Comprehension Syntax

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong

University of Pennsylvania
Department of Computer and Information Science
200 South 33rd Street , Philadelphia PA 19104-6389
Email: {peter,libkin,suciu,val,limsoon}@saul.cis.upenn.edu

Abstract. The syntax of comprehensions is very close to the syntax of a number of practical database query languages and is, we believe, a better starting point than first-order logic for the development of database languages. We give an informal account of a language based on comprehension syntax that deals uniformly with a variety of collection types; it also includes pattern matching, variant types and function definition. We show, again informally, how comprehension syntax is a natural fragment of structural recursion, a much more powerful programming paradigm for collection types. We also show that a very small "abstract syntax language" can serve as a basis for the implementation and optimization of comprehension syntax.

1 Introduction

Ever since relational databases were first conceived [12], first-order logic, i.e., relational calculus/algebra has been taken as the starting point for the design of relational query languages. It has been an invaluable tool for formulating both semantics and syntax, as well as providing deep insights into expressive power of database languages. However, with the desire to increase the expressive power of query languages and with the need to communicate with non-relational data structures – especially those that are provided by object oriented databases, we must ask whether it is appropriate to continue to stretch this once elegant paradigm of first-order logic as a programming language to cope with these richer domains.

We want to propose an alternative strategy: to look at the operations that are *naturally* associated with the data structures involved, and to use this as a guiding principle for database language design. For example a database relation is a set of records. In this case, our approach is to achieve more generality and flexibility by looking independently at the canonical operations for record types and for set types. An immediate consequence of this approach is that, with the ability to combine set and record construction in an arbitrary fashion, we can build languages for "non-flat" relations, i.e. nested relations or, more generally, complex objects [2]. Such languages are important for several reasons, among them is the ability of object-oriented databases to support objects that can themselves contain sets of objects. Another important reason is that the same principles apply to other collection types such as lists, bags (multisets), arrays, indexed structures and certain kinds of trees.

Our purpose in this paper is not to give a full account of the languages that can be developed by this approach, but to describe a simple language of *comprehensions* that bears close affinities with a number of practical database languages and to show informally how this language is just a restriction of the more general language of *structural recursion* - the language that arises from considering the canonical operations on collection types [8, 7, 9, 32]. The authors are currently engaged in writing a book on the theory and practice of programming languages for collection types, and the material presented here is based upon an introductory chapter of that book. Among the important topics we shall not cover are type-checking and inference, expressive power, optimization and implementation. We shall simply present a short "primer" on comprehension syntax together with an informal rationale for its development. In particular we shall

- show how the language of comprehensions can naturally and uniformly express operations on various collection types including sets, bags and lists;
- show how variant types, function declarations and pattern matching can also be uniformly supported in this language, and
- provide an informal introduction to structural recursion and its relationship to comprehension syntax.

Let us start by looking at a familiar query in SQL [15] that extracts all the pairs of employee names and manager names from the relations Emp of employees and Dept of departments:

```
SELECT  Name, Mgr
FROM    Emp, Dept
WHERE   Emp.D# = Dept.D#
```

A more verbose version of this query can also be written in SQL

```
SELECT  Name = p.Name, Mgr = d.Mgr
FROM    Emp p, Dept d
WHERE   p.D# = d.D#
```

We can put a different interpretation on the syntax of this query. In SQL, the symbols p and d are simply aliases for the relation names Emp and Dept respectively. Instead we take them as *tuple variables* that are bound to successive tuples in the respective relations. Each pair of tuples that satisfies the WHERE ... condition contributes to the output. The idea of having explicit tuple variables is not new; they are used in the (tuple) relational calculus and, more importantly, they are used in certain query languages for object-oriented databases [5, 23, 6]. In fact, the O_2 query language [5] has some interesting connections with what we shall develop. In our syntax this query is written:

$$\begin{aligned}&\{[\mathsf{Name} = p.\mathsf{Name},\ \mathsf{Mgr} = d.\mathsf{Mgr}] \mid\\ &\quad \backslash p \leftarrow \mathsf{Emp},\\ &\quad \backslash d \leftarrow \mathsf{Dept},\\ &\quad p.\mathsf{DNum} = d.\mathsf{DNum}\}\end{aligned}$$

The syntactic form $\{e \mid c_1,\ c_2, \ldots,\ c_n\}$ is a *comprehension*. It is an expression that denotes a collection – in this case a set. It can be read as "the set of all e such that c_1 and c_2 ...and c_n". The term "comprehension" comes from set theory and is used in certain programming languages [30]. The syntax closely resembles the second SQL query, but record construction is explicit: $[\mathsf{Name} = p.\mathsf{Name}, \mathsf{Mgr} = d.\mathsf{Mgr}]$ denotes a record with Name and Mgr fields. There is also a close resemblance between comprehension syntax and relational calculus, but there is an important difference in that we explicitly introduce (or "bind") the variables p and d by marking them with a backslash. Thus $\backslash p \leftarrow \mathsf{Emp}$ is to be thought of as binding p to successive records in the Emp relation.

