

# Comprehensions, a Query Notation for DBPLs

Phil Trinder  
Department of Computing Science  
Glasgow University  
Glasgow G12 8QQ  
Scotland  
trinder@uk.ac.glasgow.dcs\*

## Abstract

This paper argues that comprehensions, a construct found in some programming languages, are a good query notation for DBPLs. It is shown that, like many other query notations, comprehensions can be smoothly integrated into DBPLs and allow queries to be expressed clearly, concisely and efficiently. More significantly, two advantages of comprehensions are demonstrated. The first advantage is that, unlike conventional notations, comprehension queries combine computational power with ease of optimisation. That is, not only can comprehension queries express both recursion and computation, but equivalent comprehension transformations exist for all of the major conventional optimisations. The second advantage is that comprehensions provide a uniform notation for expressing and performing some optimisation on queries over several bulk data types. The bulk types that comprehensions can be defined over include sets, relations, bags and lists. A DBPL can also be automatically extended to provide and partially optimise comprehension queries over new bulk types constructed by the application programmer, providing that the new type has some well-defined properties.

## 1 Introduction

Several database programming languages, or DBPLs, are emerging that enable programmers to create data-intensive applications with greater ease [1, 2, 3, 15, 19, 23]. Each DBPL must incorporate a query notation because interrogating the database is a common task. A query notation should be clear, concise, efficient and well integrated with its host language. Arguments are constructed that comprehensions, a construct found in some programming languages, have all of these properties.

It is not clear how powerful a query notation should be. In notations based entirely on the relational algebra for example, neither computational nor recursive queries can be expressed, but queries can be made more efficient using, *inter alia*, identities in the relational algebra. Conversely, queries written in a programming language can express computation and recursion, but are not so easily optimised. Some DBPLs, such as PS-algol and Napier, express queries as procedures, a computationally complete, but not easily optimisable, notation. Other DBPLs, such as Galileo and DBPL, have chosen an optimisable, but computationally incomplete, notation. Because Galileo and DBPL are computationally complete, computational and recursive queries can still be expressed using procedures, but often neither clearly nor concisely and automatic optimisation is lost.

Comprehensions are a query notation that is both computationally complete and easily optimised. Section 4 describes how the lambda calculus semantics of comprehensions provides smooth integration of recursive functions and also makes optimisation possible. To illustrate recursion and computation two well-known bill of material queries are expressed as comprehensions. A suite of list comprehension transformations that emulate all of the major relational query optimisations has been developed [25] and two of the transformations demonstrated [27]. To illustrate the optimisations two example transformations are given. Even queries

---

\*This work supported by both the SERC Bulk Data Types Project and the ESPRIT FIDE Project (BRA 3070).

entailing computation or recursion can be improved using a combination of comprehension transformation and program transformation. To illustrate the techniques, both the computational and non-computational parts of Date’s bill of material query are improved.

Many language designers believe that a DBPL should support a rich set of data types to enable the application-programmer to model the problem domain in a natural way. The data types that are queried contain large amounts of data and are termed bulk data types [5, 17]. Sets, relations, bags, lists, trees, graphs and finite maps are all examples of bulk types that might be found in a DBPL with a rich type system. Unfortunately, supporting several bulk types adds complexity to the language because constructs must exist to declare, populate, query and modify instances of each different bulk type. The problem is exacerbated in languages, for example those with orthogonal persistence, that permit the application-programmer to define new bulk types that are appropriate to the problem domain. Ideally the DBPL should be extended to express concisely, and optimise, queries over these new types.

The complexity of the application-programmer’s world is reduced if a single query notation can be defined over several, or all, bulk types. Comprehensions are just such a uniform notation because they can be defined over any type that is a *ringad*, i.e. that has four functions which satisfy certain algebraic laws [30, 31]. Sets, relations, bags, lists, some finite maps and some trees are all ringads. However, some useful bulk types are not ringads, for example trees with data at internal nodes. It is also not known whether some other bulk types, such as graph, form a ringad. For concreteness this paper initially considers comprehensions defined over lists.

Comprehensions apparently offer several other benefits to languages supporting several bulk types. A comprehension over a ringad bulk type is translated into invocations of the appropriate ringad functions. This uniform translation scheme for queries over ringad bulk types simplifies the compiler. Because some optimisations can be proved using just the ringad laws a single optimiser can perform some improvements on comprehensions over all ringad bulk types. If the bulk type is known to be unordered, for example a set or relation, then many more optimisations are possible. Hence it seems that at least two optimisers are desirable, and possibly one for each “class” of bulk types is necessary. Finally, if an application-programmer defining a new bulk type provides the compiler with the four ringad functions, the DBPL can be automatically extended to compile and optimise comprehension queries over values of the new type.

In related work Breuer, Nikhil and Poulouvasilis have all described functional databases and also recommend *list* comprehensions as a clear and powerful notation for expressing queries [7, 20, 22]. This paper is the first analysis of the potential of ringad comprehensions as a DBPL query notation. Work by both Nikhil and Heytens, and Kato *et al*, on the parallel evaluation of comprehensions is described in Section 3. Comprehensions are similar to DBPL access expressions [23]. The fundamental difference between the approaches is that the semantics of DBPL access expressions is based on set theory, whereas the semantics of comprehensions is based on the lambda calculus. Both theories permit the proof of useful transformations. However, the comprehension semantics can be easily extended to incorporate pure recursive functions, while it is not clear how to extend a set-based semantics to incorporate computation and recursion.

