *fold* in terms of F and ε, and then immediately obtained Equations *i*–*iii* and Laws 1, 2, and 2*a*—without induction.

It is tempting to conclude from these observations that adjunctions express the essential structure of bulk types. Adjunctions do appear to have one major advantage over ringads, in that they integrate *fold* with the ringad operations in a single, coherent structure. The greater structure of the adjunction makes it less flexible; but we can regain flexibility by considering a *family* of adjunctions, each with analogous but slightly different algebraic properties. Thus, in the foregoing, we obtained two different adjunctions depending on whether $\mathcal{A}$ was MONOID or COMMUTATIVE-MONOID: one adjunction characterizing lists, the other characterizing multisets. We can go further and obtain a characterization of sets by taking $\mathcal{A}$ to be POINTED-LATTICE (the category of lattices with bottom; its arrows are lattice homomorphisms). However, it is unclear whether other bulk types (e.g., tree-structured types) can be described within the adjunction framework.

4.6 Other Algebras and Systems

FAD [BBKV87] introduced the *pump* operator, which is discussed above and is essentially the same as our *fold* operator applied to nonempty binary trees. The cited paper points out the usefulness of *pump* for parallel reduction, but does not take full advantage of its generality: a *filter* function (our *filtermap*) is

defined separately rather than being derived from *pump*.

Merrett and collaborators have extended the relational algebra to provide more of the expressiveness of general-purpose programming languages, and have implemented a number of scientific algorithms in their system [MK91].

Breazu-Tannen and Subrahmanyam [BTS91] have investigated the algebraic and logical foundations of operations on the flat bulk types. They employ a categorical framework different from the one discussed above in Section 4.5.2.

Straube and Özsu [SÖ90b, SÖ90a] have developed a set-based object-oriented query algebra, and have presented transformation rules for it. However, their algebra is less expressive than others we have examined.

The Object-Oriented Functional Data Language [MCB90] synthesizes ideas from functional and object-oriented programming, but does not concern itself with database efficiency issues.

The SETL project [SSS81] investigated optimization of programs using data abstractions through selection of suitable representations for the abstractions. However, the project focused on in-memory representations and did not consider secondary storage.

4.7 Data Modeling Issues

Our conception of subtyping is ultimately based on the pioneering work of Cardelli [Car84, CW85]. A survey of subsequent ideas about typing of object-oriented systems is given by Danforth and Tomlinson [DT88]. Still more recently, interesting work on typing of object-oriented databases has been done by Ohori, Buneman, and Breazu-Tannen; e.g., [Oho90b, Oho90a, BTBO89].

VBASE [AH90] deserves mention in connection with discriminated unions. Atypically for an object-oriented system, VBASE provided discriminated unions as a data structuring facility, and, as a separate feature, also allowed variable types to be unions of different types supporting the same messages. (These type unions are not discriminated from the user's point of view.) The VBASE treatment of these matters is very similar to our own.

Straube and Özsu [SÖ90c] have studied the problem of type unions in some detail. An important application for type unions arises in the typing of elements of a heterogeneous set. Naive approaches force the elements of such sets to be given a general type like Object, or some other type high up in the type hierarchy. This can result in the loss of type information. For example, without union types, if A and B are sets then the element type of $A \cup B$ must be a common ancestor of the element types of A and B. With union types, the element type of the set union can be taken to be the union of the element types of the two

sets. The authors apply analogous thinking to other operators, and develop a set of type inference rules to determine the most informative type for any expression.

5 Future Work

The present work has given less attention to arrays than we would have liked. Future work includes looking more closely at the ways that operations on matrices with various representations can be efficiently implemented using more primitive and general operators over multidimensional arrays. The hope is that matrix and other array operations can be boiled down to a small number of primitives without sacrificing efficiency.

In this paper, we have not discussed the part that index structures like B-trees can play in operations like selection. There is some question whether index structures should participate in the algebra. Usually they do not, but there may be benefits in including indexing abstractions in the algebra. They would allow us to duplicate the functionality of *maps* [ALR91] with reasonable efficiency. Indexes might also be necessary to support inverses as in ENCORE.

6 Conclusion

There has been an effort in recent database algebra proposals to make the primitive operators very simple and general. In this paper we have taken several unconventional steps in our pursuit of a simple and general object-oriented algebra. We have emphasized the role of union types, both in modeling subtyping and in modeling complex data. Our algebraic operators have not been limited to bulk types, but have supported scalar types and "small" constructed types as well. We have sketched how a minimal set of bulk primitives, parameterized by expressions containing arbitrary operators, can support a wide variety of traditional database operations and optimizations.

Certain details of our algebra remain in doubt. But there is little doubt that we want our algebra to provide a small number of very general operators over general, freely composed types. The present work argues for the viability of this approach.

7 Acknowledgements

I am grateful for discussions with David Maier, and with Revelation project participants Scott Daniels, Tom Keller, and Duri Schmidt. Thanks also to many OGI students and faculty, notably Jim Hook and Steve

Otto, for discussions on related topics. Françoise Bellegarde kindly took the time to read and comment on an earlier draft of this paper.