Variable bindings such as $\backslash d$ are simple examples of *patterns*, which serve both to bind variables and to match structures [22]. For example,

$$\{n \mid [\mathsf{Name} = \backslash n,\ \mathsf{DNum} = 12, \ldots] \leftarrow \mathsf{Emp}\}$$

extracts the names of employees in department 12. The pattern $[\mathsf{Name} = \backslash n,\ \mathsf{DNum} = 12, \ldots]$ matches those tuples in Emp that have a DNum equal to 12. For each of these, the variable n is bound to the Name field. The ellipsis (...) matches the remaining fields of the records in Emp. This query is equivalent to $\{e.\mathsf{Name} \mid \backslash e \leftarrow \mathsf{Emp}, e.\mathsf{DNum} = 12\}$. Our original SQL query can also be expressed using pattern matching:

$$\begin{aligned}&\{[\mathsf{Name} = n, \mathsf{Mgr} = m] \mid\\ &\quad [\mathsf{Name} = \backslash n, \mathsf{DNum} = \backslash d, \ldots] \leftarrow \mathsf{Emp},\\ &\quad [\mathsf{DNum} = d, \mathsf{Mgr} = \backslash m \ldots] \leftarrow \mathsf{Dept}\}\end{aligned}$$

The variable d is introduced in the first pattern and then used in the second. Once a variable is introduced in this way, its scope extends to the end of the comprehension. It can also be used in the "head" of the comprehension. In complicated programs it is essential to distinguish between the binding and use of a variable, which accounts for our need to make bindings explicit. The scoping rules and the meaning of comprehensions can be understood using nested *for*-loops. The comprehension from the previous example can be understood as resulting in the set S obtained as follows:

$$\begin{aligned}&S := \{\ \};\\ &\mathsf{foreach}\ [\mathsf{Name} = \backslash n, \mathsf{DNum} = \backslash d, \ldots]\ \mathsf{in}\ \mathsf{Emp}\\ &\quad \mathsf{foreach}\ [\mathsf{DNum} = d, \mathsf{Mgr} = \backslash m \ldots]\ \mathsf{in}\ \mathsf{Dept}\\ &\qquad S := S \cup \{[\mathsf{Name} = n, \mathsf{Mgr} = m]\};\end{aligned}$$

We should stress that this analogy is to clarify scoping rules; it is *not* the way to implement comprehensions efficiently.

Two further examples show that we can easily deal with "non-flat" relations:

$$\{e.\mathsf{Name}.\mathsf{LastName} \mid \backslash e \leftarrow \mathsf{Emp}\}$$

$$\begin{aligned}&\{[\mathsf{DNum} = d,\ \mathsf{Project} = p] \mid\\ &\quad [\mathsf{DNum} = \backslash d,\ \mathsf{Projects} = \backslash s, \ldots] \leftarrow \mathsf{Emp},\\ &\quad \backslash p \leftarrow s\}\end{aligned}$$

In the first of these, the Name field is assumed to be itself a record. In the second we have assumed that the Emp tuples have a Projects field which is itself a set. The query returns a (flat) relation that connects a department with a project if the department has an employee working on the project. Such queries are common in object-oriented databases. The second example is an example of unnesting in nested relational algebra. We may conversely nest a relation with attributes A and B with

$$\begin{aligned}&\{[A = a, B' = \{b \mid [A = a, B = \backslash b] \leftarrow R\}] \mid\\ &\quad [A = \backslash a, \ldots] \leftarrow R\}\end{aligned}$$

2 Collections

Until now we have tacitly assumed that the collections we have been dealing with are sets, but we cannot get far with this assumption. Consider a simple query involving aggregate functions:

$$average\{x.\mathsf{Salary} \mid \backslash x \leftarrow \mathsf{Emp}\}$$

If the expression $\{x.\mathsf{Salary} \mid \backslash x \leftarrow \mathsf{Emp}\}$ denotes a set, this query may not give the expected result, for if two employees have the same salary, that value will only occur once in the set. However, if we assume that the expression denotes a multiset, or *bag*, then multiple occurrences of the same salary will be allowed and the query should give the desired result. We use the syntax $\{2, 3, 4\}$ to denote a *set* of values ($\{3, 2, 4, 2\}$ denotes the same set). *Bags* (multisets) allow multiple occurrences of the same element, so that the syntax $\{\!|3, 2, 4, 2|\!\}$ represents a bag with two occurrences of 2 and is different from $\{\!|2, 3, 4|\!\}$. Finally, we use the notation $\{\!|\!|3, 2, 4, 2|\!|\!\}$ for sequences of values, i.e. *lists*. For example, $\{\!|\!|2, 3, 4, 2|\!|\!\}$ and $\{\!|\!|3, 2, 4, 2|\!|\!\}$ represent different lists.

The same principles of comprehension syntax make sense for all three collection types, but we shall different kinds of brackets, analogous to the brackets used above. It is possible to use the same brackets for all three types, but this puts an additional burden on type inference, and this is beyond the scope of our paper. The correct computation of the average salary is therefore:

$$average\{\!|x.\mathsf{Salary} \mid \backslash x \leftarrow SetToBag\ \mathsf{Emp}|\!\}$$

SetToBag creates a bag from a set, giving each element a multiplicity of one.

Why is the function *SetToBag* needed here? The form $p \leftarrow s$ inside a comprehension is called a *generator*. Inside a bag comprehension s must be a bag. Similarly all the generators in a set comprehension must use sets, and likewise for lists.

An important observation is that the order in which the generators appear in a comprehension dictates the way in which the elements of the resulting collection are generated. As pointed out earlier, a good analogy is to think of the generators as nested *for*-loops where outermost corresponds to leftmost. This is evident for list comprehensions:

$$\{\!|\!|(x, y) \mid \backslash x \leftarrow \{\!|\!|1, 2|\!|\!\},\ \backslash y \leftarrow \{\!|\!|10, 20, 30|\!|\!\}|\!|\!\}$$

produces the list

$$\{\!|\!|(1, 10), (1, 20), (1, 30), (2, 10), (2, 20), (2, 30)|\!|\!\}$$

Interchanging $\backslash x \leftarrow \{\!|\!|1, 2|\!|\!\}$ and $\backslash y \leftarrow \{\!|\!|10, 20, 30|\!|\!\}$ produces $\{\!|\!|(1, 10), (2, 10), (1, 20), (2, 20), (1, 30), (2, 30)|\!|\!\}$. Had this particular comprehension been a bag comprehension, or a set comprehension, interchanging the two generators would not have changed the resulting collection because of the commutativity laws that hold for bags and sets.

