Algebraic Query Optimisation for Database Programming Languages

Alexandra Poulovassilis
Dept. of Computer Science
King's College London
Strand, London WC2R 2LS
alex@dcs.kcl.ac.uk

Carol Small
Dept. of Computer Science
Birkbeck College
Malet St., London WC1E 7HX
carol@dcs.bbk.ac.uk

Abstract

A major challenge still facing the designers and implementors of database programming languages (DBPLs) is that of query optimisation. We investigate algebraic query optimisation for DBPLs in the context of a purely declarative functional language that supports sets as first-class objects. Since the language is computationally complete issues such as non-termination of expressions and construction of infinite data structures can be investigated, whilst its declarative nature allows the issue of side effects to be avoided and a rich set of equivalences to be developed. The language has a well-defined semantics which permits us to reason formally about the properties of expressions, such as their equivalence with other expressions and their termination. The support of a set bulk data type enables much prior work on the optimisation of relational languages to be utilised.

In the paper we first give the syntax of an archetypal DBPL and briefly discuss its semantics. We then define a small but powerful algebra of operators over the set data type, provide some key equivalences for expressions in these operators, and list transformation principles for optimising expressions. Along the way, we identify some caveats to well-known equivalences for non-deductive database languages. We next extend our language with two higher level constructs commonly found in functional DBPLs: set comprehensions and functions with known inverses. Some key equivalences for these constructs are provided, as are transformation principles for expressions in them. Finally, we investigate extending our equivalences for the set operators to the analogous operators over bags. Although developed in the context of a functional language, our findings are directly applicable to other DBPLs of similar expressiveness.

Categories and Subject Descriptors: H.2.3[Database Management]: Languages - query languages, data manipulation languages. I.1.3[Algebraic Manipulation]: Languages and systems - nonprocedural languages.

Additional Key Words and Phrases : query optimisation, functional languages, database programming languages.

1 Introduction

Database programming languages (DBPLs) incorporate into a single language, with a single semantics, all of the features normally expected of both a data manipulation language (DML) and a programming language. For example, DBPLs have one computational model, one type system, and bulk data types with associated access mechanisms. A major challenge still facing DBPLs is that of query optimisation. There are several reasons for limited progress in this area:

(i) The possibility of *side-effects* restricts the set of equivalences that hold.

(ii) Some bulk data types support few equivalences. For example, lists only readily support the common relational optimisations if the concept of *bag equality* is used (Trinder 1989) i.e. the order of the elements in lists is ignored.

(iii) Since DBPLs are computationally complete, the *termination* properties of expressions must be taken into account when investigating equivalences. For example, if the boolean-valued function f_1 does not terminate for some arguments whilst the boolean-valued function f_2 returns $False$ for all arguments, then the 'equivalence' $\sigma_{f_1}(\sigma_{f_2}(s)) = \sigma_{f_2}(\sigma_{f_1}(s))$ does not hold since evaluation of the LHS always terminates for finite s (returning $\{\}$) whereas evaluation of the RHS may not terminate.

(iv) DBPLs can manipulate *infinite data structures* and hence some bulk data operations cannot be implemented using established methods. For example, if A and B are infinite sets then a nested loop method cannot be used to generate $A \times B$ (for otherwise all tuples of the resulting product would have the same first coordinate).

(v) DBPLs typically support *user-defined data types*, and hence require mechanisms to prove and utilise equivalences over these data types too.

(vi) The computational completeness of DBPLs means that a greater variety of optimisations can be applied than for typical DMLs - ranging from peep-hole optimisations to algebraic optimisation and program transformation - and integrating these is an open problem.

In this paper we investigate the optimisation of DBPLs by addressing some of the above issues. We undertake our investigation in the context of a purely declarative functional language. This is a particularly natural computational paradigm in which to investigate query optimisation in DBPLs since database algebras are functional in nature. It also gives us a computationally complete formalism that can exhibit non-termination of expressions (point (iii) above) and that can result in infinite data structures (point (iv) above), whilst avoiding the issue of side-effects (point (i) above). The language supports a set bulk data type, enabling us to utilise much prior work on the optimisation of relational languages, including Datalog (point (ii) above). The language has a well-defined semantics which permits us to reason formally about the properties of expressions, including those of user-defined data types, such as their equivalence with other expressions and their termination properties (point (v) above). Finally, with respect to point (vi) above, this paper addresses algebraic optimisations which we envisage as being complementary to, and used in conjunction with, existing peep-hole optimisations (Augustsson 1984, Johnsson 1984) and program transformation techniques (Burstall and Darlington 1977, Freytag and Goodman 1989).

The structure of the paper is as follows. In Section 2 we give the syntax of our language and discuss its semantics and its provision for built-in and user-defined functions. In Section 3 we define a small but powerful algebra of operators over the set data type, provide some key equivalences for expressions in these operators, and list transformation principles for optimising expressions. Along the way, we identify some caveats to well-known equivalences for non-deductive database languages. In Section 4 we examine two higher level constructs commonly found in DBPLs - set comprehensions and functions with inverses. We provide some key equivalences for these constructs also, and give transformation principles for expressions in them. In Section 5 we investigate extending the equivalences for the set operators to the analogous operators over bags. In Section 6 we briefly compare this work with related research. Finally, in Section 7 we give our conclusions and indicate directions of further work.

2 The Language

Our language is the λ-calculus extended with constructors, *let* expressions and pattern-matching λ-abstractions:

$$\begin{aligned}
expr &= var \mid constructor \mid primitive \mid \text{``}\lambda\text{''}pattern\text{``.''}expr \mid expr_1\ expr_2 \mid \\
&\quad \text{``}let\text{''}\ var\ \text{``=''}\ expr_1\ \text{``}in\text{''}\ expr_2 \mid \text{``(''}\ expr\ \text{``)''} \\
pattern &= var \mid constructor\ pattern_1\ \ldots\ pattern_n
\end{aligned}$$

where tuples $(e_1, \ldots, e_n)$ are regarded as applications of an n-ary constructor $Tuple_n$ to n arguments $e_1, \ldots, e_n$. We use x, y, z for denoting variables, p, q, r for patterns, and e, e' for expressions.

An occurrence of a variable x in an expression e is a *bound* occurrence if it occurs in a sub-expression of e of the form $\lambda p.e'$ or $let\ p\ =\ e'\ in\ e''$, and x also occurs in p; otherwise the occurrence of x is *free* in e. $FV(e)$ (respectively, $BV(e)$) denotes the set of variables with at least one free (respectively, bound) occurrence in e. We denote by $e[e'/x]$ the expression obtained by replacing all free occurrences of x in e by e' while renaming bound variables in e as necessary to avoid changing free variables of e' to bound ones in e.

Expressions in this extended λ calculus are straight-forwardly mapped into the ordinary λ calculus for evaluation (see Peyton-Jones 1987 for details). Computation in the λ calculus proceeds by syntactically transforming terms using β and η reduction. β reduction rewrites an application of the form $(\lambda x.e)\ e'$ to the expression $e[e'/x]$. η reduction rewrites an application $(\lambda x.e\ x)$ to e, provided that $x \notin FV(e)$. The denotational semantics of the λ-calculus (see Schmidt 1986) assigns to each expression a value in a semantic domain - this is the meaning of the expression. β and η reduction are semantically sound in that they do not alter the meaning of an expression.

Expressions of the form $let\, x = e'\, in\, e$ in the extended λ calculus into translate into expressions $(\lambda x.e)e'$ in the ordinary λ calculus. The semantic soundness of β reduction thus gives the equivalence:

let/1 $\quad let\, x = e'\, in\, e \quad = \quad e[e'/x]$

This equivalence can be used to abstract common sub-expressions when used in a right-to-left direction, and to expand definitions in place when used in a left-to-right direction. The former operation will typically be useful at the end of the query transformation process, while the latter will be useful at its outset in order to generate an overall expression to optimise.

Functions are defined by equations of the form $f = e$. If $f \in FV(e)$ i.e. if f is recursively defined, the meaning (or value) of f is given by the least fixed point of the higher-order (and non-recursive) function $\lambda f.e$. This meaning may just be non-termination for some arguments, so the semantic domain contains for each type t an element $\perp_t$ which denotes 'no information' and represents a non-terminating computation (sometimes we omit the type subscript t when it can be inferred from context). For example, the boolean type consists of the elements $True$, $False$ and $\perp_{Bool}$, where $\perp_{Bool}$ is less informative than both $True$ and $False$ (written $\perp_{Bool} \sqsubseteq True$ and $\perp_{Bool} \sqsubseteq False$) and where $True$ and $False$ are not information-wise comparable. The meaning of the recursive function $f = \lambda x.not(f\, x)$ is then given by the least fixed point of the higher-order function $\lambda f.\lambda x.not(f\, x)$, and is just the function that maps all its arguments to $\perp_{Bool}$ i.e. $\lambda x.\perp_{Bool}$.

For the purposes of investigating query optimisation we use several items of information about expressions:

Referential transparency. This is a property enjoyed by our language and means that every occurrence of an expression denotes the same value in a given environment (an environment being a mapping of free variables to expressions).

Termination. The evaluation of an expression e terminates if the value of e contains no $\perp$ elements. In the sequel, whenever we say that an expression e is *non-terminating* we mean that its value contains $\perp$; otherwise, we say that e is *terminating*, or *terminates*.

Determining whether the evaluation of an expression will terminate is of course undecidable in general. There is, however, a wide class of expressions whose evaluation is known to terminate, namely well-typed, non-recursive expressions: this is the *strong normalisation theorem* (Hindley and Seldin 1986). Furthermore, it is often possible to construct a proof of the termination of an expression by using *structural induction* (see below), and the user could be permitted to annotate the expression as such.

Strictness of functions. The order in which β reduction is applied in λ expressions is significant. Lazy evaluation (which we assume) ensures termination whenever possible by only evaluating the arguments to a function if needed by the function to return a result. A function is *strict* in an argument if that argument must be evaluated for the function to return a result. One way to characterise a strict function is to state that $f\bot = \bot$ (i.e. given a non-terminating argument, f will not terminate either). Information about the strictness properties of a function can be derived from the known strictness properties of the built-in functions using *strictness analysis* (Clack and Peyton-Jones 1985).

