DRAFT
List Comprehensions and the Relational Calculus

Phil Trinder Phil Wadler

September 13, 1999

1 Introduction

Some functional languages support list comprehensions in which elements of lists can be selected and combined. The author found list comprehensions a very natural and concise means of expressing queries against a database. In both efficiency and simplicity the notation is comparable with database query languages.

The relational calculus is a well known and widely used formalism underlying some database query languages. Queries in the calculus are specified as the set of tuples satisfying a predicate. For example the set of employees whose salaries exceed £12000. There is a close similarity between relational calculus queries and their corresponding list comprehensions. In this paper it is shown that any (safe) relational calculus query can be expressed as a list comprehension.

This is done in two steps. Firstly a denotational semantics for queries with a predicate in a restricted form is given. Secondly an algorithm for converting any relational predicate into the restricted form is presented. The semantics gives the meaning of a query in terms of a set of tuples. An interpreter has

been built which is closely based on the semantics, and in which the set comprehensions become list comprehensions by a well-known homomorphism. Effectively the meaning of a query is expressed as a list comprehension and the semantics represent a translation scheme from relational calculus queries to list comprehensions.

The work is significant because it demonstrates that languages with list comprehensions directly support the relational calculus, a very high level notation for expressing queries. This makes queries simple and short and the clean integration of queries contrasts markedly with the awkward interfaces between many existing database query languages and their procedural host languages. Across this interface easy access is provided to the computational power and recursion of the functional language making it easy to do some things which are hard in the relational calculus. Queries expressed as list comprehensions can also coexist with update transactions. The semantics of the relational queries can provide operational insights for the implementor. Finally, and slightly tangentially, a denotational semantics for list comprehensions has emerged, largely due to P.L. Wadler.

The remainder of the paper is structured as follows. Section 2 introduces list comprehensions. Section 3 gives a brief description of the relational calculus. Section 4 describes the generative form for queries. Section 5 presents the semantics and section 6 concludes.

2 List Comprehensions as Database Queries

For brevity list comprehensions are not formally described, merely illustrated by example. They are a language construct based on set comprehensions and are found in some functional languages[9, 12, 10]. A set comprehension describing the set of squares of all the odd numbers in a set A can be written

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

and has a corresponding list comprehension

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

Operationally the list comprehension can be interpreted as "for each x in A, if x is odd, then put the square of x in the result list". The expressions to the right of the vertical bar are either generators or filters. Generators introduce one or more new variables and indicate the range of values they may take on, e.g. $x \leftarrow A$. Filters specify the conditions the variable must satisfy to be included in the result, e.g. $odd\ x$.

Expressing queries using list comprehensions is illustrated by two example queries which Ullman[11] uses to compare different database query languages. The underlying database is not described except to remark that members,orders and suppliers are relations. The queries use the functional data model.

"Print the names of the members with negative balances"

$$[mname\ m | m \leftarrow members; mbal < 0]$$

"Print the supplier names, items and prices of all suppliers that supply at least one item ordered by Brooks"

$$[< sname, sitem, sprice > s | o \leftarrow orders; oname\ o = \text{"}Brooks\text{"}; \\ s \leftarrow suppliers; sitem\ s = oitem\ o]$$

Here $< \ldots >$ is an operator which takes an m-tuple of functions f_i of type $\alpha \rightarrow \beta$ and returns a function of type $\alpha \rightarrow (\beta_1 X \ldots \beta_n)$. The queries expressed using list comprehensions are comparable in simplicity with those expressed in conventional query languages. Efficiency techniques such as secondary indices and filter promotion can also be used.

3 Description of the Relational Calculus

The relational calculus is a formalism for describing database queries which was first suggested by Codd [3] and underlies a number of query languages [2, 7, 13, 6]. Essentially queries describe a desired set of tuples by specifying a predicate or formula the tuples must satisfy. This represents a declarative

specification of the query and, unlike the relational algebra, no order needs to be given for the computation of the tuples satisfying the formula.

Much of the following description is paraphrased from Ullman [11], and differs slightly from the calculus described in Date [4]. There are two flavours of the calculus, tuple relational and domain relational. These differ in the type of the variables used in the formulae. In the tuple relational calculus variables represent whole tuples, and in the domain relational calculus variables represent components of tuples. The flavours are equivalent as any query expressed in the tuple relational calculus can be converted to a query in the domain relational calculus, and vice versa. The tuple relational calculus is given semantics and is described below.