Lists, bags and sets are usually available as distinguished types in object-oriented database systems [23]. In addition, the semantics of SQL involves both sets and bags, though they are not cleanly separated in that language. There are, in fact other kinds of collections: indexed structures, arrays, or-sets and certain kinds of trees for which the syntax of comprehensions is also meaningful, but they are beyond the scope of this paper.

3 Types

Before proceeding further, it is worth reviewing the syntax of types. They have been implicit in our queries, but both our data and our queries have types and it is important that there is a language to describe these types. The BNF for our data types is given by:

$$\begin{array}{l}\tau := bool \mid int \mid string \mid \ldots \mid \{\tau\} \mid \{\!|\tau|\!\} \mid \{\!|\!|\tau|\!|\!\} \mid \\ \quad <l_1 : \tau_1, \ldots, l_n : \tau_n> \mid [l_1 : \tau_1, \ldots, l_n : \tau_n]\end{array}$$

In this, *bool* | *int* | *string* | ... are the (built-in) base types. The other types are all *constructors* – they build new types from existing types. $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ constructs record types from the types $\tau_1, \ldots, \tau_n$. $<l_1 : \tau_1, \ldots, l_n : \tau_n>$ constructs variant types from the types $\tau_1, \ldots, \tau_n$ – more on these in a moment. $\{\tau\}$, $\{\!|\tau|\!\}$, and $\{\!|\!|\tau|\!|\!\}$ respectively construct set, bag, and list types from the type τ.

For example, a type for the Emp relation (though it is certainly not a first normal form relation) could be:

$$\begin{array}{l}\{\,[\mathsf{Name}{:}[\mathsf{FirstName}{:}string, \mathsf{LastName}{:}\ string],\\ \quad \mathsf{DNum}{:}int,\\ \quad \mathsf{Status}{:}<\mathsf{Regular}{:}[\mathsf{Salary}{:}int\ \mathsf{Extension}{:}string],\\ \qquad\qquad \mathsf{Consultant}{:}[\mathsf{Day\_Rate}{:}int,\ \mathsf{Phone}{:}string]>,\\ \quad \mathsf{Projects}{:}\{string\}]\,\}\end{array}$$

The *variant* or "tagged union" type $<\mathsf{Regular} : \ldots, \mathsf{Consultant} : \ldots>$ expresses disjoint possibilities for the Status field of this type. For example $<\mathsf{Regular} = [\mathsf{Salary} = 25000,\ \mathsf{Extension} = 2665]>$ denotes the status of a regular employee, while $<\mathsf{Consultant} = [\mathsf{Day\_Rate} = 750,\ \mathsf{Phone} = 9441212]>$ denotes that of a consultant. Variants are well known in programming languages [16, 34], but are often overlooked in data models, where their absence creates needless fragmentation of the database and confusion over null values. Variants can be conveniently used in pattern matching:

$$\{[\mathsf{Name} = n,\ \mathsf{Phone} = t] \mid [\mathsf{Name} = \backslash n,\ \mathsf{Status} = \texttt{<}\mathsf{Consultant} = [\mathsf{Phone} = \backslash t, \ldots]\texttt{>}, \ldots] \leftarrow \mathsf{Emp}\}$$

$$\{[\mathsf{Name} = n, \mathsf{NoProj} = count(SetToBag(p))] \mid [\mathsf{Name} = \backslash n,\ \mathsf{Status} = \texttt{<}\mathsf{Regular} = \backslash r\texttt{>},\ \mathsf{Projects} = \backslash p, \ldots] \leftarrow \mathsf{Emp}\}$$

The first query finds the names and telephone numbers of all consultants, because the pattern in this comprehension only matches consultants. The second query returns, for each regular employee, the name and number of projects to which that employee is assigned.

4 Declarations

In composing a large query, it is useful to give names to intermediate results or auxiliary functions. Our language allows us to do this through *declarations* with which we can define both variables and functions. Suppose we want to find for each project, all regular employees working on that project. It adds clarity if we first define a function *emps* which takes a project p, and returns the set of the names and salaries of the regular employees working on that project:

$$\begin{array}{l} \mathsf{define}\ emps\ \backslash p \Rightarrow \\ \quad \{[\mathsf{Name} = n, \mathsf{Salary} = s] \mid \\ \qquad [\mathsf{Name} = \backslash n, \\ \qquad\ \mathsf{Status} = \texttt{<}\mathsf{Regular} = [\mathsf{Salary} = \backslash s,\ \ldots]\texttt{>}, \\ \qquad\ \mathsf{Projects} = \backslash ps, \ldots] \leftarrow \mathsf{Emp}, \\ \qquad p \leftarrow ps\} \end{array}$$

The parameters of functions are patterns [22]. In this example the variable p is bound in the simple pattern $\backslash p$, which is the parameter of the function *emps*, but it is a constant in the pattern that occurs in $p \leftarrow ps\}$.

Now we can answer the original query by:

$$\begin{array}{l} \mathsf{define}\ assignments \Rightarrow \\ \quad \{[\mathsf{Project} = p, \mathsf{Empls} = emps(p)] \mid \\ \qquad [\mathsf{Projects} = \backslash ps, \ldots] \leftarrow \mathsf{Emp}, \\ \qquad \backslash p \leftarrow ps\} \end{array}$$

Here *emps* is a function with input p, while *assignments* is a variable. The following *delta* function makes more extensive use of pattern matching. The rule is that if the first pattern matches the input then the value of the corresponding expression is returned; if not, the next pattern is matched, and so on. Thus the *delta* function returns 1 when its input is 0, and 0 when its input is anything else:

$$\begin{array}{lll} \mathsf{define} & delta\ 0 & \Rightarrow\ 1 \\ \mid & delta\ \backslash x & \Rightarrow\ 0 \end{array}$$