The remainder of this paper is structured as follows. Section 2 describes list comprehension query notation. Section 3 touches on parallel comprehension queries, then argues that comprehension notation is adequately brief, clear, efficient and that it can be cleanly integrated with DBPLs. Section 4 demonstrates that list comprehension queries are both computationally powerful and easily optimised. Section 5 shows how comprehensions can be defined over several bulk types. Section 6 concludes.

## 2 List Comprehensions

This section informally introduces list comprehension, or ZF, notation. A formal semantics of ringad comprehensions is given in Section 5 and a full description can be found in [29]. In mathematics a set comprehension describing the set of squares of all the odd numbers in a set  $A$  can be written

$$\{square\ x \mid x \in A \wedge odd\ x\}$$

and has a corresponding list comprehension

$$[square\ x \mid x \leftarrow A; odd\ x].$$

This can be read as ‘the list of squares of  $x$  such that  $x$  is drawn from  $A$  and  $x$  is odd’. The syntax of collection comprehensions is as follows, where  $E$  stands for an expression,  $Q$  stands for a qualifier,  $P$  stands for a pattern, and  $\Lambda$  stands for an empty qualifier:

$$\begin{aligned}
 E & ::= \dots \\
 & \quad | \quad [ E \mid Q ] \\
 Q & ::= \Lambda \\
 & \quad | \quad E ; Q \\
 & \quad | \quad P \leftarrow E ; Q
 \end{aligned}$$

The result of evaluating the comprehension  $[E \mid Q]$  is a new list, computed from one or more existing lists. The elements of the new list are determined by repeatedly evaluating  $E$ , as controlled by the qualifier  $Q$ .

A qualifier is either a *filter*,  $E$ , a *generator*,  $P \leftarrow E$ , or a sequence of these. A filter is just a boolean-valued expression, expressing a condition that must be satisfied for an element to be included in the result. An example of a filter was *odd x* above, ensuring that only odd values of  $x$  are used in computing the result. A generator of the form  $V \leftarrow E$ , where  $E$  is a list-valued expression, makes the variable  $V$  range over the elements of the list. An example of a generator was  $x \leftarrow A$  above, making  $x$  range over the elements of the list  $A$ . More generally, a generator of the form  $P \leftarrow E$  contains a pattern  $P$  that binds one or more new variables to components of each element of the list.

## Queries

List comprehensions easily express queries over relational [7, 24], functional [20, 22] and object-oriented databases [11]. For concreteness let us consider the relational model, specifically a database with three relations used by Ullman [32]:

```

MEMBERS (NAME, ADDRESS, BALANCE)
ORDERS (ORDER_NO, ONAME, OITEM, QUANTITY)
SUPPLIERS (SNAME, SADDRESS, SITEM, SPRICE)

```

Let us assume that the underlying implementation provides the following support for relations. The names of the relations are bound to the current list of tuples in the database. For each attribute of every relation there is a selector function that maps from a tuple in the relation to that attribute. For example, *balance* will select the BALANCE attribute of a MEMBERS tuple.

Given this support, the query “Print the names of the members with negative balances” can be written as the comprehension

$$[name \ m \mid m \leftarrow members; balance \ m < 0].$$

The query works in a straight-forward manner. Each tuple in MEMBERS is retrieved by  $m \leftarrow members$ . If the balance attribute (*balance m*) is less than zero, then the name attribute (*name m*) is included in the result.

The query “Print the supplier names, items and prices of all the suppliers that supply at least one item ordered by Brooks” can be written

$$\begin{aligned}
 & [(sname \ s, sitem \ s, sprice \ s) \mid o \leftarrow orders; oname \ o = 'Brooks, B.'; \\
 & \quad s \leftarrow suppliers; oitem \ o = sitem \ s].
 \end{aligned}$$

In the queries above selector functions have been used to locate the attributes of tuples so that any attribute not relevant to the query can be ignored. This is a substantial advantage for real databases that contain relations with many attributes. An alternative way of writing queries is to provide a pattern that matches all of the attributes. The first query can be written in this style as follows. The pattern-matching and selector-function styles are easily interchangeable.

$$[(name) \mid (name, address, balance) \leftarrow members; balance < 0]$$

## 3 Requirements

A query notation should be brief, clear, efficient and well integrated with the DBPL. To express queries that require greater power than the relational algebra (or calculus), comprehensions must be combined with the recursive functions of the DBPL. Such queries depend greatly on the brevity, clarity and efficiency of the DBPL. Hence the following arguments for the brevity, clarity and efficiency are made for comprehensions expressing queries *requiring no more power than the relational calculus*.

Although most existing DBPLs are sequential there are a few concurrent or distributed DBPLs [14, 15, 21]. Hence it is desirable, but not essential that a query notation should be easily made concurrent. The AGNA DBPL is evaluated on a dataflow multiprocessor and uses parallel list comprehensions to process database queries at speeds comparable with other multiprocessor database machines [21]. SPL is a language that uses comprehensions to evaluate queries in parallel over a distributed database [14]. It is not clear that the results reported here hold for AGNA and SPL comprehensions as both are non-standard: AGNA uses ‘open lists’ and SPL comprehensions are elaborated to increase efficiency and include communication primitives.

### 3.1 Brevity

Comprehension queries are brief because they are a declarative specification of the query. To express a relational calculus query the programmer only specifies the predicate that entities must satisfy rather than how to locate the required entities by a sequence of operations.