Queries in the calculus are of the form $\{t|F(t)\}$, where t is a *tuple variable*, and F is a formula built from atoms and operators. Atoms are of three types.

- $s \in R$, where R is a relation name and s is a tuple variable. This stands for the assertion that s is a tuple in relation R.
- $s[i]opu[j]$, where s and u are tuple variables, i and j are integers and op is an arithmetic comparison operator ($<, =$, etc). This asserts that the ith component of s stands in relation op to the jth component of u. For example $s[1] < u[2]$ means that the first component of s is less than the second component of u.
- $s[i]opa$, or, $aops[i]$, where a is a constant. These assert that the ith component of s stands in relation op to the constant a. For example $s[1] = 3$ means that the value of the first component of s is 3.

If F and F' are formulae, then a formula may be one of the following.

- An atom.
- Formulae combined using logical operators, $F \wedge F', F \vee F'$ or$\neg F$.
- A quantified expression, $\exists s.(F)$ or $\forall s.(F)$.

Parenthesis may be used as needed.

3.1 Examples

Union: $\{t|t \in R \vee t \in S\}$

Difference: $\{t|t \in R \wedge \neg(t \in S)\}$

Cartesian Product

$$\{t^{r+s}|\exists u.\exists v.(u \in R \wedge v \in S \wedge t[1] = u[1] \wedge ... \wedge t[r] = u[r] \wedge \\ t[r+1] = v[1] \wedge ... \wedge t[r+s] = v[s])\}$$

The superscript over the tuple variable indicates the arity of the tuples it represents.

The projection $\Pi_{i_1,i2,...ik}R$ is expressed by:

$\{t^k|\exists u.(u \in R \wedge t[1] = u[i1] \wedge ...t[k] = u[ik])\}$

The selection of tuples of a relation R satisfying a predicate F, or $\sigma_F R$, is expressed by

$\{t|t \in R \wedge F'\}$

where F' is the formula F with each operand i denoting the ith component of R, replaced by $t[i]$.

3.2 Safety

In order to disallow queries with infinite answers such as

$\{t|\neg(t \in R)\}$,

safety conditions are defined. Informally a query, $\{t|F(t)\}$, is safe if it can be demonstrated that each component of any t that satisfies F is a member of $Dom(F)$, which is defined as the set of symbols that either occur explicitly in F, or are components of some tuple in some relation R mentioned in F. For example if $F(t)$ is $t[1] = `a' \vee t \in R$, where $R$ is a binary relation then $Dom(F) = \{`a'\} \cup \Pi_1 R \cup \Pi_2 R$.

More precisely, Ullman defines a query to be safe if:

- whenever t satisfies F, each component of t is a member of $Dom(F)$.
- For each subexpression of F of the form $\exists u.F(u)$, if u satisfies F then each component of u is a member of $Dom(F)$.
- For each subexpression of F of the form $\forall u.F(u)$, if any component of u is not in $Dom(F)$, then u satisfies F.

4 Generativity

The semantics is given for queries with generative formulae. This section describes the generative form and gives an algorithm for translating any relational formula into it. By restricting the syntax of the formulae the semantics is simplified. The semantics assumes a restriction present in Dates' formulation of the calculus but not in Ullmans'. This is that a tuple variable introduced by a quantifier must be drawn from a single relation. The syntax for quantifiers becomes:

$\exists u : R.F(u)$, and $\forall u : R.(F(u))$.

Date[4] notes that introduced variables almost always range over a single relation, in fact this author has not seen a query where this is not the case. If, however, we have a query in which a quantified variable is drawn from some combination of relations it can be expressed in the following way. It is assumed that the query language built using these semantics allows us to name the relations produced by queries. We write a query which constructs the desired combination and name it, N, say. The quantified variable is then asserted to be drawn from N. An example semantics for an operation which binds a relation name to the result of a query is given in 5.2, but the semantics of the remainder of the database manipulating operations is not described.

Note that formulae in this form satisfy safety conditions 2 and 3 directly. Condition 2 always hold because all of the components of u are members of projections of R, which are, in turn part of $Dom(F)$. Condition 3 always

holds because *all* of the components of u are members of projections of R and hence in $Dom(F)$.