We can use arbitrary patterns, including records and variants, in function declarations. Function with patterns are very useful when dealing with variants. Suppose we want to compute, for each project p, the total weekly pay of employees working on that project. We start by defining a function *wage* that computes the weekly income of an employee, as determined by Status:

$$\begin{array}{ll} \mathsf{define} & wage\ \texttt{<}\mathsf{Regular} : [\mathsf{Salary} = \backslash s, \ldots]\texttt{>} \Rightarrow s/52 \\ \mid & wage\ \texttt{<}\mathsf{Consultant} : [\mathsf{Day\_Rate} = \backslash d, \ldots]\texttt{>} \Rightarrow d * 5 \end{array}$$

Using that, we define a function *WageTotal* which computes the total weekly pay of employees attached to the project p:

$$\begin{array}{l} \mathsf{define}\ WageTotal\ \backslash p \Rightarrow sum(\{\!|wage(s) \mid \\ \quad [\mathsf{Status} = \backslash s, \\ \quad\ \mathsf{Projects} = \backslash ps, \ldots] \leftarrow SetToBag(\mathsf{Emp}), \\ \quad p \leftarrow ps|\!\}) \end{array}$$

A simple comprehension then allows us to find the wage expenses for every project.

5 Comprehensions and the Relational Algebra

We have already seen that the general form of a comprehension is $\{e \mid c_1, \ldots, c_n\}$ in which each c_i is either a generator or a condition (a boolean valued expression). A generator has the form $p \leftarrow s$, where p is a pattern and s some expression that denotes a collection. A condition can be a simple condition, like $v\ in\ s$, in which v and s are expressions and which holds iff v is in s, $v = v'$, $v \leq v'$, a conjunction C and C', a disjunction C or C', a negation not C, or a quantifier forall $(p \leftarrow s).C$ or exists $(p \leftarrow s).C$, where p is a pattern, s is an expression denoting a collection, and C is a condition. Note that the quantifier exists $(p \leftarrow s).C$ is just a shorthand for the condition not$(\{[\ ] \mid p \leftarrow s, C\} = \{\ \})$. In this, we could use any constant in place of the empty record $[\ ]$. Finally, forall $(p \leftarrow s).C$ is a shorthand for not(exists $(p \leftarrow s)$.(not C)).

In the following example we compute the set *All_Emps* of employees working at all projects on which John Smith is working, while *Some_Emps* is the set of employees having at least one project in common with John Smith:

define $J\_S \Rightarrow$
  [FirstName = "John", LastName = "Smith"]

define $P\_J\_S \Rightarrow \{p \mid$
  [Name = J_S, Projects = $\backslash ps$] $<-$ Emp,
  $\backslash p <- ps\}$

define $All\_Emps \Rightarrow \{e \mid \backslash e <-$ Emp,
  forall $(\backslash p <- P\_J\_S).p$ in e.Projects$\}$

define $Some\_Emps \Rightarrow \{e \mid \backslash e <-$ Emp,
  exists $(\backslash p <- P\_J\_S).p$ in e.Projects$\}$

All operations from the relational algebra [31] can be expressed using comprehensions:

define $union(\backslash x, \backslash y) \Rightarrow \{v \mid \backslash z <- \{x, y\}, \backslash v <- z\}$

define $difference(\backslash x, \backslash y) \Rightarrow \{v \mid \backslash v <- x, \text{not}(v \; in \; y)\}$

define $product(\backslash x, \backslash y) \Rightarrow$
  $\{$ [A $= u$, B $= u'$, C $= v$, D $= v'$] $\mid$
    [A $= \backslash u$, B $= \backslash u'$] $<- x$,
    [C $= \backslash v$, D $= \backslash v'$] $<- y\}$

define $select_{\mathsf{A=C}}(\backslash x) \Rightarrow \{u \mid \backslash u <- x, u.A = u.C\}$

define $project_{(\mathsf{A,C})}(\backslash x) \Rightarrow$
  $\{$ [A $= u$, C $= v$] $\mid$ [A $= \backslash u$, B $= \backslash v, \ldots$] $<- x\}$

In these definitions we have used $(\backslash x, \backslash y)$ as shorthand for the record pattern [#1 $= \backslash x$, #2 $= \backslash y$]. Thus two-argument functions, such as *union* are, strictly speaking, functions of a two-field record pattern [22].

It is also possible to express the GROUP-BY construct of SQL using nested collections (i.e. by doing the grouping explicitly). E.g., consider some relation R(A, B, C) with attributes A, B and C, of which B is an integer, and consider the following SQL query:

```
SELECT     A, sum(B)
FROM       R
GROUP-BY   C
```

We assume that R satisfies the functional dependence C$\rightarrow$A (else the SQL program is not correct). This query can be expressed with comprehensions as:

define $group\_by\_c(\backslash c) \Rightarrow \{\!| r.\mathsf{B} \mid \backslash r <- \mathsf{R}, r.\mathsf{C} = c |\!\}$

define $response \Rightarrow$
  $\{\!|$ [A $= r$.A, SB $= sum(group\_by\_c(r.\mathsf{C}))$] $\mid \backslash r <- \mathsf{R} |\!\}$

It is then straightforward to add a generic *group_by* operator as a shorthand for this program.

6 Comprehension Language

In this section we put the pieces together and define a functional programming language for collection types, which we call the *comprehension language* (*CL*). A *program* is a sequence of declarations of variables and/or functions:

define $id1 \ldots$
define $id2 \ldots$
  $\vdots$

where id1, id2, ..., are variable or function identifiers. A *variable declaration* has the form:

define $x \Rightarrow e$

where e is some expression, while a *function declaration* has the form:

define f $p1 \Rightarrow e1$
  $\mid$ f $p2 \Rightarrow e2$
  $\vdots$

Here p1, p2, ...are patterns, while e1, e2, ..., are expressions. Recursively defined functions are not allowed within *CL*.