Continuity of functions. Any function defined in the λ notation is *continuous* (see, for example, Schmidt 1986 Theorem 6.24). This has two important implications. First, it guarantees that any recursive definition has a unique meaning. Second, it means that when proving an equivalence of the form $\forall x. f\,x = g\,x$, where f and g are continuous functions, induction over the structure of x can be used to prove the equivalence even if x is non-terminating i.e. contains $\bot$ elements. In the terminology of Schmidt 1986 an equivalence is an *inclusive predicate* for which fixpoint induction is valid. Bird and Wadler 1988 give an accessible discussion of structural induction and use it to prove equivalences over infinite lists and trees. It can similarly be used to prove our equivalences over, possibly infinite, sets below.

Sets are important in our language, and we now briefly review their semantics. The computation of a set commences with no information about the set. The set becomes successively better defined through the introduction of new elements or through the better approximation of existing elements. The computation terminates when no more elements can be added and existing elements can be refined no further.

The *Plotkin powerdomain* (Plotkin 1976, Schmidt 1986) is commonly used to model this kind of computation. Specifically, the Plotkin powerdomain, $P(t)$, over a type t consists of subsets of t that represent all the possible results of a non-deterministic or parallel computation with a finite number of alternatives at each branch point. The information-wise ordering on two sets $A, B \in P(t)$ is defined as follows:

$$A \sqsubseteq B \quad iff \quad \forall a \in A. \exists b \in B. a \sqsubseteq b \;\; and \;\; \forall b \in B. \exists a \in A. a \sqsubseteq b$$

Thus, the least element of $P(t)$ is $\{\bot_t\}$. The empty set is not an element of $P(t)$ since all non-deterministic computations must have *some* result, even if this is just non-termination, $\bot_t$. Also, infinite sets contain $\bot_t$ because if a computation returns infinitely many results it executes for ever. For example, the value of $nats = \lambda n.\{n\} \cup (nats(n+1))$ is $\lambda n.\{n, n+1, \ldots, \bot_{Num}\}$.

If t is an enumerated type, the members of $P(t)$ are all non-empty subsets of t that are either finite or contain $\bot_t$. If t is a structured type, non-equal subsets of t may convey the same information, in which case they become identified in $P(t)$. For example, $\{\{True, \bot\}, \{True, False, \bot\}, \{True, False\}\} \sqsubseteq \{\{True, \bot\}, \{True, False\}\}$ and $\{\{True, \bot\}, \{True, False\}\} \sqsubseteq \{\{True, \bot\}, \{True, False, \bot\}, \{True, False\}\}$, so these two sets are indistinguishable in $P(P(Bool))$.

The Plotkin powerdomain is clearly indicated to model our computation of sets. However, we also allow the possibility of a set-valued function returning an empty set, and so the structure of a set type over a type t is a *coalesced sum* (Schmidt 1986) of $P(t)$ with a two-element domain $Empty(t) = \{\bot, \{\}\}$ representing the empty set of type t. The coalesced sum is so called because the least elements of its summands (in this case $\{\bot_t\}$ from $P(t)$ and $\bot$ from $Empty(t)$) coalesce to form the least element of the sum. We use the least element of $P(t)$, $\{\bot_t\}$, to denote the least element of the overall set type. Figure 1 shows the structure of $P(Bool)$.

2.1 The type system

Our language is strongly, statically typed and supports a number of primitive types such as *Bool*, *Str* and *Num*. The user can declare new enumerated types and introduce new constants of such a type. For every user-defined enumerated type T, a built-in zero argument function *allT* returns all constants of that type.

We will use as a running example a database that records results for the Winter Olympics. The user-defined enumerated types include *Comp* (competitor id), *Sex*, *Country*, *Category* (category of events) and *Event*, where:

allComp = {*C1, C2, C3, ...*}
allSex = {*Male, Female*}
allCountry = {*Norway, Austria, France, ...*}
allCategory = {*Alpine, Nordic, FigureSkating, ...*}
allEvent = {*MensDownhill, WomensSlalom, ...*}

Also supported are polymorphic product, list, set and function types. In particular, $(t_1, \ldots, t_n)$ is an n-product type for any types $t_1, \ldots, t_n$, $[t]$ is a list type and $\{t\}$ a set type for any type t, and $t_1 \rightarrow t_2$ the type of functions from a type t_1 to a type t_2. Note that the function type constructor $\rightarrow$ is right associative, so that $t_1 \rightarrow t_2 \rightarrow t_3$ and $t_1 \rightarrow (t_2 \rightarrow t_3)$ are synonymous.

We use the notation $e :: t$ to indicate that an expression e has type t. We also use letters from the start of the alphabet to indicate type variables in type expressions. For example, the infix 'compose' function, $\circ$, defined by $(f \circ g)\ x = f\ (g\ x)$ is of type $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$, where the type variables a, b and c can be instantiated to any type.

The user can also declare sum types and introduce new constructors of such a type c.f. the list constructors $(:) :: a \rightarrow [a] \rightarrow [a]$ and $[\,] :: [a]$.

2.2 Built-in functions

The usual arithmetic $(+, -, *, /)$ and comparison $(==, !=, <, <=, >, >=)$ operators are built-in (note that $==$ is the syntactic equality operator as opposed to equality, $=$, in the semantic domain). These operators may be written either infix or prefix (in which case they are bracketed e.g. $(+)\ 1\ 2$), and are of necessity strict in both their arguments. For example, the value of $e == e'$ will be $\perp$ if either e or e' has value $\perp$: operationally, both operands of $==$ must be evaluated in order to determine if they are equal, and if either yields 'no information' so does the overall evaluation of the equality test. With structured values, the arguments to constructors are compared left-to-right. Thus, for example, $(1 : \perp) == (2 : [\,])$ returns $False$ whereas $(1 : \perp) == (1 : [\,])$ returns $\perp$. The other main comparison operator, $<$, works in a similar fashion and hinges on an alphanumeric ordering of constants and constructors where, by convention, $[\,] < (h : t)$ for any list $(h : t)$. Thus, for example, $[\,] < (\perp : \perp)$ and $(1 : \perp) < (2 : \perp)$ both return $True$. Functions cannot be meaningfully compared by these comparison operators, i.e. $\perp$ is returned. Lastly, sets are converted to lists (by the operator *set_to_list* below) for comparison.

The 3-argument conditional function if is also built-in and has the following semantics:

$$\begin{aligned}
&if\ \perp_{Bool}\ x\ y\ = \perp\\
&if\ True\ x\ y\ = x\\
&if\ False\ x\ y = y
\end{aligned}$$

Thus, if is strict in its first argument, but not in its second and third arguments. Logical operators can be defined in terms of if as follows:

$$\begin{aligned}
and &= \lambda x.\lambda y.if\ x\ y\ False\\
or &= \lambda x.\lambda y.if\ x\ True\ y\\
not &= \lambda x.if\ x\ False\ True
\end{aligned}$$

Consequently these too are strict in their first argument. We will also require on occasion counterparts to *and* and *or* that are commutative and '$\perp$-avoiding'; operationally, $\vee$ and $\wedge$ are implemented by evaluating their operands in parallel:

$$\begin{array}{llll}
True \vee y & = True & False \wedge y & = False\\
x \vee True & = True & x \wedge False & = False\\
False \vee y & = y & True \wedge y & = y\\
x \vee False & = x & x \wedge True & = x\\
\perp \vee \perp & = \perp & \perp \wedge \perp & = \perp
\end{array}$$

Two set-building functions are also built-in, the singleton-forming operator and the union operator:

$$\begin{aligned}
\{_\} &:: a \to \{a\}\\
_ \cup _ &:: \{a\} \to \{a\} \to \{a\}
\end{aligned}$$

A set can also be represented by enumerating its elements, where $\{e_1, \ldots, e_n\}$ is equivalent to $\{e_1\} \cup \ldots \cup \{e_n\}$.

Breazu-Tannen *et al.* 1991 propose a function, Φ, for folding a binary operator *op* into a finite set:

$$\begin{array}{ll}
\Phi\ f\ op\ e\ \{\} & = e \\
\Phi\ f\ op\ e\ \{x\} & = f\ x \\
\Phi\ f\ op\ e\ (s\ \cup\ s') & = op\ (\Phi\ f\ op\ e\ s)\ (\Phi\ f\ op\ e\ s')
\end{array}$$

where e and *op* must form a commutative-idempotent monoid on the return type of f in order for this definition to have a unique meaning i.e. they must satisfy the following conditions:

$$\begin{array}{lll}
x\ op\ (y\ op\ z) & = (x\ op\ y)\ op\ z & \\
e\ op\ x & = x\ op\ e & = x \\
x\ op\ y & = y\ op\ x & \\
x\ op\ x & = x &
\end{array}$$

These conditions arise from the semantics of the set union operator i.e. its associativity, the fact that $\{\}$ is the identity element, its commutativity, and its idempotence.

In our case of possibly infinite sets the second equation for Φ has to be modified, giving ϕ below:

$$\begin{array}{ll}
\phi\ f\ op\ e\ \{\} & = e \\
\phi\ f\ op\ e\ \{x\} & = if\ (x = \bot)\ \bot\ (f\ x) \\
\phi\ f\ op\ e\ (s\ \cup\ s') & = op\ (\phi\ f\ op\ e\ s)\ (\phi\ f\ op\ e\ s')
\end{array}$$

In operational terms, ϕ keeps distributing f to the elements of an infinite set for ever. In semantic terms, the modified definition ensures that ϕ is monotonic. For example, $\{\bot\} \sqsubseteq \{\}$ implies that $\phi\ f\ op\ e\ \{\bot\} \sqsubseteq \phi\ f\ op\ e\ \{\}$ must hold, which in turn implies that $\phi\ f\ op\ e\ \{\bot\} \sqsubseteq e$ must hold for all possible e, and this can only be true if $\phi\ f\ op\ e\ \{\bot\} = \bot$. Thus, as one might expect, we cannot use ϕ to devise terminating cardinality or summation functions for infinite sets. Others have defined similar functions to Φ e.g. the 'pump' operator of Bancilhon *et. al.* 1987 and Beeri and Kornatzky 1990, and the 'hom' operator of Ohori *et. al.* 1989. Breazu-Tannen *et al.* 1991 give a comparison of these.