4.1 Generative Form

Many formulae are naturally expressed in generative form, for instance all of the examples above. The definition of generative formulae depends on a distiction between atoms; atoms are either generators or filters. Generators are

- assertions that a tuple varible is drawn from a relation.
- assertions that a component of a tuple is equal to some value - either a constant or a component of another tuple variable.

Essentially generators indicate the range of a tuple variable i.e. its possible values. All other atoms are filters and are used to select those tuples which have the desired properties. A generative formula has three properties

- It is in prenex form. This is a well-known normal form[5] in which the quantifiers occur on the left of the expression, with the remainder of the formula quantifier free (i.e. a proposition). This resolves variable scoping issues, and Date[4] recommends it as a natural way of expressing queries.
- The quantifier free part of the formula is in disjunctive normal form, another well known form [5].
- All generators in each conjunct occur before, i.e. to the left of, all filters.

Essentially the generative property ensures that the range of each tuple variable is known before the desired values are filtered out. Note that safety and generativity are independent, an unsafe formula may be generative, and vice

versa. However, any safe formula can be transformed into a safe and generative formula in three steps described below. Each of the steps can always be performed, references are given to the non trivial proofs. The transformation is illustrated with a running example based on the following formula.

$(t[2] = 3 \vee t[2] = 4) \wedge \exists u : R.(u[1] < 6 \wedge t[1] = u[1])$.

Manipulate the formula into prenex form; the example formula becomes

$\exists u : R.((t[2] = 3 \vee t[2] = 4) \wedge (u[1] < 6 \wedge t[1] = u[1]))$

Manipulate the quantifier free formula into disjunctive normal form. The example formula becomes

$$\begin{aligned}\exists u : R.((&t[2] = 3 \wedge u[1] < 6 \wedge t[1] = u[1]) \vee \\ &(t[2] = 4 \wedge u[1] < 6 \wedge t[1] = u[1]))\end{aligned}$$

Using the commutativity of conjunction the generators can always be manipulated to occur in the leftmost positions of each conjunct. The example formula becomes

$$\begin{aligned}\exists u : R.((&t[2] = 3 \wedge t[1] = u[1] \wedge u[1] < 6) \vee \\ &(t[2] = 4 \wedge t[1] = u[1] \wedge u[1] < 6))\end{aligned}$$

which is generative.

4.2 Generative Syntax

The following abstract syntax guarantees that a formula is in generative form. It could be used in a compiler or interpreter to validate the generative property. Safety would have to be validated seperately as it is not guaranteed by the syntax. the query is safe. It is slightly too restrictive in that it forces all generators to precede all filters. This disallows filter promotion, a program transformation which increases the efficiency of comprehensions by doing the filtering as soon as possible, often before some of the generation. Filter

promotion is analagous to the database technique of distributing selection through join.

Syntactic Categories

Query	Relational Calculus Query	Q
Prenexp	Formula in Prenex Form	P
Disj	Formula in disjunctive normal form	D
Conj	Conjuct part of formula	C
Exp	Filters	E
Atom	Relational calculus atom	A
Ops	Comparison Operators	op
TIde	Tuple Identifiers	T
RIde	Relation Identifiers	R
Nmls	Numerals	N
Bools	Booleans	B

$Q = \{T^N | P\}$

$P = \exists T : R.P | \forall T : R.P | D$

$D = D \vee D | C$

$C = C \wedge C | C \wedge E | T \in R | A = A$

$E = E \wedge E | \neg E | AopA | T \notin R$

$A = N | B | T[N]$

$op = < | > | \leq | \geq | \neq$

5 Semantics

The semantics assumes that the query to be described is safe, has been transformed into generative form and is fully bracketed. The semantics is straight

forward, except to note that an unbound component of a tuple variable gives the name of the variable and the index of the component. This is so that, when the unbound component is asserted to be equal to a value, the environment can be extended to reflect this.