The syntax of *expressions* depends on their type. We consider some built-in operations associated with the base types, and allow numeric expressions such as $e1+e2$, boolean expressions such as $e1$ and $e2$, and a conditional if e then $e1$ else $e2$ where e is a boolean expression and $e1, e2$ have the same type. Some given set of aggregate functions and conversion functions between the three different collection types is assumed. Thus $SetToBag(s)$ is an expression of type $\{\!|\tau|\!\}$, provided that s is some expression of type $\{\tau\}$, and $sum(e)$ is an expression of type *int*, when e is an expression of type $\{\!|int|\!\}$. The more interesting syntactic constructs for expressions are associated with collection types, record and variant types. Thus we can (1) construct a record, by [A1 $= e1, \ldots,$ An $= en$] where $e1, \ldots, en$ are expressions, and (2) extract from a record, by e.A. Also we can (1) construct a collection, e.g. a set $\{e1, \ldots, en\}$, or (2) inspect a collection, using comprehension syntax. We can construct variants using expressions like <A = e> and take them apart with pattern matching.

The *patterns* are of four kinds: (1) a *variable binding* $\backslash x$, (2) a *constant pattern* (which may an arbitrary expression, but recall that expressions do not contain variable bindings), (3) a *record pattern*, possibly with trailing ellipsis meaning that additional fields might be present,

$[\mathsf{A1} = p1, \ldots \mathsf{An} = pn, \ldots]$ (here the trailing "$\ldots$" are part of the syntax), where $p1, \ldots, pn$ are pattern, and (4) a *variant pattern* $\langle \mathsf{A} = p \rangle$, where p is a pattern.

Through examples we have given enough evidence that the comprehension language can express everything SQL can, provided that the same built-in functions are available. A legitimate question to ask is whether its additional features, such as nested collections, variants, and functions, make it too powerful. Is it still possible to compile it and execute it with a decent performance? The answer is surprisingly yes. The language *CL* has some intersting "conservativity" properties. Suppose, first, we consider *CL* with no aggregate functions and no conversion functions, and consider only those functions that have "flat" (i.e. first-normal-form) relations as input and output. Those queries are precisely the queries that are expressible in the (standard) relational algebra. Suppose further that we add some aggregate functions, but still restrict the input and output to have the flat relational types. Queries expressible in this language are exactly those that are expressible in the relational algebra with a GROUP-BY operator and the same aggregate functions — essentially SQL. This has two implications: first relational algebra and SQL both occur as natural fragments of *CL*. Second, on those queries that have flat relational type, we can exploit the same optimization techniques that are used in SQL and the relational algebra. *CL* deals with a much richer variety of data structures and non-flat input and output types, and optimization is an interesting issue; however we know that *CL* queries can be evaluated in PTIME.

As evidence for its potential of clean and efficient implementation, we give in section 8 an abstract syntax language, into which the comprehension language can be translated.

7 Structural Recursion

It can be shown [35] that there are queries such as transitive closure that are not expressible in *CL*. This is one reason for considering an extension of the comprehension language with a new construct called *structural recursion*. On the other hand, these forms of recursions are not *ad hoc* programming constructs, but natural iterators associated with the collection types: part of their very definitions as mathematical objects [8]. Moreover, as we will see, a basic restriction of these iterators has exactly the expressive power of comprehensions.

First, we introduce some further notation: $s1 \cup s2$ denotes the union of two sets, $b1 \uplus b2$ is *bag sum* in which one adds the multiplicity of elements, and $l1 @ l2$ is the result of appending lists $l1$ and $l2$. Also, another related group of operators *inserts* elements into collections: $\mathsf{ins}(x, s) \stackrel{\text{def}}{=} \{x\} \cup s$ for sets, $\mathsf{add}(x, b) \stackrel{\text{def}}{=} \{\!|x|\!\} \uplus b$ for bags, and $\mathsf{cons}(x, l) \stackrel{\text{def}}{=} \{\!\|x\|\!\} @ l$ for lists,

Structural recursion comes in two different flavors, corresponding to two different ways of viewing sets, bags, and lists. First we view a collection, say a bag, as being obtained from the empty bag by repeatedly *adding* its elements, one by one. E.g., the bag $\{\!|x_1, x_2, x_3|\!\}$ is viewed as $\mathsf{add}(x_1, \mathsf{add}(x_2, \mathsf{add}(x_3, \{\!|\,|\!\})))$. This suggests the following form of iteration to compute the sum of the elements of a bag of numbers:

$$\begin{array}{lll} \mathsf{define} & sum(\{\!|\,|\!\}) & \Rightarrow 0 \\ & \mid sum(\mathsf{add}(\backslash x, \backslash s)) & \Rightarrow x + sum(s) \end{array}$$

Even though this uses the syntax of general recursive definitions with patterns of collection type we mean it only as syntactic sugar, as an instance of the following specific template:

$$\begin{array}{lll} \mathsf{define} & h(\{\!|\,|\!\}) & \Rightarrow e \\ & \mid h(\mathsf{add}(\backslash x, \backslash s)) & \Rightarrow i(x, h(s)) \end{array}$$

In this case, we say that the function h is defined on bags *by structural recursion on the insert presentation* (*SRI*), with parameters e and i . In order for the definition to make sense on bags, i must be "commutative", i.e. satisfy the condition $i(x_1, i(x_2, v)) = i(x_2, i(x_1, v))$, because the decomposition of some nonempty bag as $\mathsf{add}(\backslash x, \backslash s)$ is not unique. Similarly, we define *SRI* for sets and lists, replacing $\mathsf{add}(x, s)$ and $\{\!|\,|\!\}$ by their counterparts for lists and sets. In the case of sets, i must be in addition "idempotent", i.e. satisfy $i(x, i(x, v)) = i(x, v)$, while for lists, no conditions have to be imposed on i. In fact, structural recursion over lists has been known under the name *fold* or *reduce* in textbooks of functional programming [1].