References

[AH90] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 186–196. Morgan Kaufmann, 1990. Appeared previously in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Orlando, FL, October 1987.

[AK90] Serge Abiteboul and Paris C. Kanellakis, editors. *Proceedings of the Third International Conference on Database Theory*, volume 470 of *LNCS*. INRIA, Springer-Verlag, December 1990.

[ALR91] Malcolm Atkinson, Christophe Lécluse, and Philippe Richard. Bulk types for data programming languages, a proposal. Technical Report 67-91, Altaïr, February 1991.

[BBKV87] Francois Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD, a powerful and simple database language. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 97–105. Morgan Kaufmann, September 1987.

[BK90] Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. In Abiteboul and Kanellakis [AK90], pages 72–88.

[BP87] J. L. Bell and G. S. Patterson, Jr. Data organization in large numerical computations. *The Journal of Supercomputing*, 1:105–136, 1987.

[BTBO89] Val Breazu-Tannen, Peter Buneman, and Atsushi Ohori. Static type-checking in object-oriented databases. *IEEE Data Engineering*, 12(3), September 1989.

[BTS91] Val Breazu-Tannen and Ramesh Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. Technical Report MS-CIS-91-29, University of Pennsylvania, April 1991. Also available in Proceedings of ICALP '91.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Giles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67. IFIP and ETACS, Springer-Verlag, June 1984.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[D+91] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10), October 1991.

[DGK+91] Scott Daniels, Goetz Graefe, Thomas Keller, David Maier, Duri Schmidt, and Bennet Vance. Query optimization in Revelation, an overview. *IEEE Data Engineering Bulletin*, 14(2), June 1991. Special Issue on Theoretical Foundations of Object-Oriented Database Systems.

[DT88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1), March 1988.

[FI68] A. D. Falkoff and K. E. Iverson. *APL\360 User's Manual.* IBM Thomas J. Watson Research Center, August 1968.

[GK90] Rajiv Gupta and Jim Kajiya. Compiler optimization of array data storage. Technical Report CS-TR-90-07, Caltech, April 1990.

[GM88] Goetz Graefe and David Maier. Query optimization in object-oriented database systems: A prospectus. In Klaus R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *LNCS*, pages 358–363. 2nd International Workshop on Object-Oriented Database Systems, Springer-Verlag, September 1988.

[Gol84] Adele Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley, 1984.

[HN91] Michael L. Heytens and Rishiyur S. Nikhil. List comprehensions in AGNA, a parallel persistent object system. Computation Structures Group Memo 326, M.I.T., March 1991.

[Lom76] David B. Lomet. Objects and values: The basis of a storage model for procedural languages. *IBM Journal of Research and Development*, 20(2), March 1976.

[Lom80] David B. Lomet. A data definition facility based on a value-oriented storage model. *IBM Journal of Research and Development*, 24(6), November 1980.

[Mai83] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[MCB90] Michael V. Mannino, In Jun Choi, and Don S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11), November 1990.

[MK91] T. H. Merrett and Bassem Khalife. Geographical information systems development using a general purpose database programming language. Technical Report TR-SOCS-91-2, McGill University, February 1991.

[Oho90a] Atsushi Ohori. Orderings and types in databases. In Francois Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, ACM Press Frontier Series, pages 97–116. Addison-Wesley, 1990.

[Oho90b] Atsushi Ohori. Representing object identity in a pure functional language. In Abiteboul and Kanellakis [AK90], pages 41–55.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.

[SÖ90a] Dave D. Straube and M. Tamer Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ90b] Dave D. Straube and M. Tamer Özsu. Query transformation rules and heuristics for an object algebra. Preprint of conference submission, 1990.

[SÖ90c] Dave D. Straube and M. Tamer Özsu. Type consistency of queries in an object-oriented database system. In Norman Meyrowitz, editor, *Proceedings of the European Conference on Object-Oriented Programming: Systems, Languages, and Applications.* ACM/SIGPLAN, October 1990.

[Spi89] Mike Spivey. A functional theory of exceptions. Technical Report JMS-89-10a, Oxford University Computing Laboratory, 1989.

[Spi90] Mike Spivey. Category theory for functional programming. Unpublished manuscript, 1990.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2), April 1981.

[SZ89a] Gail M. Shaw and Stanley B. Zdonik. An object-oriented object query algebra. In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages.* Oregon Center for Advanced Technology Education, Morgan Kaufmann, June 1989.

[SZ89b] Gail M. Shaw and Stanley B. Zdonik. Object-oriented queries: Equivalence and optimization. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 264–277, December 1989.

[Tri91] Phil Trinder. Comprehensions, a query notation for DBPLs. Unpublished manuscript, June 1991.

[VD90] Scott L. Vandenberg and David J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. Technical Report CS-TR 987, University of Wisconsin–Madison, December 1990.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78. ACM Press, June 1990.

[WT91] David A. Watt and Phil Trinder. Towards a theory of bulk types. Manuscript; to appear in expanded form as a FIDE Technical Report, June 1991.