### 3.2 Clarity

How to express a query in the notation should be intuitively obvious, conversely, what a query means should also be immediately apparent. List comprehensions are clear in both senses because of their similarity to the relational calculus. The similarity arises because both notations are based on Zermelo-Fraenkel set theory. The correspondence is particularly evident between domain relational queries and pattern-matching list comprehension queries. The pattern matching version of Ullman’s first query was written

$$[(name) \mid (name, address, balance) \leftarrow members; balance < 0],$$

and the domain relational specification of the same query is

$$\{(name) \mid (name, address, balance) \in members \wedge balance < 0\}.$$

The similarity is not always this strong. For example, if  $\#$  is list concatenation, the calculus query

$$\{(t) \mid (t) \in R \vee (t) \in S\}$$

corresponds to the comprehension

$$[(t) \mid (t) \leftarrow R \# S].$$

### 3.3 Efficiency

A list comprehensions is efficient in the sense that it performs the minimum number of *cons* operations required to produce the result list [29]. Proofs of the efficiency of comprehensions over bulk types other than lists have not yet been attempted, but the lambda calculus semantics of comprehensions makes proof or disproof possible. Pragmatic evidence for the efficiency of list processing is given by commercial products that use it, for example in FQL [8], and by the speed of a prototype parallel database machine which uses list comprehensions [21].

### 3.4 Integration

#### Syntactic Integration

It is important that the syntax of the query notation is consistent with the syntax of the remainder of the language. For some languages a keyword style may be preferred to the bracket notation used in this paper. The first example query from Section 2 might be expressed in a keyword style as follows.

**for each m in members where balance m < 0 seq of name m end**

## Semantic Integration

The fundamental reason for the mismatch between many query notations and their host language is that the query notation is founded on relations, i.e. set theory, whereas the host language is founded on a von Neuman machine. In contrast comprehensions have the lambda calculus as a common foundation with a functional DBPL, and several implementations exist [7, 20, 22]. The lambda calculus is also the foundation of (non-side-effecting) expressions in procedural and object-oriented languages. Hence comprehensions integrate cleanly as expressions in a procedural or object-oriented language — they are simply transformed into a sequence of function calls or method invocations. A preprocessor using this technique has been constructed and permits the inclusion of comprehensions in PS-algol programs [27].

The interaction between comprehensions and side-effecting expressions is quite subtle. Many of the optimisations described in the next section are correct if the functions invoked have no side-effects or if the side-effects have certain properties, e.g. being commutative. It is hard for a compiler to guarantee that a function has no side-effects. Indeed in object-oriented languages it is impossible to guarantee at compile-time that a method invocation will have no side-effects because late binding permits the method to be redefined before the comprehension is executed. Hence it becomes the programmer's responsibility to ensure that the functions invoked are pure, i.e. side-effect free, or that the side-effects are well-behaved. A systematic treatment of side-effecting comprehensions in an object-oriented language has been proposed [11].

## 4 Power and Optimisation

While query notations based on the relational algebra are easily optimised, they are not computationally complete — neither computation nor recursion can be expressed. Conversely, queries expressed as procedures in a programming language can perform arbitrary computation, but are hard to optimise. This section demonstrates the power of comprehension notation and that it can be optimised.

The fundamental reason for comprehensions being both powerful and easily optimised is that their semantics is based on the typed lambda calculus. Computation and recursion are available because comprehensions can easily invoke pure recursive functions, as they share the same semantic basis. Pure recursive functions, i.e. those without side-effects, are computationally complete. To illustrate recursion and computation two well-known bill of material queries are expressed as comprehensions. Optimisation is possible because the lambda calculus semantics permits the proof of equivalences, or transformations, between comprehensions. A suite of list comprehension transformations that emulate all of the major conventional query optimisations has been developed [25] and two of the transformations demonstrated [27]. To illustrate the optimisations two example transformations are given. Even queries entailing computation or recursion can be improved in part using these transformations. The computational parts of such queries are also amenable to program transformation. To illustrate the techniques, both the computational and non-computational parts of Date's bill of material query are improved.

### 4.1 Power

Codd defined a query notation to be relationally *complete*, i.e. adequately expressive, if it is at least as powerful as the relational calculus [9]. The relational calculus and relational algebra have equivalent expressive power. The relational completeness of list comprehensions, augmented by three auxiliary functions, was proved by giving a translation that maps any relational calculus query into an equivalent list comprehension [24]. The equivalent comprehension could contain list concatenation, generalised list intersection, and an emptiness test.

Comprehension notation, combined with recursive functions, has greater power than the relational calculus as it can express both recursion and computation. The functions invoked in comprehensions must be chosen with care. Firstly, because arbitrary recursive functions can be invoked, there is no guarantee that a recursive comprehension query will terminate, unlike relational calculus queries. Secondly, the interaction with side-effecting functions has already been described, and for the remainder of the paper we consider only pure, i.e. non side-effecting, functions.

## Recursion

An example illustrating just recursion is Date's bill-of material query. The problem is stated as follows [10], given a relation:

PARTS (MAIN\_COMPONENT, SUB\_COMPONENT, QUANTITY)

determine the set of all component and sub-component parts of a given main part (to all levels). A solution using comprehensions is a recursive function *explode* with a single argument *main*, the main part:

$$\text{explode main} = [p \mid (m, s, q) \leftarrow \text{parts}; m = \text{main}; p \leftarrow ([s]\% \text{explode } s)]$$

The *explode* function works as follows. Each tuple in the *parts* relation is obtained by  $(m, s, q) \leftarrow \text{parts}$ . If a tuple's main component is the assembly being exploded ( $m = \text{main}$ ), then the parts  $p$  returned are the subassembly itself ( $s$ ) and its sub-components (*explode*  $s$ ). This solution is considerably briefer than the 27-line SQL solution Date presents. It is also arguably clearer.