5.1 Syntactic Categories

Query	Relational Calculus Query	Q
Exp	Relational Formula	E
Atom	Relational Calculus atom	A
Ops	Comparison Operators	ω
TIde	Tuple Identifiers	T
RIde	Relation Identifiers	R
Nmls	Numerals	N
Bools	Booleans	B

5.2 Abstract Syntax

$Q = \{T^N | E\}$

$E = E \wedge E | E \vee E | \neg E | T \in R | A \omega A | \exists T : R.E | \forall T : R.E$

$A = N | B | T[N]$

$op = < \mid > \mid \leq \mid \geq \mid \neq \mid =$

5.3 Semantic Domains

$Tuple = List\ Val$

$Val = Num + Bool + Unb$

$Unb = Tide\ X\ Num$

$Dbase = RIde \to Set\ Tuple$

$Env = TIde \to Tuple$

5.4 Semantic Functions

5.4.1 $\eta : Nmls \to Val$

Not specified

5.4.2 $\beta : Bools \to Val$

Not specified

5.4.3 $\mathcal{Q} : Query \to Dbase \to SetTuple$

$$\mathcal{Q}[\![\{T^N|E\}]\!]\ \delta = \{\rho[\![T]\!]|\ \rho \in \mathcal{E}[\![E]\!]\ \delta\ \{T \to unbtuple\}\}$$
where
$unbtuple = null\ [\![T]\!]\ \eta[\![N]\!]$

5.4.4 $\mathcal{E} : Exp \to Dbase \to Env \to Set\ Env$

$$\mathcal{E}[\![E_0 \wedge E_1]\!]\ \delta\ \rho = \{\rho_1|\rho_0 \in (\mathcal{E}[\![E_0[\![\delta\rho) \wedge \rho_1 \in (\mathcal{E}[\![E_1]\!]\delta\rho_0)\}$$

$$\mathcal{E}[\![E_0 \vee E_1]\!]\ \delta\ \rho = \{\rho_0|\rho_0 \in (\mathcal{E}[\![E_0[\![\delta\rho) \vee \rho_0 \in (\mathcal{E}[\![E_1]\!]\delta\rho)\}$$

$$\mathcal{E}[\![\neg E]\!]\ \delta\ \rho = filter\ (\mathcal{E}[\![E[\![\delta\rho) = \phi\ \rho$$

$$\mathcal{E}[\![T \in R]\!]\ \delta\ \rho = (unbtuple?\ \rho[\![T]\!] \to \{\rho \oplus \{T \mapsto v\}|\ v \in \delta[\![R]\!]\};\ filter\ (\rho[\![T]\!] \in \delta[\![R]\!])\ \rho)$$

$\mathcal{E}[\![A_0\ \omega\ A_1]\!]\ \delta\ \rho = \Theta\ [\![\omega]\!]\ (\mathcal{A}[\![A_0]\!]\rho)\ (\mathcal{A}[\![A_1]\!]\rho)\ \rho$

$\mathcal{E}[\![\exists T : R.E]\!]\ \delta\ \rho = \{\rho_1 | v \in \delta[\![R]\!] \wedge \rho_1 \in (\mathcal{E}[\![E]\!]\ \delta\ \rho \oplus \{T \mapsto v\})\}$

$\mathcal{E}[\![\forall T : R.E]\!]\ \delta\ \rho = filter\ (all\ (\lambda v.(\mathcal{E}[\![E]\!]\ \delta\ \rho \oplus \{T \mapsto v\}) \neq \phi)\ \delta[\![R]\!])\ \rho$

5.4.5 $\mathcal{A} : Atom \to Env \to Val$

$\mathcal{A}[\![N]\!] = \eta[\![N]\!]$

$\mathcal{A}[\![B]\!] = \beta[\![B]\!]$

$\mathcal{A}[\![T[N]]\!] = v!i$
where
$v = \rho[\![T]\!]$
$i = \eta[\![N]\!]$

5.4.6 $\Theta : Ops \to Val \to Val \to Env \to Set\ Env$

$\Theta[\![<]\!]\ a\ b\ \rho = filter\ (a < b)\ \rho$

Similarly for $>, \leq, \geq, \neq$.