SRI is a specific program template. We stress once again that none of the languages considered in this paper allows general recursion, nor do they allow general pattern matching on collection types. In contrast with general recursion, structural recursion always terminates.

In the example of the function *sum*, the constant e is 0, while the function i is $i(x, v) = x + v$, so it is commutative but not idempotent: therefore *sum* is correctly defined on bags, and could be defined on lists as well in the same way, but not on sets. A trick is necessary to compute the sum of elements of some set through structural recursion, namely to compute successive pairs $(s, sum(s))$ using structural recursion. We leave the details as an exercise.

For a more elaborate example, consider the Components relation, of type:

$$\text{Components} : \{[\text{Part}:string, \text{Subpart}:string]\}$$

But Components only gives us the direct subparts of each part. To find the indirect subparts, we apply the following function computing the transitive and reflexive closure, trc:

$$
\begin{array}{l}
\text{define } All\_Names \Rightarrow \\
\quad union(\{p.Part \mid \backslash p \leftarrow Components\}, \\
\qquad \{p.Subpart \mid \backslash p \leftarrow Components\})
\end{array}
$$

$$
\begin{array}{l}
\text{define } compose(s1, s2) \Rightarrow \\
\quad \{[\text{Part} = p, \text{Subpart} = r] \mid \\
\qquad [\text{Part} = \backslash p, \text{Subpart} = \backslash q] \leftarrow s1, \\
\qquad [\text{Part} = q, \text{Subpart} = \backslash r] \leftarrow s2\}
\end{array}
$$

$$
\begin{array}{l}
\text{define } trc(\{\ \}) \Rightarrow \\
\quad \{[\text{Part} = p, \text{SubPart} = p] \mid \backslash p \leftarrow All\_Names\} \\
\mid \ trc(\text{ins}(\backslash x, \backslash s)) \Rightarrow \\
\quad trc(s) \cup compose(trc(s), compose(\{x\}, trc(s)))
\end{array}
$$

For the second form of structural recursion we view a collection, say a bag, as a *sum* of smaller bags, these as sums of even smaller bags, and so on, until singleton bags, or the empty bag is reached. This suggests a different kind of recursion for adding up the elements of a bag of numbers:

$$
\begin{array}{lll}
\text{define} & sum(\{\!|\ |\!\}) & \Rightarrow 0 \\
 & \mid sum(\{\!|\backslash x|\!\}) & \Rightarrow x \\
 & \mid sum(\backslash s1 \oplus \backslash s2) & \Rightarrow sum(s1) + sum(s2)
\end{array}
$$

In a similar fashion, we can define the function *reverse* on lists:

$$
\begin{array}{lll}
\text{define} & reverse(\{\!\|\ \|\!\}) & \Rightarrow \{\!\|\ \|\!\} \\
 & \mid reverse(\{\!\|\backslash x\|\!\}) & \Rightarrow \{\!\|x\|\!\} \\
 & \mid reverse(\backslash s1 @ \backslash s2) & \Rightarrow reverse(s2) @ reverse(s1)
\end{array}
$$

In its general form, the *structural recursion over the union presentation*, SRU, allows us to define a function h, with the following template:

$$
\begin{array}{lll}
\text{define} & h(\{\!|\ |\!\}) & \Rightarrow e \\
 & \mid h(\{\!|\backslash x|\!\}) & \Rightarrow f(\{\!|x|\!\}) \\
 & \mid h(\backslash s1 \oplus \backslash s2) & \Rightarrow u(h(s1), h(s2))
\end{array}
$$

Again, some conditions have to be imposed on e and on the functions u. For lists we have to impose (1) *identity*: $u(v, e) = u(e, v) = v$, and (2) *associativity*: $u(v_1, u(v_2, v_3)) = u(u(v_1, v_2), v_3)$. For bags, we have to impose, in addition, (3) *commutativity*: $u(v_1, v_2) = u(v_2, v_1)$, while for sets, we also have to impose (4) *idempotence*: $u(v, v) = v$.

SRI is naturally associated with *sequential* processing of collections: the elements of some collection s are processed one by one. In contrast, SRU is naturally associated with the *parallel* processing of collections, in a divide and conquer manner: to compute $f(s)$, one divides s into two components $s1 \oplus s2$, computes $f(s1)$ and $f(s2)$ in parallel and independently, then compose the two results.

The structural recursion over the insert presentation SRI is at least as expressive as SRU. Indeed, consider the definition of the function h above using SRU. We can define the same function using SRI:

$$
\begin{array}{lll}
\text{define} & h(\{\!|\ |\!\}) & \Rightarrow e \\
 & \mid h(\text{add}(\backslash x, \backslash s)) & \Rightarrow u(f(x), h(s))
\end{array}
$$

The side conditions imposed on the functions used in both SRI and SRU are annoying. As the functions i or u become more complicated, it becomes impossible for a compiler to check them: in fact, checking them is undecidable [8, 28]. However, there are important special cases of SRU where these equations are automatically satisfied. Namely when $e = \{\!|\ |\!\}, u = \oplus$ (or $e = \{\ \}, u = \cup$ for sets, and $e = \{\!\|\ \|\!\}, u = @$ for lists). Then, the more restrictive form of structural recursion is given by the following template:

$$
\begin{array}{lll}
\text{define} & h(\{\!|\ |\!\}) & \Rightarrow \{\!|\ |\!\} \\
 & \mid h(\{\!|\backslash x|\!\}) & \Rightarrow f(\{\!|x|\!\}) \\
 & \mid h(\backslash s1 \oplus \backslash s2) & \Rightarrow h(s1) \oplus h(s2)
\end{array}
$$

In this restricted form, the structural recursion is always well defined. Since in this definition the only function we are able to chose is f, we abbreviate the function h with $\text{ext}(f)$. Its meaning is $\text{ext}(f)(\{\!|x_1, \ldots, x_n|\!\}) = f(x_1) \oplus \ldots \oplus f(x_n)$ (similarly for sets and lists).