## Recursion and Computation

An example illustrating computation and recursion is Atkinson and Buneman's bill of material [4]. Their bill is more complex than Date's in that there are two types of parts, *composite* and *base*. Composite parts are assembled from other parts, whereas base parts are not. A base part has a name, a mass, a cost and a list of suppliers. A composite part has a name, a cost increment (assembly cost), a mass increment and a list of the parts required to assemble it, including the quantity required of each sub-part. Assuming that the subsidiary types such as *mass* have already been defined, the type of parts can be specified as the following abstract data type.

$$\begin{aligned} \text{part} ::= & \text{Base name cost mass [suppliers]} \mid \\ & \text{Comp name [(name, qty)] costinc massinc} \end{aligned}$$

The task set is to compute the total cost and total mass of a composite part, and proves impossible in most relational query languages. Two subsidiary functions are required. The first, *lookup*, locates a part given the part-name key. The second, *sumpair*, performs addition on a list of pairs:

$$\text{sumpair abs} = (\text{sum}[a \mid (a, b) \leftarrow \text{abs}], \text{sum}[b \mid (a, b) \leftarrow \text{abs}]).$$

Given these, and assuming that the parts are stored in a structure named *parts*, a simple list comprehension solution is as follows.

$$\begin{aligned} \text{costandmass } p &= \text{cm } (\text{lookup } p \text{ parts}) \\ \text{cm } (\text{Base } p \text{ c m ss}) &= (c, m) \\ \text{cm } (\text{Comp } p \text{ pqs ci mi}) &= \text{sumpair } ((ci, mi) : [(q * c, q * m) \mid (p, q) \leftarrow \text{pqs}; (c, m) \leftarrow [\text{costandmass } p]]) \end{aligned}$$

The solution works in a straightforward manner. The *costandmass* function simply calls a subsidiary function, *cm*, with the part record corresponding to the given part number. The *cm* function computes the cost and mass of a part record. The cost and mass of a base record are simply the cost and mass attributes of the record. The cost and mass of a composite part are the sum of a list of pairs of costs and masses. The first cost and mass pair is the cost increment and the mass increment for this assembly stage,  $(ci, mi)$ . The remainder of the list of cost, mass pairs are the costs and masses of the subcomponents. Each subcomponent is retrieved by  $(p, q) \leftarrow \text{pqs}$ . The cost and mass of each subcomponent is calculated,  $(c, m) \leftarrow [\text{costandmass } p]$ . Finally the costs and masses of each subcomponent are multiplied by the quantity of the subcomponent required,  $(q * c, q * m)$ .

This solution is as short as any of those presented by Atkinson and Buneman. It is considerably shorter than most of the solutions given, for example the 9-line ML solution and the 43-line Pascal solution. The solution is also clearer than many of those given in Atkinson and Buneman's paper.

## 4.2 Optimisation

It is important that a simple specification of a query can be transformed into a version that can be evaluated efficiently. An equivalent list comprehension transformation exists for each major conventional optimisation strategy [25]. There are two classes of optimisation, or improvement, strategies — algebraic and implementation-based [10, 32]. Algebraic improvements are the result of transforming a query into a more efficient form using identities in the relational algebra. Implementation-based improvements are obtained by using information about how the data is stored. This information might include the size of the relations and what indices exist. To give a flavour of comprehension optimisation, one algebraic transformation and one implementation-based transformation are given.

For the purposes of optimisation it is assumed that the order of the tuples in the result of a query is not significant. This is consistent with the relational model, and means that *bag equality* can be used between lists. Two lists are bag equal if they contain the same elements, although possibly in different orders.

### 4.2.1 Selection Promotion

Selection promotion is an algebraic improvement that Ullman identifies as the most important. Performing selection as early as possible reduces the size of the intermediate results by discarding tuples that are not required. Selection promotion is achieved using a comprehension identity that allows the interchange of qualifiers.

**Qualifier Interchange** states that any two qualifiers  $q$  and  $q'$  can be swapped, if they don't refer to variables bound in each other. Using  $\cong$  to denote bag equality, it may be stated

$$\begin{aligned} & [e \mid q; q'] \\ \cong & [e \mid q'; q]. \end{aligned}$$

Qualifier interchange is a generalisation of selection promotion as it allows us to change the order of generation as well as the order of filtration. This generality is also reflected by the fact that qualifier interchange is analogous to several relational algebra identities. These are the identities governing the commuting of products and selections.  $\square$

For example the query

$$[d \mid (a, b) \leftarrow AB; (c, d) \leftarrow CD; a = 99; b = c]$$

can be improved by applying qualifier interchange to ' $(c, d) \leftarrow CD$ ' and ' $a = 99$ ' to yield

$$[d \mid (a, b) \leftarrow AB; a = 99; (c, d) \leftarrow CD; b = c]$$

This is considerably more efficient than the original query. If the size of  $AB$  and  $CD$  is  $n$ , then the time complexity of the original query is  $O(n^2)$ . Usually the number of tuples with  $a$  value 99 is much smaller than  $n$ . If we assume that it is a small constant, i.e. independent of  $n$ , the new query is  $O(n)$ . The transformations have been used to improve comprehension queries in a PS-algol variant and the results obtained reflect the difference in complexity between the two queries [27].