We can use ϕ to define a membership operator over, possibly infinite, sets:

$$x\ in\ s \quad = \quad \phi\ ((==)\ x)\ (\vee)\ False\ s$$

in returns $True$ if any comparison of x with an element of s returns $True$, $False$ if all comparisons of x with elements of s return $False$, and $\bot$ otherwise. So in can only return $False$ for finite sets: in operational terms, in keeps on searching an infinite set for a value that equals x until it finds one. Also, since in depends on the results of equality tests, it will return $\bot$ if applied to higher-order sets i.e. sets of functions.

We can also use ϕ to define a conversion function from sets to lists, where $++$ is the usual list append operator:

$$set\_to\_list\ s \quad = \phi\ (\lambda x.[x])\ (\lambda xs.\lambda ys.remove\_duplicates\ (sort\ (xs ++ ys)))\ []\ s$$

Note that since set_to_list depends on $sort$ it will return $\bot$ if applied to higher-order or infinite sets. Given set_to_list we can determine the cardinality of a set, the sum of its elements etc. We can also extend the comparison operators ($==, !=, <, <=, >, >=$) to sets by converting the sets to lists i.e.

$$s\ comparison\_op\ s' \quad = (set\_to\_list\ s)\ comparison\_op\ (set\_to\_list\ s')$$

2.3 User-defined functions

These can be specified using one or more equations rather than a single λ abstraction, and can use pattern-matching to deconstruct their arguments. For example, we can define a function $foldl$ which is similar to ϕ but which works over lists:

$$\begin{array}{ll} foldl\ f\ op\ e\ [] & = e \\ foldl\ f\ op\ e\ (x : xs) & = op\ (f\ x)\ (foldl\ f\ op\ e\ xs) \end{array}$$

This function can be used to convert a list to a set:

$$list\_to\_set\ xs \quad = foldl\ (\lambda x.\{x\})\ (\cup)\ \{\}\ xs$$

0-ary set-valued functions can be used to represent bulk data, the assumption being that such functions are updatable by the insertion and deletion of values of the appropriate type. For our Winter Olympics Database examples, we will use functions which define: the sponsors of each country; the name, sex and country of each competitor; the category of each event; the set of competitors registered for each event; and the list of medalists in rank order for each event:

$$\begin{array}{lll} sponsors & :: & \{(Country, Sponsor)\} \\ comps & :: & \{(Comp, Name, Sex, Country)\} \\ events & :: & \{(Event, Category)\} \\ registered & :: & \{(Event, \{Comp\})\} \\ results & :: & \{(Event, [Comp])\} \end{array}$$

3 The Algebra

Our algebra consists of three built-in operators. These are the $\{\_\}$ and $\_\cup\_$ operators already introduced and an operator $setmap$ which has the following semantics:

$$\begin{array}{ll} setmap & :: (a \to \{b\}) \to \{a\} \to \{b\} \\ setmap\ f\ \{\} & = \{\} \\ setmap\ f\ \{x\} & = if\ (x = \bot)\ \{\bot\}\ (f\ x) \\ setmap\ f\ (s\ \cup\ s') & = (setmap\ f\ s)\ \cup\ (setmap\ f\ s') \end{array}$$

setmap thus distributes a function of type $(a \rightarrow \{b\})$ over a set of type $\{a\}$ and returns the union of the results. A consequence of the second equation above is that if a set is infinite then so will be the result of *setmap*. Notice that $setmap\, f$ is just $\phi\, f\, (\cup)\, \{\}$: Breazu-Tannen *et al.* 1991 similarly give a finite-set version of *setmap* defined in terms of Φ.

Two functions that frequently appear in algebras are $map :: (a \rightarrow b) \rightarrow \{a\} \rightarrow \{b\}$ and $filter :: (a \rightarrow Bool) \rightarrow \{a\} \rightarrow \{a\}$ e.g. Cluet and Delobel 1992, Beeri and Milo 1992. These functions generalise relational projection and selection. Although they could be built-in for efficiency purposes, *map* and *filter* can be defined in terms of *setmap* as follows:

$$
\begin{aligned}
filter\ f\ s &= setmap\ (\lambda x.if\ (f\, x)\ \{x\}\ \{\})\ s \\
map\ f\ s &= setmap\ (\lambda x.\{f\, x\})\ s
\end{aligned}
$$

A further operation that can be expressed using *setmap* is the join of two relations according to a selection function f and a projection function g, $join :: ((a, b) \rightarrow Bool) \rightarrow ((a, b) \rightarrow c) \rightarrow \{a\} \rightarrow \{b\} \rightarrow \{c\}$:

$$
join\ f\ g\ s\ s' = setmap\ (\lambda x.setmap\ (\lambda y.if\ (f\ (x, y))\ \{g\, (x, y)\}\ \{\})\ s')\ s
$$

join subsumes the various flavours of join and product operations found in relational databases. It can also operate upon infinite sets of structured tuples. In particular, for any $x \in s$ and $y \in s'$, $g\ (x, y)$ is present in $join\ f\ g\ s\ s'$ if $f\ (x, y)$ evaluates to $True$, regardless of the finiteness or otherwise of s and s'.

3.1 Other set-theoretic operators

Other set-theoretic operators can be defined in terms of the operators above, although these too could be built-in for efficiency purposes. We give definitions for two of these operators, since they raise some interesting issues. Operators such as *nest*, *unnest* and *powerset* are also easily defined in our language.

Set difference can be defined using *filter* and *in*:

$$
\begin{aligned}
\_minus\_ &:: \{a\} \rightarrow \{a\} \rightarrow \{a\} \\
s\ minus\ s' &= filter\ (\lambda x.not\, (x\ in\ s'))\ s
\end{aligned}
$$

Thus *minus* will terminate if both s and s' are terminating, or if s is terminating and is a subset of s'.

Intersection can also be defined using *filter* and *in*:

$$
\begin{aligned}
\_inter\_ &:: \{a\} \rightarrow \{a\} \rightarrow \{a\} \\
s\ inter\ s' &= filter\ (\lambda x.x\ in\ s')\ s
\end{aligned}
$$

However, this definition is not in general commutative e.g. $\{3\}\ inter\ \{3, \perp\} = \{3\}$ whereas $\{3, \perp\}\ inter\ \{3\} = \{3, \perp\}$. Clearly it is desirable for intersection to be commutative for optimisation purposes, and to achieve this we can use $\wedge$:

$s \; inter \; s' \;\; = filter \; (\lambda x.(x \; in \; s) \; \wedge \; (x \; in \; s')) \; (s \; \cup \; s')$

This definition is both less efficient and has worse termination properties than the first e.g. $\{3\}$ *inter* $\{3, \perp\}$ now gives $\{3, \perp\}$, but is nevertheless the one we assume for optimisation purposes. If, however, both s and s' are known to be terminating then the original definition can safely be used in its place.

3.2 Equivalences

We now investigate some equivalences for the functions defined above. Some of these are generalisations of well-known equivalences for relational databases (Jarke and Koch 1984, Ullman 1989). The first set of equivalences, with their stated provisos, follow easily from the definitions of the logical operators in Section 2.2:

if/1	*if e1 (if e2 e3 e4) e4*	= *if (e1 and e2) e3 e4*	
if/2	*if e1 e3 (if e2 e3 e4)*	= *if (e1 or e2) e3 e4*	
if/3	*if (not e1) e2 e3*	= *if e1 e3 e2*	
if/4	*f (if e1 e2 e3)*	= *if e1 (f e2) (f e3)*	provided *f* is strict
and/1	*e1 and e2*	= *e2 and e1*	provided *e1* = $\perp$ iff *e2* = $\perp$
and/2	$e1 \wedge e2$	$= e2 \wedge e1$	
or/1	*e1 or e2*	= *e2 or e1*	provided *e1* = $\perp$ iff *e2* = $\perp$
or/2	$e1 \vee e2$	$= e2 \vee e1$	
not/1	*not (not e1)*	= *e1*	
not/2	*not (e1 or e2)*	= *(not e1) and (not e2)*	

The $\cup$ and *inter* operators obey the expected properties of commutativity and associativity (in operational terms, this means that the two branches of a $\cup$ must be evaluated in parallel), while *inter* and *minus* distribute over $\cup$:

$\cup$/1	$s \cup s'$	$= s' \cup s$	
$\cup$/2	$s \cup (s' \cup s'')$	$= (s \cup s') \cup s''$	
$\cup$/3	$s \cup s'$	$= s$	if $s' \subseteq s$
$\cap$/1	$s \; inter \; s'$	$= s' \; inter \; s$	
$\cap$/2	$s \; inter \; (s' \; inter \; s'')$	$= (s \; inter \; s') \; inter \; s''$	
$\cap$/3	$(s \; inter \; s'') \cup (s' \; inter \; s'')$	$= (s \cup s') \; inter \; s''$	
$-$/1	$(s \; minus \; s'') \cup (s' \; minus \; s'')$	$= (s \cup s') \; minus \; s''$	

However, the following equivalences only hold subject to the stated provisos:

$\cap$/4	$s \; inter \; s'$	$= s'$	if $s' \subseteq s$	provided s is terminating
$-$/2	$s \; minus \; s'$	$= s$	if $s \cap s' = \{\}$	provided s and s' are terminating
$-$/3	$s \; minus \; s'$	$= \{\}$	if $s \subseteq s'$	provided s is terminating

To illustrate the proviso associated with $\cap$/4 consider the case $s = \{1, 2, \perp\}$ and $s' = \{1\}$, whence $s' \subseteq s$, but $s \; inter \; s' = \{1, \perp\} \neq s'$. To illustrate the provisos associated with $-/2$ consider the two cases $s = \{1\}$, $s' = \{2, \perp\}$ and $s = \{1, \perp\}$, $s' = \{2\}$. For the proviso associated with $-/3$ consider the case $s = s' = \{1, \perp\}$.