8 An Abstract Syntax Language

We are now going to define a language that stands in the same relation to comprehension syntax as relational calculus does to practical relation query languages. The salient point is that this language is built around the restricted form of structural recursion ext defined in section 7 and that, as we will show below, ext is equivalent to comprehensions (this connection was first observed by Wadler [32]). Namely, in one direction we have:

$$\text{ext}(f)(s) = \{\!|v \mid \backslash x \leftarrow s, \backslash v \leftarrow f(x)|\!\}$$

Conversely, ext can be used to "compile away" comprehension syntax by the following simple rules:

1. $\{e \mid \backslash x \leftarrow S, G\} \stackrel{\text{def}}{=} \mathsf{ext}(f)S$, where f is a function defined by $f(\backslash x) = \{e \mid G\}$

2. $\{e \mid C, G\} \stackrel{\text{def}}{=} \mathsf{if}\ C\ \mathsf{then}\ \{E \mid G\}\ \mathsf{else}\ \{\ \}$, when C is a condition.

In this, G stands for the remaining components (generators and conditions) of the comprehension. By repeated application of these rules we can remove successive components of the comprehension until we are left with the simple comprehension $\{e \mid \}$, which is trivially equivalent to $\{e\}$.

To present this abstract syntax language built around ext we make some simplifications: we will ignore variant types and collection types other than sets. While they are very important as data structures and they can easily be added to this presentation, their addition does not fundamentally affect the language. Hence, in what follows, "object-types" are the types of structures that can be built from the base types b using record and set construction:

$$\tau ::= b \mid [l_1 : \tau_1, \ldots, l_n : \tau_n] \mid \{\tau\}$$

where b stands for the built-in types. The language defines two different syntactic categories: *terms* e, having some type τ as defined above, and *functions* f, having some type $\sigma \rightarrow \tau$, where σ, τ are types as defined above. This constrains our functions to be first order, i.e. they cannot take functions as inputs or return functions as results.

Now to the language. We assume that all variables are tagged with their type; that is x^τ is a variable x whose type is τ. The following typing rules express both the syntax and the properly typed expressions for the language. For example read the rule [REC-I] as "If $e_1 \ldots e_n$ are expressions of the language with respective types $\tau_1 \ldots \tau_n$ then we can form the expression $[l_1 = e_1, \ldots, l_n = e_n]$ whose type is $[l_1 : \tau_1, \ldots, l_n : \tau_n]$."

[VAR] $$\frac{}{x^\tau : \tau}$$

[CONST] $$\frac{}{c : b}$$

[REC-I] $$\frac{e_1 : \tau_1 \quad \ldots \quad e_n : \tau_n}{[l_1 = e_1, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

[REC-E] $$\frac{e : [\ldots l : \tau \ldots]}{e.l : \tau}$$

[SET-I] $$\frac{e_1 : \tau, \ldots, e_n : \tau}{\{e_1, \ldots, e_n\} : \{\tau\}}$$

[SET-E] $$\frac{f : \tau_1 \rightarrow \{\tau_2\}}{\mathsf{ext}(f) : \{\tau\} \rightarrow \{\tau_2\}}$$

[FUN-I] $$\frac{e : \tau_2}{\lambda x^{\tau_1}.e : \tau_1 \rightarrow \tau_2}$$

[FUN-E] $$\frac{f : \tau_1 \rightarrow \tau_2 \quad e : \tau_1}{f(e) : \tau_2}$$

[EQUAL] $$\frac{e_1 : o \quad e_2 : o}{e_1 = e_2 : \{[\,]\}}$$

A few comments on this language:

1. We have used lambda terms (anonymous function definition). Read $\lambda x^\tau.e$ as "that function of x^τ whose value is e". In general e will be an expression involving x^τ.

2. We have given an explicitly typed language. However polymorphic type inference is possible [22, 24, 26, 10], which justifies our presentation of a practical language without type annotations.

3. The boolean values *true* and *false* are represented by $\{[\,]\}$ and $\{\}$ respectively, where $[\,]$ is the empty record. These are the two values of *type* $\{[\,]\}$. This simplifies translation from comprehensions, because a condition C in a comprehension can be replaced by $\backslash x \leftarrow C$ where x is a new variable. Thus conditions are generators.

We claim that *CL* can be effectively translated into this abstract syntax. A full justification of this fact is beyond the scope of this paper. However, we have seen that comprehensions can be translated into expressions involving ext, the remaining translations (e.g. the removal of patterns) is not hard. As an example, union at any type is the function (omitting type annotations):

$$\lambda x.\mathsf{ext}(\lambda x.x)\{x.\#1, x.\#2\}$$

That is, the union operation works on a tuple – a record with two fields labelled #1 and #2 – by placing these fields in a set and applying ext of the identity function to this set.

The abstract syntax language has been the subject of several studies (see the discussion in section 9) and is now well understood. This makes it extremely useful in understanding both the expressive power and optimization strategies for comprehension syntax.

One might ask if there is a variable-free presentation of comprehension syntax – just as the relational algebra serves as a variable-free presentation of relational calculus. In fact there is one, but it is an algebra of functions on complex objects rather than on the objects themselves; and it is an algebra that is well known to mathematicians as a (categorical) *monad* with products and sums [32, 9]. It was this construct that suggested the abstract syntax language to the authors. Category theory has been a useful tool in generalizing mathematical structures, and it is unsurprising that it should be useful in generalizing database structures.