### 4.2.2 Indices

Using an index instead of a sequential search is an example of an implementation-based improvement. A transformation that allows an index to be used in a comprehension query is given below.

**Index introduction.** If  $e'$  is an expression, and there is an index  $jindex_A$  on an attribute  $a_j$  of relation  $A$ , then

$$\begin{aligned} & [e \mid (a_0 \dots a_n) \leftarrow A; a_j = e'] \\ \cong & [e \mid (a_0 \dots a_n) \leftarrow jindex_A e']. \quad \square \end{aligned}$$

Let us assume that the appropriate indices exist in the example above:

$$[d \mid (a, b) \leftarrow AB; a = 99; (c, d) \leftarrow CD; b = c].$$

Index introduction can then be applied to ‘ $(a, b) \leftarrow AB; a = 99$ ’, to obtain

$$[d \mid (a, b) \leftarrow aindex_{AB} 99; (c, d) \leftarrow CD; b = c].$$

A second application gives

$$[d \mid (a, b) \leftarrow aindex_{AB} 99; (c, d) \leftarrow cindex_{CD} b].$$

This is a very efficient form of the query. If we continue to assume that the number of  $AB$  tuples with a value 99 is constant, then only a constant number of  $cindex_{CD}$  lookups need be performed. Each lookup requires  $\log n$  accesses giving a total cost of  $O(\log n)$ .

### 4.3 Improving Computational Queries

Because comprehension notation is both powerful and optimisable the powerful queries, i.e. those entailing computation and recursion, can also be optimised. The powerful queries are improved in two ways. Firstly, the parts of the query that do not entail recursion or computation can be optimised using the transformations described above. Secondly, the computational or recursive parts of the query are amenable to program transformation techniques. The improvement of Atkinson and Buneman’s computational and recursive query is described in [26]. The improvement of both the recursive and non-recursive parts of Date’s bill of material query is given here. Recall the solution from Section 4.1:

$$explode\ main = [p \mid (m, s, q) \leftarrow parts; m = main; p \leftarrow (s : explode\ s)],$$

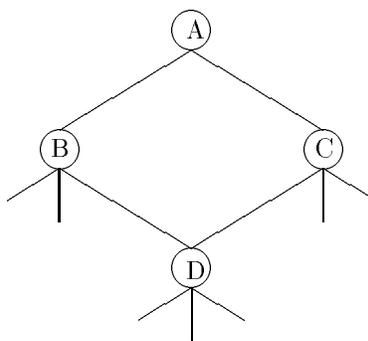
The non-recursive parts of *explode* can be improved by considering the first two qualifiers,  $(m, s, q) \leftarrow parts; m = main$ . These qualifiers cause the entire parts relation to be scanned to locate the immediate sub-components of a main component, *main*. As *explode* is invoked for each component, the parts relation is scanned once for each component in the bill. Hence if the bill is a tree, the size of the parts relation is  $n$ , and the number of nodes in the bill being exploded is  $m$  then the explosion requires  $mn$  block accesses. If there is an index *uses* that lists all of the sub-components of a main component, then index introduction can be applied to obtain

$$explode\ main = [p \mid (m, s, q) \leftarrow uses\ main; p \leftarrow (s : explode\ s)]$$

This is far more efficient as it simply looks up the subcomponents of a part without having to scan the entire relation for them. If we assume that there are nearly as many main components as there are parts, then the index lookup requires  $O(\log n)$  block accesses. As lookup is performed for each node in the bill in turn, the total cost is  $O(m \log n)$  accesses.

The recursive part of *explode* can be improved using *memoising*, a program transformation technique. One source of inefficiency in *explode* arises because a bill of material is a directed acyclic graph (DAG), rather than a tree. As written, *explode* will revisit any subcomponent that is common to two or more components in the bill. In the bill sketched in Figure 4.1, Node D, and its subcomponents, will be visited in the processing of both node B and node C.

Figure 4.1



The redundant processing can be eliminated by memoising *explode*. A memo function is like an ordinary function except that it stores the arguments it is applied to, together with the results computed from them. If a memo function is applied to the same argument again the previously computed value can simply be looked up [18]. Because *explode*'s argument is a large data structure a variant of memo functions, namely lazy memo functions [13], is appropriate. On encountering a node that has already been processed, a memoised instance of *explode* can simply lookup the value already computed and need not reprocess the node.

## 5 Multiple bulk types

### 5.1 Language Complexity

Many language designers believe that a DBPL should support a rich set of data types to enable the application programmer to model the problem domain in a natural way. In a travel agency, for example, the people on a tour might be represented as a set because nobody should be booked on the tour twice, and there is often no order between people. In contrast, the tour destinations are a list because the order of visits is important, and the same destination may be visited twice. Programming languages with orthogonal persistence provide additional support for a rich type system by permitting data of any type to persist.

Queries are not expressed over all types, for example integer values are not interrogated. The data types that are interrogated are those that contain large amounts of data, such as sets, list or trees. Such types are called Bulk Data Types and are characterised by the fact that the size of the value is independent of the size of the type description [5, 17].

Supporting many bulk types can introduce complexity into the application programmer's world. Language constructs must exist to declare, populate and query each bulk type. Let us consider the task of querying the bulk types. In Napier, for instance, a procedure that expresses a query over a tree will be different from, although similar to, a procedure that expresses the same query over a list. An alternative to writing procedures to traverse bulk types is to provide language support for queries over several useful bulk types. Even language supported bulk types may complicate the application programmer's world because there must be a query notation for each bulk type. The DBPL compiler and optimiser may also become complex as they must translate and improve queries over each supported bulk type.