The reason for the above provisos is the definition of both *inter* and *minus* in terms of the set membership operator, *in*. In particular, $x\ in\ s$ undertakes equality tests between x and elements of s which may result in the value $\perp$, as discussed in Section 2.2.

The set membership operator, *in*, itself obeys the following properties subject to the stated provisos:

in/1	$e\ in\ \{\}$	$= False$	
in/2	$e\ in\ \{e'\}$	$= e\ ==\ e'$	
in/3	$e\ in\ (s\ \cup\ s')$	$= (e\ in\ s)\ \vee\ (e\ in\ s')$	
in/4	$e\ in\ (s\ inter\ s')$	$= (e\ in\ s)\ \wedge\ (e\ in\ s')$	provided $e,\ s,\ s'$ are terminating
in/5	$e\ in\ (s\ minus\ s')$	$= (e\ in\ s)\ \wedge\ not\ (e\ in\ s')$	provided $e,\ s,\ s'$ are terminating

To illustrate that e must be terminating for in/4 and in/5 consider the case $e = \perp$, $s = \{1\}$, $s' = \{2\}$, while to illustrate that s and s' must be terminating consider the case $e = 1$, $s = \{\perp\}$, $s' = \{\}$ for in/4 and the case $e = 1$, $s = \{\perp\}$, $s' = \{1\}$ for in/5.

$\cup/3$, $\cap/4$ and $-/3$ allow us to simplify the following expressions involving the built-in functions *allT*, provided s is terminating:

all/1	$s\ \cup\ allT$	$= allT\ \cup\ s$	$= allT$
all/2	$s\ inter\ allT$	$= allT\ inter\ s$	$= s$
all/3	$s\ minus\ allT$	$= \{\}$	

A number of optimisations apply to *setmap*, and hence also to operators defined in terms of *setmap* such as *filter* and *map*:

setmap/1 $\quad setmap\ f\ (s\ \cup\ s') = (setmap\ f\ s)\ \cup\ (setmap\ f\ s')$

setmap/2 $\quad setmap\ f\ (setmap\ g\ s) = setmap\ (\lambda x.setmap\ f\ (g\ x))\ s$

setmap/3 $\quad setmap\ (\lambda x.if\ (x\ in\ s')\ e\ e')\ s\ =$
$(setmap\ (\lambda x.e)\ (s\ inter\ s'))\ \cup\ (setmap\ (\lambda x.e')\ (s\ minus\ s'))$
provided $s = \{\} \Rightarrow \perp \notin s'$

setmap/4 $\quad setmap\ (\lambda x.setmap\ (\lambda y.e)\ s')\ s\ = setmap\ (\lambda y.setmap\ (\lambda x.e)\ s)\ s'$
if $x\ \notin\ FV(s')$ and $y\ \notin\ FV(s)$
provided $s = \{\} \Rightarrow \perp \notin s'$ and $s' = \{\} \Rightarrow \perp \notin s$

setmap/1 states that *setmap* distributes over $\cup$. setmap/2 states that two successive applications of *setmap* can be compressed into one application with a second nested within it. setmap/3 states when application of a set membership test can be replaced by a set union while setmap/4 states when the nesting of one *setmap* within another commutes. To illustrate the provisos associated with these last two equivalences consider the case $s = \{\}$, $s' = \{\perp\}$.

The main optimisations for *map* are to combine successive applications into one. In particular map/2 below corresponds to combining cascades of projections:

map/1 $map\ f\ (map\ g\ s) = map\ (f \circ g)\ s$

map/2 $map\ (\lambda q.r)\ (map\ (\lambda p.q)\ s) = map\ (\lambda p.r)\ s$ if $FV(r) \subseteq FV(q) \subseteq FV(p)$

For $filter$, filter/1 below is a generalised cascade of selections, filter/2 and filter/3 combine successive applications of $filter$ and $setmap$ into a single $setmap$, filter/4 states when selection distributes over difference, and filter/5 states when two selections commute:

filter/1 $filter\ f\ (filter\ g\ s) = filter\ (\lambda x.(g\ x)\ and\ (f\ x))\ s$

filter/2 $setmap\ f\ (filter\ g\ s) = setmap\ (\lambda x.if\ (g\ x)\ (f\ x)\ \{\})\ s$

filter/3 $filter\ g\ (setmap\ f\ s) = setmap\ (\lambda x.filter\ g\ (f\ x))\ s$

filter/4 $filter\ f\ (s\ minus\ s') = (filter\ f\ s)\ minus\ (filter\ f\ s')$
provided s and s' are terminating and $(f\,x)$ is terminating for terminating x

filter/5 $filter\ f\ (filter\ g\ s) = filter\ g\ (filter\ f\ s)$
provided $\forall x.f\,x = \perp\ iff\ g\,x = \perp$

To illustrate the provisos for filter/4 consider the two cases $f = \lambda x.False$, $s = \{1\}$, $s' = \{\perp\}$ and $f = \lambda x.\perp$, $s = \{1\}$, $s' = \{1\}$. The proviso for filter/5 follows easily from the equivalences filter/1 and and/1.

We have the expected equivalences regarding combining selection with join (join/1 below) and distributing selection over join (join/2):

join/1 $filter\ f\ (join\ f'\ g\ s\ s') = join\ (\lambda z.(f'\ z)\ and\ (f\ (g\ z)))\ g\ s\ s'$

join/2 $join\ (\lambda(x,y).(f\ x)\ and\ (f'\ y))\ g\ s\ s' =$
$setmap\ (\lambda x.if\ (f\ x)\ (setmap\ (\lambda y.if\ (f'\ y)\ \{g(x,y)\}\ \{\})\ s')\ \{\})\ s$
provided s' is terminating and $(f\ x)$ is terminating for terminating x

To illustrate the provisos for join/2 consider the two cases $f = \lambda x.False$, $s = \{1\}$, $s' = \{\perp\}$ and $f = \lambda x.\perp$, $s = \{1\}$, $s' = \{\}$.

In summary, most of the expected equivalences for the logical and set operators hold. In some cases, however, we require a priori knowledge about the termination properties of expressions. The provisos associated with the many of the equivalences arise from the semantics of the built-in functions, and built-in functions with different semantics, e.g. sequential $\vee$, $\wedge$ and $\cup$, would give rise to different provisos.

The equivalences above can be proved by structural induction over the set arguments. Since sets are constructed by successive unions of singleton sets and the empty set, structural induction over a set s has two base cases which must first be proved: $s = \{\}$ and $s = \{e\}$. The induction hypothesis is then that the given proposition holds for sets $s1$ and $s2$, from which it remains to show that it holds for $s = s1 \cup s2$. For equivalences involving two sets, structural induction is employed for one set within each case of the structural induction over the other set.

The main class of equivalences which do not have counterparts in our language are the commutative laws for joins and products. However, if records are used instead of tuples (c.f. Ohori *et al.* 1989) these equivalences also apply, subject to the same proviso as for setmap/4 above since the definition of $join$ consists of a nesting of one $setmap$ within another.

3.3 Transformation Principles

Essentially the same principles apply to our language as to relational algebra expressions (Ullman 1989), except that they need to be successively applied starting from the outermost level of an expression and moving through to expressions nested within aggregation functions:

(i) use filter/1 in a right to left direction, to split up complex filter conditions;

(ii) perform *filter* as early as possible by commuting it with other applications of *filter* and *setmap* (setmap/4, filter/5), eliminating set membership tests (setmap/3), and distributing *filter* over $\cup$ (setmap/1), *minus* (filter/4) and *join* (join/2);

(iii) perform *map* as early as possible by distributing it over $\cup$ (setmap/1);

(iv) combine cascades of *setmaps* of various kinds into a single *setmap* (setmap/2, map/1, map/2, filter/1-3, join/1);

(v) at any stage during the above steps, simplify set unions, intersections and differences whenever possible by using $\cup$/3, $\cap$/4, $-$/*, all/*, in/*;

(vi) apply setmap/4, using physical-level knowledge such as expected sizes of sets and availability of indexes.

The final step of the transformation process is to abstract common subexpressions using let/1 in a right-to-left direction.

Note that there is no general heuristic about which direction to apply setmap/1, since the size of the result returned by *setmap* cannot be predicted in general. Note also that we could derive an equivalence that moves *map* through *join* in the special case that the former is a projection and the latter a cartesian product, but this would be quite contrived. In any case, such an equivalence would go into category (iii) above.

4 Higher-Level Constructs

The algebraic equivalences discussed above are fine-grained and low level. We now examine two additional sets of equivalences at a higher conceptual level of modelling and querying: those for set comprehensions and those for functions with known inverses. Our reasons for doing so are two-fold. Firstly, both these constructs are commonly found in database languages and we wish to extend optimisations identified by others (see Section 6) to our richer computational environment. Secondly, optimising directly at this conceptual level is likely to be more efficient than first translating into the syntax of the previous section and then applying the optimisations.