9 Conclusions

Let us try to summarize our results by reversing, more or less, the development of this paper. Structural recursion on sets, bags and lists together with the canonical operations on records and variants provide us with a powerful programming paradigm for database structures. By using a natural, but restricted form of structural recursion we obtain a language that is equivalent to comprehension syntax; in fact it is a language into which one can readily compile comprehension syntax. This language has an associated functional algebra. Comprehension syntax itself can be further restricted by constraining the input and output types to be (flat) relational types (sets of records). This language expresses precisely those queries that are definable in the relational algebra. By adding fixed set of aggregate operations such as sum, count, max to comprehension syntax and again restricting it to those queries whose input and output are flat relations, we obtain a "rational reconstruction" of SQL. Perhaps the most surprising observation is that these well-known database languages are natural fragments of a simple and powerful functional language; and there are many more connections with known database languages that are beyond the scope of this paper.

For further reading on this subject, the idea of using structural recursion for database languages was suggested in [4, 24, 7], and the properties of well-defined programs using structural recursion were examined in [8]. Comprehension syntax and its associated algebra was studied in [32, 33, 9] and its connection with structural recursion and complex-object algebras was studied in [9]. That comprehension syntax at relational types gives us a language equivalent to the relational algebra was shown in [25, 35], even if nesting is used in intermediate results. Another result of this kind [27] shows that by adding a bounded fixed-point construct to comprehension syntax gives us, again at relational types, inflationary datalog, and in [19, 21] it is shown that nesting at intermediate types does not add expressiveness in presence of aggregate functions and certain generic queries. Other results on expressive power are to be found in [19, 20, 21]. Our approach can be used for different collections: languages for or-sets were studied in [17, 13] and bag languages in [18]. [29] shows that transitive closure, which is efficiently expressible using structural recursion, has a necessarily exponential implementation in complex-object algebra [3]. [14] show how to encode related database languages in the simply-typed lambda-calculus. The possibility of using comprehension syntax for arrays is examined in [11]. Connections with parallel complexity classes are studied in [28].

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] S. Abiteboul and P. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, 21(3):9–18, 1990.

[3] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988.

[4] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. A Powerful and Simple Database Language. In *Proceedings of International Conference on Very Large Data Bases*, pages 97–105, 1988.

[5] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-oriented Database System. In *Proceedings of 2nd International Workshop on Database Programming Languages*, pages 122–138. Morgan Kaufmann, 1989.

[6] José A. Blakeley. OQL[C++]: Extending C++ with an Object Query Capability. "Database Challenges in the 1990s," Won Kim (ed.), ACM Press Addison-Wesley, 1994 (to appear).

[7] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991.

[8] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[9] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992.

[10] P. Buneman and A. Ohori. Polymorphism and Type Inference in Database Programming, *ACM Transactions on Database Systems*, to appear.

[11] P. Buneman. The Fast Fourier Transform as a Database Query. Technical Report MS-CIS-93-37/L&C 60, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, March 1993.

[12] E. F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–387, June 1970.

[13] E. Gunter and L. Libkin, A Functional Database Programming Language with Support for Disjunctive Information, AT&T Technical Memo BL0112610-931203-47, 1993.

[14] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 332–343, Montreal, Canada, June 1993.

[15] ISO. *Standard 9075. Information Processing Systems. Database Language SQL*, 1987.

[16] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, Berlin, 2nd edition, 1975.

[17] L. Libkin and L. Wong. Semantic Representations and Query Languages for Orsets. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 37–48, Washington, D. C., May 1993.

[18] L. Libkin and L. Wong. Some Properties of Query Languages for Bags. In *Proceedings of 4th International Workshop on Database Programming Languages, Manhattan, New York*, Springer Verlag, 1993.

[19] L. Libkin and L. Wong. Aggregate Functions, Conservative Extension, and Linear Orders. In *Proceedings of 4th International Workshop on Database Programming Languages, Manhattan, New York*, Springer Verlag, 1993.

[20] L. Libkin and L. Wong. A Bounded Degree Property and Finite-cofiniteness of Graph Queries. Technical Report MS-CIS-93-95/L&C 75, University of Pennsylvania, December 1993.

[21] L. Libkin and L. Wong. Conservativity of Nested Relational Calculi with Internal Generic Functions. *Information Processing Letters*, 1994, to appear.

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

[23] Object Design Inc., Burlington, MA. *Object Store Reference Manual*, 1991.

[24] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli, a Polymorphic Language with Static Type Inference. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[25] J. Paredaens and D. Van Gucht. Converting Nested Relational Algebra Expressions into Flat Algebra Expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[26] D. Remy. Typechecking Records and Variants in a Natural Extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[27] D. Suciu. Fixpoints and Bounded Fixpoints for Complex Objects. In *Proceedings of 4th International Workshop on Database Programming Languages, Manhattan, New York*, 1993.

[28] D. Suciu and V. Breazu-Tannen. A Query Language for the Class NC, August 1993. Manuscript available from `suciu@saul.cis.upenn.edu`.

[29] D. Suciu and J. Paredaens. Any Algorithm in the Complex Object Algebra Needs Exponential Space to Compute Transitive Closure, December 1993. Manuscript available from `suciu@saul.cis.upenn.edu`.

[30] D. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In J. P. Jouannaud, editor, *LNCS 201: Proceedings of Conference on Functional Programming Languages and Computer Architecture, Nancy, 1985*, pages 1–16. Springer-Verlag, 1985.

[31] J.D. Ullman. *Principle of Database Systems.* Pitman, 2nd edition, 1982.

[32] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[33] D.A. Watt and P. Trinder. Towards a Theory of Bulk Types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

[34] A. Wijngaarden. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 5:1–236, 1975.

[35] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993.