### 5.2 Ringads

The complexity introduced by multiple bulk types can be reduced if a single notation can be used to express queries over several, or all, bulk types. Comprehensions provide just such a uniform, optimisable, notation. Comprehensions can be defined in terms of just three functions, providing the functions obey certain algebraic laws [30]. Collections of the functions, called *iter*, *single* and *zero*, exist for several bulk types including sets, relations, bags, lists, finite maps and some trees. Hence comprehensions can be used to query values of all of these types.

A type constructor, such as *List*, together with *iter* and *single* functions that satisfy three algebraic laws are called a *monad*, by category theorists [16]. A monad augmented with a well-behaved *zero* function is termed a quad and permits the definition of comprehensions with filters. Augmenting a quad with a well-behaved *combine* function permits the construction of bulk values in a uniform manner. A *ringad* is a monad with both a *zero* and a *combine* function that obey the laws given in Appendix A.

Each of the ringad functions has a clear intuitive meaning for bulk types. For concreteness, let us consider the *Set* ringad. The *single* function takes an element and creates a bulk value containing just that element. For sets, *single* corresponds to the singleton operator, e.g.  $single\ 3 = \{3\}$ . The *zero* function takes an element and returns an empty bulk value. For sets, *zero* corresponds to a function that returns the empty set, e.g.  $zero\ 3 = \{\}$ . The expression  $combine\ c\ c'$  merges two collections so that every element in  $c$ , and every element in  $c'$ , has a counterpart in the result. For sets *combine* is simply set union, e.g.  $combine\ \{3,4\}\ \{4,5\} = \{3,4,5\}$ . The expression  $iter\ f\ c$  iterates over  $c$  applying a function  $f$  to every element. The function  $f$  is multi-valued, i.e. it returns a bulk value, and the resulting values are combined together to produce the result of *iter*. For example, if *factors* is a function that returns the set of prime factors of an integer, then

$$\begin{aligned}
\text{iter factors } \{8, 9, 10\} &= \text{combine (factors 8) (combine (factors 9) (factors 10))} \\
&= \text{combine } \{2\} \text{ (combine } \{3\} \{2, 5\}) \\
&= \{2, 3, 5\}.
\end{aligned}$$

Most of the eight laws that the ringad functions must obey also have intuitive meanings for bulk types. For example, we would expect that inserting a value into a bulk type does not change the value inserted. If  $f$  is any observer function, then this intuition is captured by the following ringad law.

$$\text{iter } f \text{ (single } x) = f \ x.$$

Given a ringad, comprehensions can be defined as follows, where  $\Lambda$  denotes the empty qualifier.

$$\begin{aligned}
[E \mid \Lambda] &= \text{single } E \\
[E \mid E'; Q] &= \text{if } E' \text{ then } [E \mid Q] \text{ else zero } E \\
[E \mid V \leftarrow E''; Q] &= \text{iter } (\lambda V. [E \mid Q]) \ E''
\end{aligned}$$

These rules are a uniform compilation scheme for comprehensions over all ringad bulk types. The uniform scheme will simplify the compiler for a DBPL supporting several bulk types. Comprehensions over different kinds of bulk type can be distinguished by labelling them with the type constructor, e.g.  $[x \mid x \leftarrow A; x < 3] \text{Set}$  might be a comprehension generating a set of integers. Comprehension defined using the rules above are called ringad comprehensions.

### 5.3 Ringad Optimisation

It might initially appear that a separate optimiser is required to improve queries over each different bulk type supported by a language. In general the more information known about a type, e.g. identities, the more optimisation possible. Hence an optimiser specific to a particular bulk type will be the most effective. However, some optimisations can be proved using just the ringad laws in conjunction with the lambda calculus. Thus a single optimiser can perform some improvements over any ringad bulk type. More optimisations are possible if more information is supplied in the form of a classification of bulk types. An optimiser would exist for each class of bulk types. As elaborated below, a critical distinction between bulk types, and possibly the only classification necessary, is whether the bulk type is ordered.

Some useful optimisations that can be proved using the ringad laws and the lambda calculus include associativity of combine, promoting projections, eliminating common subexpressions, distributing selection over combine, promoting some selections and combining selections. Examples of the latter two order-preserving transformations are given here. If  $b$  and  $c$  are filters, then the ringad identity

$$[t \mid b; c] = [t \mid b \wedge c]$$

represents the combination of selections [32]. That is, it states that iterating over the bulk type twice checking one predicate each time is the same as iterating over the bulk type once, checking the conjunction of the predicates. Similarly, if  $q$  is a qualifier not binding variables free in  $b$ , then selection promotion is captured by

$$[t \mid q; b] = [t \mid b; q].$$

Several useful optimisations cannot be proved using just the ringad laws and the lambda calculus. The selection promotion transformation above is a restricted form of the qualifier interchange transformation from Section 4.2. Qualifier interchange permits the order of generation to be changed, as well as the promotion of filters, but only under the assumption that the order of elements in the result is immaterial. However, several ringad bulk types are ordered, in particular lists and trees. Several other important optimisations depend on being able to change the order of elements in the result, including converting a product into a join, the use of indices, commuting combine and selecting inner and outer loops for cartesian products. These optimisations are so important that it may be worth having an annotation for comprehensions over ordered bulk types that can be used to indicate when the preservation of order is not essential.