4.1 Set comprehensions

The syntax of set comprehensions is as follows:

$$
\begin{array}{lcl}
set\_comprehension & = & \text{“}\{\text{”}\ expr\ \text{“}|\text{”}\ qualifiers\ \text{“}\}\text{”} \\
qualifiers & = & qualifier \mid qualifier\ \text{“;”}\ qualifiers \\
qualifier & = & generator \mid filter \\
generator & = & pattern\ \text{“}\in\text{”}\ expr \\
filter & = & expr
\end{array}
$$

For example, the following equations define a set $father$, given a set $parent :: \{(Person, Person)\}$ and a set $mother :: \{(Person, Person)\}$, and a recursive set anc:

$$
\begin{array}{ll}
father & = \{t \mid t \in parent;\ not\ (t\ in\ mother)\} \\
anc & = parent\ \cup\ \{(a, d) \mid (a, d') \in anc;\ (a', d) \in anc; a' == d'\}
\end{array}
$$

Optimisation of set comprehensions is important since these provide a unifying query formalism for relational, functional, and deductive languages. For example, the head of a set comprehension corresponds to the SELECT clause of an SQL query, the generators correspond to the FROM clause, and the filters to the WHERE clause. Trinder 1989 gives a translation of the relational calculus into list (as opposed to set) comprehensions, Paton and Gray 1990 note that DAPLEX queries are easily translated into set comprehensions, and in previous papers, e.g. Poulovassilis and Small 1993, we have observed the syntactic and semantic correspondence between set-valued functions such as $father$ and anc and the analogous Datalog predicates. However, set comprehensions are just syntactic sugar for nested applications of $setmap$ and if. In the interests of simplicity we give the translation scheme, T below, only for the case that the patterns in generators are *irrefutable* i.e. the pattern matches all the elements of the generator set. The interested reader can find the translation scheme for refutable patterns in Poulovassilis and Small 1993. In the translation equations below Q denotes a sequence of zero or more qualifiers:

$$
\begin{array}{ll}
T[\![\{e|\}]\!] & = \{T[\![e]\!]\} \\
T[\![\{e_1|e_2; Q\}]\!] & = if\ (T[\![e_2]\!])\ (T[\![\{e_1|Q\}]\!])\ \{\} \\
T[\![\{e_1|p \in e_2; Q\}]\!] & = setmap\ (\lambda p.T[\![\{e_1|Q\}]\!])\ (T[\![e_2]\!])
\end{array}
$$

For example, the definition of $father$ above translates into the following expression:

$$setmap\ (\lambda t.if\ (not(t\ in\ mother))\ \{t\}\ \{\})\ parent$$

Three classes of equivalences can be identified for comprehensions, those for qualifier interchange, for qualifier introduction/elimination, and for moving qualifiers into nested set comprehensions. These equivalences can be proved by translating expressions into the extended λ calculus of Section 2 and using structural induction. Alternatively, Wadler 1990 explores the relationship between monads and comprehensions and derives most of the equivalences below.

The following equivalences for interchanging the qualifiers in set comprehensions have well-known counterparts for list comprehensions with bag equality (Trinder 1989):

cmp/1 $\{e \mid Q;\ p1 \in s1;\ p2 \in s2;\ Q'\} = \{e \mid Q;\ p2 \in s2;\ p1 \in s1;\ Q'\}$
if $FV(p1) \cap FV(s2) = FV(p2) \cap FV(s1) = \{\}$

cmp/2 $\{e \mid Q;\ p \in s;\ f;\ Q'\} = \{e \mid Q;\ f;\ p \in s;\ Q'\}$
if $FV(p) \cap FV(f) = \{\}$

cmp/3 $\{e \mid Q;\ f;\ g;\ Q'\} = \{e \mid Q;\ g;\ f;\ Q'\}$

cmp/1 states that generators can be interchanged. It follows directly from setmap/4 and has the same proviso on *s1* and *s2* provided also that *p1* and *p2* are irrefutable. In the case of refutable patterns the equivalence may fail to hold for non-empty sets too e.g. $\{x \mid x \in \{\perp\};\ (1,y) \in \{(2,3)\}\} = \{\perp\}$ whereas $\{x \mid (1,y) \in \{(2,3)\};\ x \in \{\perp\}\} = \{\}$. cmp/2 states that a generator and a filter can be interchanged, and it requires that both s and f are terminating. Of course, if we do not mind improving the termination properties of an expression, the rule may be used in a right-to-left direction if s is known to be terminating, and in a left-to-right direction if f is known to be terminating. cmp/3 states that two filters can be interchanged. Its proof requires if/1 and and/1, and consequently this equivalence holds only if f fails to terminate whenever g does.

Numerous equivalences can be identified for eliminating qualifiers, of which the following is a representative sample:

cmp/4 $\{e \mid Q;\ f;\ g;\ Q'\} = \{e \mid Q;\ f \text{ and } g;\ Q'\}$

cmp/5 $\{e \mid Q;\ x \in \{e'\};\ Q'\} = \{e[e'/x] \mid Q;\ Q'[e'/x]\}$ if $x \notin BV(Q')$
$= \{e \mid Q;\ Q'[e'/x]\}$ if $x \in BV(Q')$

cmp/6 $\{e \mid Q;\ p \in s1;\ p \text{ in } s2;\ Q'\} = \{e \mid Q;\ p \in s1 \text{ inter } s2;\ Q'\}$ if $FV(p) \cap FV(s2) = \{\}$

cmp/4 states that two filters can be compressed into one: its proof follows directly from if/1. cmp/5 states that a generator over a singleton can be eliminated: its proof follows from the semantic soundness of β reduction. cmp/6 states that a filter can be eliminated: its proof follows from the definition of *inter* and only holds if both *s1* and *s2* are terminating.

The third set of equivalences governs the moving of qualifiers into and out of nested set comprehensions:

cmp/7 $\{e \mid Q;\ p \in s;\ Q'\} = \{e \mid Q;\ p \in \{p \mid p \in s\};\ Q'\}$

cmp/8 $\{e \mid Q;\ p \in \{p \mid Q'\};\ f;\ Q''\} = \{e \mid Q;\ p \in \{p \mid Q';\ f\};\ Q''\}$ if $FV(f) \subseteq FV(p)$

More sophisticated forms of these are

$$\{e \mid Q;\ p \in s;\ Q'\} = \{e \mid Q;\ p \in \{p' \mid p' \in s\};\ Q'\}$$
$$\{e \mid Q;\ p \in \{p' \mid Q'\};\ f;\ Q''\} = \{e \mid Q;\ p \in \{p' \mid Q';\ f'\};\ Q''\}$$

where p' is obtained from p by a renaming of variables and f' is obtained from f by the same renaming.

4.2 Transformation of set comprehensions

The following equivalences are added to steps (i) - (vi) of Section 3.3 in order to cater for set comprehensions:

(i) split up complex filter conditions using cmp/4;

(ii) perform filters as early as possible using cmp/2 and cmp/3;

(iii) none added;

(iv) eliminate redundant qualifiers using cmp/4-6;

(v) none added;

(vi) interchange groups consisting of a generator and its dependent filters using cmp/1-3 (based on physical-level knowledge);

There is however an additional seventh step, which is illustrated in our second example below:

(vii) pass filters into preceding, nested, set comprehensions using cmp/8, where they can subsequently be incorporated into the optimisation of the nested expression.

We illustrate these equivalences via two queries. The first requires the countries of women competitors who won alpine events. A naive formulation iterates through all countries, events, results and competitors, and then specifies the join condition:

$$\{c \mid c \in \mathit{allCountry};\ (\mathit{ev},\mathit{cat}) \in \mathit{events};\ (\mathit{ev}',(n{:}\mathit{ns})) \in \mathit{results};\ (\mathit{num},\mathit{name},\mathit{sex},\mathit{cc}) \in \mathit{comps};\\ \mathit{ev}' == \mathit{ev} \text{ and } \mathit{cat} == \mathit{Alpine} \text{ and } \mathit{num} == n \text{ and } \mathit{sex} == \mathit{Female} \text{ and } \mathit{cc} == c\}$$

Applying cmp/4 in a right to left direction, followed by a promotion of filters gives:

$$\{c \mid c \in \mathit{allCountry};\ (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\ (\mathit{ev}',(n{:}\mathit{ns})) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\ (\mathit{num},\mathit{name},\mathit{sex},\mathit{cc}) \in \mathit{comps};\ \mathit{num} == n;\ \mathit{sex} == \mathit{Female};\ \mathit{cc} == c\}$$

Interchange of groups of generators and their dependent filters gives:

$$\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};(\mathit{ev}',(n{:}\mathit{ns})) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\ (\mathit{num},\mathit{name},\mathit{sex},\mathit{cc}) \in \mathit{comps};\ \mathit{num} == n;\ \mathit{sex} == \mathit{Female};\\ c \in \mathit{allCountry};\ \mathit{cc} == c\}$$

or, alternatively:

$$\{c \mid (num,name,sex,cc) \in comps;\ sex == Female;\ c \in allCountry;\ cc == c;\\ (ev',(n{:}ns)) \in results;\ num == n;\ (ev,cat) \in events;\ cat == Alpine;\ ev' == ev\}$$

Compressing filters, and removing $c \in allCountry;\ cc == c$ by using in/2, followed by cmp/6 and all/2, gives the following for the first of these alternatives:

$$\{c \mid (ev,cat) \in events;\ cat == Alpine;\ (ev',(n{:}ns)) \in results;\ ev' == ev;\\ (num,name,sex,cc) \in comps;\ num == n \text{ and } sex == Female;\ c \in \{cc\}\}$$

Finally, using cmp/5 gives:

$$\{cc \mid (ev,cat) \in events;\ cat == Alpine;\ (ev',(n{:}ns)) \in results;\ ev' == ev;\\ (num,name,sex,cc) \in comps;\ num == n \text{ and } sex == Female\}$$

A similar process for the second alternative gives:

$$\{cc \mid (num,name,sex,cc) \in comps;\ sex == Female;\\ (ev',(n{:}ns)) \in results;\ num == n;\ (ev,cat) \in events;\ cat == Alpine \text{ and } ev' == ev\}$$

The second query requires the competitors sponsored by Atomic who won alpine events. A naive formulation iterates through all events, all results and all tuples of a join of competitors with sponsors over country, and then specifies a further join condition:

$$\{c \mid (ev,cat) \in events;\ (ev',(n{:}ns)) \in results;\\ (c,spn) \in \{(c,spn) \mid (c,name,sex,cc) \in comps;\ (spc,spn) \in sponsors;\ cc == spc\}\\ spn == Atomic \text{ and } ev' == ev \text{ and } cat == Alpine \text{ and } c == n\}$$