$\Theta[\![=]\!]\ a\ b\ \rho = (unb?\ a \to \{\rho \oplus \{T_a \mapsto (ins\ b\ i_a\ v_a)\}\};$
$(unb?\ b \to \{\rho \oplus \{T_b \mapsto (ins\ a\ i_b\ v_b)\}\};$
$filter\ (a = b)\ \rho))$
where
$T_a = fst\ a$ and $T_b = fst\ b$
$v_a = \rho[\![T_a]\!]$ and $v_b = \rho[\![T_b]\!]$
$i_a = snd\ a$ and $i_b = snd\ b$

5.5 Auxilliary Functions

5.5.1 $null : TIde \to Num \to Tuple$

$null\ [\![T]\!]\ u = null'\ [\![T]\!]\ 1\ u$

$$null'\ [\![T]\!]\ l\ u = \begin{cases} (unbound, T, l) :: null'\ [\![T]\!]\ l\ u, l \leq u \\ [], otherwise \end{cases}$$

5.5.2 $unbtuple? : Tuple \to Bool$

$unbtuple?\ (Unbound, t, n) :: es = True$

$unbtuple?\ (e :: es) = unbtuplees, otherwise$

$unbtuple?\ [] = False$

5.5.3 $all : (Tuple \to Bool) \to Set\ Tuple \to Bool$

$all\ F\ s = \forall x \in s.f\ x$

5.5.4 $filter : Bool \to Env \to Set\ Env$

$filter\ p\ \rho = (p \to \{\rho\}; \phi)$

5.5.5 $ins : Tuple \to Int \to Val \to Tuple$

$$ins\ (x :: xs)\ n\ x' = \begin{cases} x' :: xs, n = 1 \\ x :: ins\ xs\ (n-1)\ x', otherwise \end{cases}$$

6 Conclusions

The most important result described in this paper is that database queries can be simply and concisely expressed using list comprehensions. While the author was lead to this belief by practice, theoretical evidence is provided by showing that relational calculus queries, a high level, declarative specification of queries, can be directly represented as list comprehensions.

The computational power and recursion provided by functional languages can be easily accessed from within list comprehensions. This means that computation, such as sub-totalling, which is hard to do in the relational calculus can be performed. Further, for inherently recursive data manipulation, e.g. Bill of Material processing, the recursion provided in such languages is appropriate.

Queries described using list comprehensions can be expressed as functions from the database to some output and hence are simply transactions against the database. This means that they fit well with the transaction processor already described[1] and queries can coexist with updates.

To the best of the authors knowledge this is the first denotational semantics for the relational calculus. The relational calculus does not really need a semantics because it is sufficiently closely based on set theory. To an extent the semantics is simply a device for expressing the translation of relational calculus queries into list comprehensions in a familiar way. However, while it is desirable for the user to state declaratively which tuples she wishes to see, this specification gives an implementor little indication as to how the desired tuples can be retrieved. It is hoped that some operational insights can be gained by following the reduction of the semantic equations :- by 'cranking the handle', in the vernacular.

Finally, a denotational semantics for list comprehensions, largely attributable to P.L. Wadler has emerged. Previously the meaning of a list comprehension had been described by a set of reduction rules, or by a set of translation rules to the lambda calculus[8].

Finally, thanks are due to Gabreselassie Baraki and John Launchbury for

fruitful discussions and advice which was both sound and complete.

References

[1] Argo et al. Implementing Functional Databases. Proc of the Workshop on Databse Programming Languages, September 1987.

[2] Chang C.L. DEDUCE - a Deductive Query Language for Relational Databases. Pattern Recognition an Artificial Intelligence, Chen C.H. (ed), New York Academic Press, 1976.

[3] Codd E.F. Relational Completeness of Database Sublanguages. Database Systems: Courant Computer Science Series Vol 6, Englewood Cliffs, New Jersey, Prentice Hall, 1972.

[4] Date C.J. An Introduction to Database Systems (3rd Ed). Addison Wesley, 1976.

[5] Hamilton A.G. Logic for Mathematicians. Cambridge University Press, 1978.

[6] Held G.D., Stonebraker M.R., Wong E. Ingress a Relational Database System. Proc. NCC 44, May 1975.

[7] Lacroix M., Wodon P. A Comprehensive Formal Query Language for Relational Databases. RAIRO Informatique/CS Vol 11, No 2, 1977.

[8] Peyton Jones S.L. The Implementation of Functional Programming Languages. Prentice Hall, 1987.

[9] Turner D.A. Miranda System Manual, Research Software Limited, 1987.

[10] Turner D.A. Recursion Equations as a Programming Language. In Functional Programming and its Applications. Darlington J. et al (eds). Cambridge University Press, 1982.

[11] Ullman J.D. Principles of Database Systems, Pitman, 1980.

[12] Wadler P.L. An Introduction to Orwell. Oxford University Handbook, December 1985.

[13] Zloof M.M. Query By Example. Proc. NCC 44, May 1975.