## 5.4 User-defined Bulk Types

In some DBPLs, for example those with orthogonal persistence, an application programmer may define a new bulk type that is appropriate to the problem domain. The application programmer’s task is simplified if comprehensions can be defined over the new bulk type because they provide a ready-made query notation, complete with some optimisation, that is consistent with the query notation used over other bulk types.

Comprehensions can be supported over application-programmer defined ringad bulk types if the programmer obeys a naming convention. The compiler may first optimise all comprehensions using ringad identities. The compiler then translates a comprehension of the form  $[t|q]_M$  into invocations of the ringad functions named *M.iter*, *M.single* etc. Now, an application programmer defining a new ringad bulk type, e.g. *bunch*, must construct a module, also called *bunch*, containing the four ringad functions, called *iter*, *single* etc. The bunch module must be made visible to any program that includes a bunch comprehension. Thus, when a comprehension such as

$$[square\ x \mid x \leftarrow u]Bunch$$

is translated to

$$Bunch.iter\ (\lambda\ x.\ Bunch.single\ (square\ x))\ u,$$

the correct ringad functions will be located. If the programmer indicates which class of bulk types the new type belongs to the appropriate optimiser can be selected.

## 6 Conclusion and Discussion

It has been argued that comprehensions are a desirable construct to be included in a DBPL because they can be smoothly integrated with the language and allow queries to be expressed clearly, concisely and efficiently. Moreover, two advantages of comprehension notation have been demonstrated. The first advantage is that, unlike existing notations, comprehensions combine computational power with ease of optimisation. The second advantage is that comprehensions provide a uniform and extensible notation for expressing and optimising queries over several different bulk types.

There are, however, some limitations to the approach, and many open questions. Not all types that might intuitively be thought of as bulk types are ringads. For example trees with data at the nodes do not form a ringad because it is not clear how to combine two such trees together — not only are two trees required, but also a new data value. Another bulk type that is not a ringad is that of finite maps with destructive combine. A map combine is destructive if one map overrides the other when the two maps being combined contain an entry with identical key, but different non-key, values. It is also not known whether some bulk types form a ringad or not. An important class of these types are graphs, for example the directed acyclic graphs found in the bill of material examples of Section 4.1. It is also not yet clear how best to classify bulk types for optimisation purposes.

Another open question is whether it is possible to iterate over two different bulk types within a single comprehension. For example one might iterate over a *set* of people, and for each person iterate over their *list* of destinations. It may be possible to achieve this by mapping one bulk type into another, but it is important to preserve properties of the mapping to permit the optimisation of such mixed-type comprehensions. Monad morphisms [30] might offer an elegant solution, but may not exist between all of the bulk types under consideration. It is not clear whether a different mapping is required for each pair of bulk types (e.g. *tree\_to\_set*, *list\_to\_set*, *bag\_to\_set*, etc.) or whether a single mapping will suffice for each collection (e.g. *to\_set*). If *Passengers* is a set of people, and *destinations* is a method returning a list, and *to\_set* converts a list into a set then the query above might be written

$$[d \mid p \leftarrow Passengers; d \leftarrow to\_set(p.destinations)]Set$$

## Acknowledgements

I would especially like to thank Daniel Chan for challenging me to present the advantages of comprehensions in a coherent form. Malcolm Atkinson and David Watt also made useful comments on the work. Discussions with Phil Wadler helped in applying his work on Monads to bulk types.

# Appendix A: Monads and Ringads

## A.1 Kleisli Monads

This description of monads is closely based on [31], although the monad functions have been renamed. In the following, functions may be subscripted with their type, e.g.,  $id_\tau$  is the identity function on type  $\tau$ . Function composition is denoted by  $\circ$ , i.e.,  $(f \circ g)x = f(gx)$ .

A Kleisli monad is a triple  $(collection, single, iter)$  consisting of a type constructor  $collection$ , and functions  $single$  and  $iter$ . In category-theoretic terms,  $collection$  is a mapping from objects to objects,  $single_\sigma : \sigma \rightarrow \sigma collection$  is a family of arrows, and for each arrow  $f : \sigma \rightarrow \tau collection$  there is an arrow  $(iter f) : \sigma collection \rightarrow \tau collection$ , satisfying:

$$iter\ single_\sigma = id_\sigma collection \quad (1)$$

$$iter\ f \circ single_\sigma = f \quad (2)$$

$$(iter\ g) \circ (iter\ f) = iter\ ((iter\ g) \circ f) \quad (3)$$

## A.2 Monads with a Zero

A monad can be augmented with a zero function. We define a family of functions  $zero_{\sigma,\tau} : \sigma \rightarrow \tau collection$ , each of which simply ignores its argument and returns an empty collection:

$$zero_{\sigma,\tau} x = empty_\tau collection$$

The laws that a zero function must obey are as follows:

$$zero_{\sigma,\tau} \circ e = zero_{\rho,\tau} \quad (4)$$

$$iter\ zero_{\sigma,\tau} = zero_\sigma collection,\tau \quad (5)$$

$$iter\ f \circ zero_{\rho,\sigma} = zero_{\rho,\tau} \quad (6)$$

for any functions  $e : \rho \rightarrow \sigma$  and  $f : \sigma \rightarrow \tau collection$ .

A monad with a zero, sometimes termed a quad, provides exactly the functions required to support comprehensions.

## A.3 Ringads

A ringad incorporates a  $combine$  function, in addition to the functions already mentioned. The  $combine$  function has  $zero$  as left and right unit, i.e.:

$$combine\ (zero_{\sigma,\tau} x)\ (f x) = f x \quad (7)$$

$$combine\ (f x)\ (zero_{\sigma,\tau} x) = f x \quad (8)$$

for any function  $f : \sigma \rightarrow \tau collection$ .