Applying cmp/4 in a right to left direction, followed by filter promotion, followed by interchange of groups of generators and their dependent filters gives two alternatives:

$$\{c \mid (ev,cat) \in events;\ cat == Alpine;\ (ev',(n{:}ns)) \in results;\ ev' == ev;\\ (c,spn) \in \{(c,spn) \mid (c,name,sex,cc) \in comps;\ (spc,spn) \in sponsors;\ cc == spc\}\\ spn == Atomic;\ c == n\}$$

or, alternatively:

$$\{c \mid (c,spn) \in \{(c,spn) \mid (c,name,sex,cc) \in comps;\ (spc,spn) \in sponsors;\ cc == spc\}\\ spn == Atomic;\\ (ev',(n{:}ns)) \in results;\ c == n;\ (ev,cat) \in events;\ ev' == ev;\ cat == Alpine\}$$

For the first of these alternatives, passing the last two filter conditions into the preceding set comprehension using cmp/8 gives:

$$\{c \mid (ev,cat) \in events;\ cat == Alpine;\ (ev',(n{:}ns)) \in results;\ ev' == ev;\\ (c,spn) \in \{(c,spn) \mid (c,name,sex,cc) \in comps;\ (spc,spn) \in sponsors;\ cc == spc;\\ spn == Atomic;\ c == n\}\}$$

Then performing filter promotion within the nested set abstraction followed by filter compression gives:

$$\{c \mid \textit{(ev,cat)} \in \textit{events; cat}{==}\textit{Alpine; } (ev',(n{:}ns)) \in \textit{results; } ev' {==} ev;$$
$$(c,spn) \in \{(c,spn) \mid (c,name,sex,cc) \in \textit{comps; } c {==} n;$$
$$(spc,spn) \in \textit{sponsors; } cc {==} spc \textit{ and } spn {==} \textit{Atomic}\}\}$$

or:

$$\{c \mid \textit{(ev,cat)} \in \textit{events; cat}{==}\textit{Alpine; } (ev',(n{:}ns)) \in \textit{results; } ev' {==} ev;$$
$$(c,spn) \in \{(c,spn) \mid (spc,spn) \in \textit{sponsors; } spn {==} \textit{Atomic;}$$
$$(c,name,sex,cc) \in \textit{comps; } cc {==} spc \textit{ and } c {==} n\}\}$$

The second alternative is optimised similarly.

4.3 Functions with known inverses

In this section we consider how use may be made of information about the inverse relationship between pairs of functions. Such information may be readily available in languages which use a functional or object-oriented data model, and may also be available from other sources e.g. via proofs supplied by the user (Harrison and Khoshnevisan 1992). In the case of extensional functions, inverses typically correspond to fast access paths provided by indexes.

Given two one-to-one functions $f :: s \to t$ and $f^{-1} :: t \to s$ which are inverses of each other i.e. equivalence inv/1 below holds, then equivalence inv/2 also holds, by the definition of *in* and *map*:

$$\begin{array}{lllll} \text{inv/1} & (f\ e)\ {==}\ e' & = & e\ {==}\ (f^{-1}\ e') \\ \text{inv/2} & (f\ e)\ \textit{in}\ s & = & e\ \textit{in}\ (\textit{map}\ f^{-1}\ s) \end{array}$$

For example, if no two competitors at the Winter Olympics have the same name then inv/1 holds for $\textit{name\_of} :: \textit{Comp} \to \textit{Name}$ and $\textit{comp\_no} :: \textit{Name} \to \textit{Comp}$.

Given two functions $f :: s \to t$ and $f^{-1} :: t \to \{s\}$ such that inv/3 below holds, then inv/4 also holds, by the definition of *in* and *setmap*:

$$\begin{array}{lllll} \text{inv/3} & (f\ e)\ {==}\ e' & = & e\ \textit{in}\ (f^{-1}\ e') \\ \text{inv/4} & (f\ e)\ \textit{in}\ s & = & e\ \textit{in}\ (\textit{setmap}\ f^{-1}\ s) \end{array}$$

For example, inv/3 holds for $\textit{country\_of} :: \textit{Comp} \to \textit{Country}$ and $\textit{team\_of} :: \textit{Country} \to \{\textit{Comp}\}$.

Finally, given two functions $f :: s \to \{t\}$ and $f^{-1} :: t \to \{s\}$ it may be the case that inv/5 holds:

$$\begin{array}{lll} \text{inv/5} \quad e'\ \textit{in}\ (f\ e) & = & e\ \textit{in}\ (f^{-1}\ e') \end{array}$$

For example, inv/5 holds for $\textit{competes\_in} :: \textit{Comp} \to \{\textit{Event}\}$ and $\textit{competitors\_of} :: \textit{Event} \to \{\textit{Comp}\}$.

4.4 Transformation of functions with inverses

Equivalences inv/1-5 are applied between steps (i) and (ii) of Section 3.3 in order to generate tests for set-membership and equality on variables i.e. tests of the form $v == e$ and $v\ in\ e$ where v is a variable.

We again illustrate the use of these equivalences on our Winter Olympics database, assuming the following functions and inverses which can be defined in terms of the base functions of Section 2.3:

$country\_of$	$:: Comp \rightarrow Country$	
$team\_of$	$:: Country \rightarrow \{Comp\}$	$//= country\_of^{-1}$; inv/3 and inv/4 applicable
$winner$	$:: Event \rightarrow Comp$	
sex	$:: Comp \rightarrow Sex$	
sex^{-1}	$:: Sex \rightarrow \{Comp\}$	// inv/3 and inv/4 applicable
$category$	$:: Event \rightarrow Category$	
$events$	$:: Category \rightarrow \{Event\}$	$//= category^{-1}$; inv/3 and inv/4 applicable

The query is the same as the first query of Section 4.2, i.e. 'Which countries have female Alpine gold medalists?', and can be expressed as follows:

$$\{c \mid c \in allCountry;\ comp \in allComp;\ e \in allEvent;$$
$$Alpine == (category\ e)\ and\ Female == (sex\ comp)$$
$$and\ comp == (winner\ e)\ and\ c == (country\_of\ comp)\}$$

First cmp/4 is repeatedly applied to break up the filter condition:

$$\{c \mid c \in allCountry;\ comp \in allComp;\ e \in allEvent;$$
$$Alpine == (category\ e);\ Female == (sex\ comp);\ comp == (winner\ e);\ c == (country\_of\ comp)\}$$

Next inv/1-5 are applied, and we reach the point:

$$\{c \mid c \in allCountry;\ comp \in allComp;\ e \in allEvent;$$
$$e\ in\ (events\ Alpine);\ comp\ in\ (sex^{-1}\ Female);\ comp == (winner\ e);\ c == (country\_of\ comp)\}$$

at which we have a choice: whether or not to apply inv/1 to $c == (country\_of\ comp)$. If we do not do so, we move on to the removal of filters using cmp/6:

$$\{c \mid e \in allEvent\ inter\ (events\ Alpine);\ comp \in allComp\ inter\ (sex^{-1}\ Female)\ inter\ \{winner\ e\};$$
$$c \in allCountry\ inter\ \{country\_of\ comp\}\}$$

The lower-level optimisations (in particular, all/2) reduce the size of the sets over which we iterate, giving:

$$\{c \mid e \in events\ Alpine;\ comp \in (sex^{-1}\ Female)\ inter\ \{winner\ e\};\ c \in \{country\_of\ comp\}\}$$

Finally, application of cmp/5 gives the first query plan:

$$\{country\_of\ comp \mid e \in events\ Alpine;\ comp \in (sex^{-1}\ Female)\ inter\ \{winner\ e\}\}$$

Alternatively, applying inv/1 again gives:

$\{c \mid c \in$ *allCountry; comp* $\in$ *allComp; e* $\in$ *allEvent;*
e in events Alpine; comp in (sex^{-1} *Female); comp==(winner e); comp in (team_of c)*$\}$

and removing the filters using cmp/6 gives:

$\{c \mid c \in$ *allCountry; e* $\in$ *allEvent inter (events Alpine);*
comp $\in$ *allComp inter (sex*$^{-1}$ *Female) inter* $\{$*winner e*$\}$ *inter (team_of c)*$\}$

Again, all/2 reduces the size of the sets over which we iterate, giving a second query plan:

$\{c \mid c \in$ *allCountry; e* $\in$ *events Alpine;*
comp $\in$ *(sex*$^{-1}$ *Female) inter* $\{$*winner e*$\}$ *inter (team_of c)*$\}$

These two query plans differ only in that the extra use of inv/1 during the construction of the second query plan did not allow the iteration through *allCountry* to be eliminated. Thus, it is likely that the first plan would be chosen for execution after applying some physical-level heuristics. We finally observe that both query plans correspond to the first plan of Section 4.2 for the same query. The second plan of Section 4.2 is not generated here due to the non-availability of an inverse for the *winner* function.

5 Equivalences for the Bag Type

We now investigate extending the equivalences developed above to the analogous operators over bags. We represent the type consisting of bags of elements of type t by $\langle t \rangle$. Analogously to sets, bags are created from: (i) the empty bag, which we represent by $\langle\ \rangle$, (ii) singleton bags, which we represent by $\langle e \rangle$, and (iii) unions of (i) and (ii).

The computation of a bag commences with no information about the bag i.e. with the element $\langle \bot_t \rangle$. The bag becomes successively better defined through the introduction of new elements or through the better approximation of existing elements. The computation terminates when no more elements can be added and existing elements can be refined no further.

Given two bags $A, B \in \langle t \rangle$, we say that a mapping $\psi : A \to B$ is *information-preserving (i-p)* if $a \sqsubseteq \psi(a)$ for all $a \in A$. The ordering on bags is then defined as follows:

$$A \sqsubseteq B \;\; iff \;\; \bot_t \notin A, \bot_t \notin B \textit{ and } \exists \textit{ an i-p bijection } \psi : A \to B, \textit{ or}$$
$$\bot_t \in A \textit{ and } \exists \textit{ an i-p injection } \psi : (A \setminus \{\bot_t\}) \to (B \setminus \{\bot_t\})$$

The first clause of this definition deals with bags that have a defined number of elements: such a bag, A, can become better-defined only by making its individual elements better-defined. The second clause of the definition deals with the case that the number of elements in A is undefined, in which case A can become better-defined by (a) defining the number of elements (in which case there is no $\perp_t$ in B), (b) adding more elements, (c) making existing elements better-defined, or (d) combinations of (a) - (c).

As a consequence of this definition, the least element of type $\langle t \rangle$ is $\langle \perp_t \rangle$. Also, for any two bags A, B such that $\perp_t \notin A$ and $\perp_t \in B$ we have $A \not\sqsubseteq B$. Finally, the only other bag that the empty bag $\langle\ \rangle$ is comparable with is the least bag $\langle \perp_t \rangle$ (where $\langle \perp_t \rangle \sqsubseteq \langle\ \rangle$). Figure 2 shows the structure of the $\langle Bool \rangle$ domain.

5.1 Operators over bags

By analogy to sets, the three fundamental operators over bags are:

$$
\begin{array}{ll}
\langle \_ \rangle & :: a \to \langle a \rangle \\
\_ \cup_B \_ & :: \langle a \rangle \to \langle a \rangle \to \langle a \rangle \\
\phi_B & :: (a \to b) \to (b \to b \to b) \to b \to \langle a \rangle \to b
\end{array}
$$

$\langle \_ \rangle$ maps its argument, e, to the singleton bag, $\langle e \rangle$, $\cup_B$ is additive with respect to the cardinalities of all elements other than $\perp$ (there can be at most one occurrence of 'no information' in a bag) and ϕ_B has the following semantics:

$$
\begin{array}{ll}
\phi_B\ f\ op\ e\ \langle\ \rangle & = e \\
\phi_B\ f\ op\ e\ \langle x \rangle & = if\ (x = \perp)\ \perp\ (f\ x) \\
\phi_B\ f\ op\ e\ (b\ \cup_B\ b') & = op\ (\phi_B\ f\ op\ e\ b)\ (\phi_B\ f\ op\ e\ b')
\end{array}
$$

In order for this definition to have a unique meaning, op and e must satisfy the first three of the four conditions given for ϕ in Section 2.2 but since $\cup_B$ is not idempotent the fourth condition for ϕ need not hold.

We can construct a bag consisting of n occurrences of a given value e by:

$$
\begin{array}{ll}
mk\_bag & :: Num \to a \to \langle a \rangle \\
mk\_bag\ n\ e & = if\ (n > 0)\ (\langle e \rangle\ \cup_B\ (mk\_bag\ (n-1)\ e))\ \langle\ \rangle
\end{array}
$$

ϕ_B can be used to define cardinality and membership operators over bags of arbitrary type, and a summation operator over bags of numbers:

$$
\begin{array}{ll}
cardinality\ b & = \phi_B\ (\lambda x.1)\ (+)\ 0\ b \\
sum\ b & = \phi_B\ (\lambda x.x)\ (+)\ 0\ b \\
x\ in\ b & = \phi_B\ ((==)\ x)\ (\vee)\ False\ b
\end{array}
$$

We can convert between bags and lists:

$$
\begin{array}{ll}
bag\_to\_list & = \phi_B\ (\lambda x.[x])\ (\lambda xs.\lambda ys.(sort\ (xs ++ ys)))\ [\ ] \\
list\_to\_bag & = foldl\ (\lambda x.\langle x \rangle)\ (\cup_B)\ \langle\ \rangle
\end{array}
$$

and we can use *bag_to_list* to extend the comparison operators ==, <, <= etc. to work over bags. We can also define a function *unique* which removes duplicates from a bag by converting the bag to a list, removing duplicates, and converting back to a bag:

$$unique \quad = list\_to\_bag \circ remove\_duplicates \circ bag\_to\_list$$

Since *bag_to_list* depends on *sort*, the functions defined in terms of it will only terminate for terminating bags.

We can convert between bags and sets as follows:

$$\begin{aligned} bag\_to\_set &= \phi_B \ (\lambda x.\{x\}) \ (\cup) \ \{\} \\ set\_to\_bag &= \phi \ (\lambda x.\langle x\rangle) \ (\lambda xs.\lambda ys.unique \ (xs \ \cup_B \ ys)) \ \langle\ \rangle \end{aligned}$$

Since *set_to_bag* depends on *unique*, it is only guaranteed to terminate for terminating sets. The following equivalences can be shown to hold over the various conversion operators, provided in each case that xs is terminating:

$$\begin{array}{lll} \text{conv/1} & (list\_to\_set \circ set\_to\_list) \ xs & = xs \\ \text{conv/2} & (bag\_to\_set \circ set\_to\_bag) \ xs & = xs \\ \text{conv/3} & (list\_to\_bag \circ bag\_to\_list) \ xs & = xs \end{array}$$

The analogue of *setmap* is *bagmap*, where $bagmap \ f = \phi_B \ f \ (\cup_B) \ \langle\ \rangle$. *bagmap* can be used to support bag comprehensions with the following syntax. The translation is analogous to that for sets but using *bagmap* rather than *setmap*:

$$bag\_comprehension = \text{“}\langle\text{”} \ expr \ \text{“}|\text{”} \ qualifiers \ \text{“}\rangle\text{”}$$

bagmap can also be used to define analogues for *map*, *filter* and *join* that operate over bags:

$$\begin{aligned} map \ f \ b &= bagmap \ (\lambda x.\langle f \ x\rangle) \ b \\ filter \ f \ b &= bagmap \ (\lambda x.if \ (f \ x) \ \langle x\rangle \ \langle\ \rangle) \ b \\ join \ f \ g \ s \ s' &= bagmap \ (\lambda x.bagmap \ (\lambda y.if \ (f \ (x,y)) \ \langle g \ (x,y)\rangle \ \langle\ \rangle) \ s') \ s \end{aligned}$$

Given *filter*, we can determine the number of occurrences of a given element in a bag:

$$count \ b \ e \quad = cardinality \ (filter \ (\lambda x.((==) \ e)) \ b)$$

Bag difference, i.e. the *monus* operator of Libkin and Wong 1993, can then be defined as follows:

$$b \ monus \ b' \quad = \langle y \mid x \in unique \ (b \ \cup_B \ b'); \ y \in mk\_bag \ ((count \ b \ x) \ - \ (count \ b' \ x)) \ x\ \rangle$$

and *sub_bag* and intersection are just:

$$\begin{aligned} b \ sub\_bag \ b' &= (b \ monus \ b') \ == \ \langle\ \rangle \\ b \ inter \ b' &= b \ monus \ (b \ monus \ b') \end{aligned}$$

Note that *monus*, and by extension *sub_bag* and *inter*, will return $\langle\bot\rangle$ if either b or b' is non-terminating.

5.2 Equivalences

$\cup_B$ and *inter* are both commutative and associative i.e. they satisfy the analogues of $\cup/1$, $\cup/2$, $\cap/1$ and $\cap/2$. Clearly, $\cup/3$ does not hold for bag union. Neither do $\cap/3$ and $-/1$ e.g. put $s = s' = s'' = \langle 1 \rangle$.

Interpreting *minus* as *monus* and $\subseteq$ as bag containment, equivalences $\cap/4$, $-/2$ and $-/3$ hold subject to the same provisos. Equivalences in/1, in/2 and in/3 also hold as does in/4 with the same provisos. However, in/5 does not hold e.g. put $e = 1$, $s = \langle 1, 1 \rangle$ and $s' = \langle 1 \rangle$.

Interpreting *setmap* as *bagmap*, setmap/1 and setmap/2 hold as does setmap/4 subject to the same provisos. However, setmap/3 does not hold e.g. put $s = \langle 1, 1 \rangle$, $s' = \langle 1 \rangle$, $e = \langle True \rangle$, $e' = \langle False \rangle$.

The analogues of map/*, filter/* and join/* hold, subject to the same provisos. Finally, of the equivalences for comprehensions, only cmp/6 does not hold for bag comprehensions e.g. $\langle x \mid x \in \langle 1, 1 \rangle; \ x \ in \ \langle 1 \rangle \rangle = \langle 1, 1 \rangle$ whereas $\langle x \mid x \in \langle 1, 1 \rangle \ inter \ \langle 1 \rangle \rangle = \langle 1 \rangle$.

6 Related Work

Our framework draws considerably from Breazu-Tannen *et al.* 1991 who proposed a programming paradigm based upon structural recursion over sets. A major motivating factor behind this paradigm is that it facilitates the optimisation of database programs. Indeed, several optimisations are stated, including filter/2, filter/3 and join/2 above, while structural induction is used as a proof technique. However, we have investigated termination issues, the possibility that sets may be infinite, and equivalences expressed at several levels of abstraction.

Cluet and Delobel 1992 propose a query optimisation formalism for O_2 based upon classes and algebraic query rewriting. One assumption made is that methods have no side-effects, although it would seem difficult in practice to guarantee that this condition holds. A select-project-join algebra is discussed, and the introduction of types into algebraic expressions facilitates its optimisation by allowing the introduction of functions that enumerate the constants of a type. Our use of the functions *allT* and the equivalences inv/1-5 is analogous. The approach of Cluet and Delobel 1992 retains a separation of DML and programming language, and so the possibility of non-termination or infinite data structures are not considered.

Several others have introduced object algebras and strategies for their optimisation (Demuth *et. al.* 1994, Lieuwin and DeWitt 1992, Shaw and Zdonik 1989, Stonebraker 1991, Vandenburg and DeWitt 1991) with analogous equivalences to those we propose here. In general these algebras are either computationally incomplete or support optimisations for only a subset of their operators. Also, some provide only limited facilities for optimising user-defined data types; while others allow few algebraic transformations to be applied to an expression without changing its value.