A ringad consists of a monad  $(collection, single, iter)$  and a pair  $(combine, zero)$ , such that  $zero$  is a unit of  $combine$ ,  $zero$  is the zero of the monad, and  $iter$  distributes rightward through  $combine$ , i.e.:

$$iter\ f\ (combine\ c\ c') = combine\ (iter\ f\ c)\ (iter\ f\ c') \quad (9)$$

for any function  $f : \sigma \rightarrow \tau collection$ .

## References

- [1] Abitboul S. Grumbach S. COL: A Logic-based Language for Complex Objects. Proceedings of the Workshop on Database Programming Languages, Roscoff, France (September 1987), 301-333.
- [2] Albano A. Cardelli L. Orsini R. Galileo: A Strongly Typed Interactive Conceptual Language. ACM Transactions on Database Systems 10,2 (June 1985), 230-260.

- [3] Atkinson M.P. PS-Algol Reference Manual 2nd Ed. University of Glasgow Computing Science PPR Report 12 (1985).
- [4] Atkinson M.P. Buneman O.P. Types and Persistence in Database Programming Languages. ACM Computing Surveys 19,2 (June 1987), 105-190.
- [5] Atkinson M.P. Richard P. Trinder P.W. Bulk Types for Large Scale Programming. Proceedings of Information Systems for the 90's, Kiev, Ukraine (October 1990).
- [6] Bancilhon F. Briggs T. Khosafian S. Valduriez P. FAD, A Powerful and Simple Database Language. Proceedings of the 13th International Conference on Very Large Databases, Brighton, England (September 1987), 97-107.
- [7] Breuer P.T. Applicative Query Languages. Cambridge University Engineering Department (1988).
- [8] Buneman P. Nikhil R. Frankel R. An Implementation Technique for Database Query Languages. ACM Transactions on Database Systems 7,2 (June 1982), 164-187.
- [9] Codd E.F. Relational Completeness of Database Sublanguages. Database Systems: Courant Computer Science Series Vol 6, Englewood Cliffs, New Jersey, Prentice Hall, 1972.
- [10] Date C.J. *An Introduction to Database Systems* 4th Ed. Addison Wesley (1976).
- [11] Ghelli G. Orsini R. Pereira Paz A. Trinder P.W. Design of an Integrated Query and Manipulation Notation for Database Languages. Submitted for publication.
- [12] Hudak P. Wadler P.L. (Eds) Report on the Functional Programming Language Haskell. Draft (April 1989).
- [13] Hughes R.J.M. Lazy Memo Functions, in *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture* Nancy, France, Springer Verlag LNCS 201, (September 1985), 129-146.
- [14] Kato K. Masuda T. Kiyoki Y. A Comprehension-based Database Language and its Distributed Execution. Proceedings of Distributed Computer Systems (1990).
- [15] Lécluse C. Richard, P. Velez F.  $O_2$ , an Object-Oriented Data Model. Proceedings of the ACM-SIGMOD Conference, Chicago, USA (1988).
- [16] Mac Lane S. *Categories for the Working Mathematician*. Springer-Verlag (1971).
- [17] Mathes F. Schmidt J.W. Bulk Types; Built-in or Added-on? Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece (August 1990).
- [18] Michie D. 'Memo' functions and machine learning. Nature 218 (April 1968).
- [19] Morrison R. Brown F. Connor R. Dearle A. The Napier 88 Reference Manual. Universities of Glasgow and St Andrews, PPRR-77-89.
- [20] Nikhil R.S. Semantics of Update in a FDBPL. Proceedings of the Workshop on Database Programming Languages, Roscoff, France (September 1987), 365-383.
- [21] Nikhil R.S. Heytens M.L. List Comprehensions in AGNA, a Parallel Persistent Object System. To appear in the Proceedings of the Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts (August 1991).
- [22] Poulouvasilis A.P. FDL: An Integration of the Functional Data Model and the Functional Computational Model. Proceedings of the 6th British National Conference on Databases (BNCOD 6) (July 1988), 215-236.
- [23] Schmidt J.W. Eckhardt H. Mathes F. DBPL Report. DBPL-memo 111-88, Fachbereich Informatik, Johan Wolfgang Goethe-Universität, Frankfurt, West Germany (1988).

- [24] Trinder P.W. Wadler P.L. List Comprehensions and the Relational Calculus. Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland, (August 1988), 115-123.
- [25] Trinder P.W. Wadler P.L. Improving List Comprehension Database Queries. Proceedings of TENCON'89, Bombay, India (November 1989), 186-192.
- [26] Trinder P.W. A Functional Database. D.Phil Thesis, Oxford University (December 1989).
- [27] Trinder P.W. Chan D.K.C. Harper D.J. Improving Comprehension Queries in PS-algol. Proceedings of the 1990 Glasgow Database Workshop, Glasgow, Scotland (March 1990).
- [28] Turner D.A. Miranda System Manual, Research Software Limited (1987).
- [29] Wadler P.L. List Comprehensions. Chapter 7 of Peyton Jones S.L. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
- [30] Wadler P.L. Comprehending Monads. Proceedings of the ACM Conference on Lisp and Functional Programming, Nice, France (June 1990)
- [31] Wadler, P.L. Notes on Monads and Ringads. Internal document, Computing Science Dept. Glasgow University (September 1990).
- [32] Ullman J.D. *Principles of Database Systems*, Pitman, 1980.