The optimisation of functional database languages has been examined by several other researchers e.g. Trinder 1989, Beeri and Kornatzky 1990, Erwig and Lipeck 1991, Paton and Gray 1990, Heytens and Nikhil 1991. Trinder 1989 advocates analogues of cmp/1-3 for list (as opposed to set) comprehensions. These equivalences are justified by assuming bag equality over lists. Implicit assumptions made are that lists are finite (otherwise, the equivalent of cmp/1 would not hold for example) and that functions over their elements are terminating (otherwise cmp/3 would not hold). Trinder also proposes *filter hiding*, which corresponds to a combination of our cmp/7 and cmp/8. Heytens and Nikhil 1991 too are concerned with the optimisation of list-valued expressions, and in this context propose combining unary expressions (analogous to our map/1 and filter/1-3), eliminating iteration over functions of the form *allT*, and using information regarding indexes to select preferred query paths. Paton and Gray 1990 discuss the optimisation of DAPLEX queries, essentially using cmp/1-3 and inv/1-5. Beeri and Kornatzky 1990 and Erwig and Lipeck 1991 discuss the optimisation of FP-like functional database languages. Beeri and Kornatzky extend the laws for transforming relational queries to laws over more general, finite, data collections. Erwig and Lipeck develop similar equivalences in the context of strict functions over finite sets; a set of equivalences over inverse functions are also given, including inv/1, and others that we have not discussed here.

With respect to previous work on bags, Albert 1991 defines two bag operators $\cup$ and $\cap$ which reduce to set-theoretic union and intersection when restricted to bags without duplicates. He explores the relationship between $\cup$, $\cap$ and two further operators $\sqcup$ and $\setminus$, which correspond to our $\cup_B$ and *monus* respectively. A number of equivalences are developed for applications of $filter$ over bags, some of which generalise to our language but others of which require caveats on the termination properties of the bags and/or the filter conditions.

Libkin and Wong 1993 investigate the power of query languages over bags and give comprehensive references to other work on the expressiveness of bag query languages. However, they do not consider termination issues, and thus use an information-wise ordering over bags which does not explicitly take into account the presence or absence of $\perp$ in a bag.

Finally, the optimisation of aggregation functions is a topic of recent interest e.g. Chaudhuri and Shim 1994. For example, it is possible to push a maximise or minimise operation into a set comprehesion:

$$max\ \{x \mid Q;\ x \in s\} = max\ \{max\ s \mid Q\}$$
$$min\ \{x \mid Q;\ x \in s\} = min\ \{min\ s \mid Q\}$$

As we would expect, the bag data type is more amenable to aggregation optimisations than the set data type. In particular it is interesting to note that by defining bag aggregation operations (such as $cardinality$ and sum in Section 5 above) in terms of ϕ_B we automatically obtain several of the equivalences given by Chaudhuri and Shim. For example, we can push summation through bag union since:

$$
\begin{aligned}
sum\ (b\ \cup_B\ b') &= \phi_B\ (\lambda x.x)\ (+)\ 0\ (b\ \cup_B\ b') \\
&= (\phi_B\ (\lambda x.x)\ (+)\ 0\ b)\ +\ (\phi_B\ (\lambda x.x)\ (+)\ 0\ b') \\
&= (sum\ b)\ +\ (sum\ b')
\end{aligned}
$$

We can similarly push cardinality through bag union.

7 Concluding Remarks

We have investigated algebraic query optimisation in the context of a functional DBPL furnished with a set bulk data type. We have examined the extent to which prior work on the optimisation of relational languages can be utilised. The declarative nature of our language has enabled us to avoid the problems associated with side effects, whilst its well defined semantics provides a framework in which to show formally termination properties of expressions and equivalences between expressions. We have identified caveats to several well-known equivalences in this richer computational paradigm. For processing tasks such as aggregation and transitive closure, our optimisations can be fully exploited for all sub-expressions of a query since there is no dichotomy between the optimisation of 'programs' and 'DML statements'.

Although developed in the context of a functional language, our findings are directly applicable to other DBPLs operating over sets and/or bags. Conversely, we can incorporate equivalences discovered by others into our formalism. Finally, although we have concentrated upon showing equivalences relevant to the set and bag data types, the same approach can be used for other, possibly user-defined, data types: see for example the equivalences shown in Bird and Wadler 1988 for list and tree data types.

From our findings in Sections 3.2 and 4.1, it is clear that the user must be provided with sophisticated tools to aid the optimisation process. Such tools have already been developed for functional languages, examples being strictness analysis (Clack and Peyton-Jones 1985) and Cambridge LCF (Paulson 1987), which can be used to prove properties of experssions such as equivalence and termination. It is also clear that further work remains to be done into physical-level heuristics for reducing the query search space.

Another important issue is the optimisation of retrieval from recursively defined sets. For example, consider the functions *parent* and *anc* given in Section 4.1 above. There are several ways in which the function *anc* can be used, depending upon whether we are searching for specified values of the first and/or the second component of its tuples. For example, we can pose queries which find Janet's descendants and the ancestors of Republican presidents, respectively:

$\{d \mid (a, d) \in anc; a == Janet\}$
$\{a \mid (a, d) \in anc; (president\ d)\ and\ (republican\ d)\}$

In Poulovassilis and Small 1993 we discussed a class of functions called *selectors* which generalise the inverse functions of a functional data model by allowing associative look-up into n-ary, as opposed to just binary, relations. We proposed a magic-sets like rewriting of set-valued functions given specific binding patterns, or adornments. For example, for the *parent* and *anc* functions the possible adornments are (ϕ, ϕ), (ϕ, β), (β, ϕ) and (β, β), where β denotes 'bound' and ϕ denotes 'free'. When rewritten, *anc* gives rise to the following selectors for the (β, ϕ) and (ϕ, β) adornments, where the argument p is a predicate corresponding to the β adornment which tuples must satisfy:

$$
\begin{array}{ll}
anc^{(\beta,\phi)} & :: (Person \rightarrow Bool) \rightarrow \{Person\} \\
anc^{(\beta,\phi)}\ p & = parent^{(\beta,\phi)}\ p\ \cup\ \{z \mid y \in anc^{(\beta,\phi)}\ p;\ z \in anc^{(\beta,\phi)}\ ((==)\ y)\}
\end{array}
$$

$$
\begin{array}{ll}
anc^{(\phi,\beta)} & :: (Person \rightarrow Bool) \rightarrow \{Person\} \\
anc^{(\phi,\beta)}\ p & = parent^{(\phi,\beta)}\ p\ \cup\ \{x \mid y \in anc^{(\phi,\beta)}\ p;\ x \in anc^{(\phi,\beta)}\ ((==)\ y)\}
\end{array}
$$

Often the predicate p will simply be a test for equality with a given value, but more generally it can be *any* boolean valued function. In particular, the two queries above rewrite to:

$$
\begin{array}{l}
anc^{(\beta,\phi)}\ ((==)\ Janet) \\
anc^{(\phi,\beta)}\ (\lambda x.(president\ x)\ and\ (republican\ x))
\end{array}
$$

In a forthcoming paper we show formally that our rewriting scheme for pushing selection through recursion can only improve the termination properties of expressions.

Acknowledgements

We are grateful to the anonymous referees for their pertinent remarks on a previous version of this paper. The work described here has been supported by the U.K. Engineering and Physical Sciences Research Council (grant no. GR/J 98134).

References

[Alb91] Albert, J. Algebraic properties of bag data types, *Proc. 17th VLDB*, Barcelona, September 1991, pp 211-219.

[Aug84] Augustsson, L. A compiler for lazy ML, *Proc. ACM Symposium on Lisp and Functional Programming*, Austin, August 1984, pp 218-27.

[Ban87] Bancilhon, F. Briggs, T. Khoshafian, S. and Valduriez, P. FAD, a powerful and simple database language, *Proc. 13th VLDB*, Brighton, September 1987. pp 97-106.

[Bee90] Beeri, C. and Kornatzky, Y. Algebraic optimization of object-oriented query languages, *Proc. 3rd ICDT*, Paris, December 1990, pp 72-88. Springer-Verlag LNCS 470.

[Bee92] Beeri, C. and Milo, T. Functional and predicative programming in OODBs, Proc. ACM PODS, San Diego, June 1992, pp 176-190.

[Bir88] Bird, R. and Wadler, P. *Introduction to functional programming*, Prentice-Hall, 1988.

[Bre91] Breazu-Tannen, V. Buneman, P. and Naqvi, S. Structural recursion as a query language, *Proc. 3rd DBPL*, Nafplion, August 1991, pp 9-19.

[Bur77] Burstall, R.M. and Darlington, J. A transformation system for developing recursive programs, *JACM*, 24(1), 1977, pp 44-67.

[Cha94] Chaudhuri, S. and Shim, K. Including group-by in query optimisation, *Proc. 20th VLDB*, Santiago, September 1994, pp 354-366.

[Cla85] Clack, C. and Peyton-Jones, S. Strictness analysis - a practical approach, *Proc. FPCA*, Nancy, September 1985, pp 35-49. Springer-Verlag LNCS 201.

[Clu92] Cluet, S. and Delobel, C. A general framework for the optimization of object-oriented Queries, *Proc. ACM SIGMOD*, San Diego, June 1992, pp 383-392.

[Dem94] Demuth, B. Geppert, A. Gorchs, T. Algebraic query optimisation in the CoOMS structurally object-oriented database system, *Query Processing for Advanced Database Systems*, eds. Freytag, Maier and Vossen, Morgan Kaufman, 1994.

[Erw91] Erwig, M. and Lipeck, U. A functional DBPL revealing high level optimizations, *Proc. 3rd DBPL*, Nafplion, August 1991, pp 306-321.

[Fre89] Freytag, J.C. and Goodman, N. On the translation of relational queries into iterative programs, *ACM Trans. on Database Systems*, 14(1), 1989, pp 1-27.

[Har92] Harrison, P. and Khoshnevisan, H. The mechanical transformation of data types, *The Computer Journal*, 35(2), 1992, pp 138-147.

[Hey91] Heytens, M.L. and Nikhil, R.S. List comprehensions in Agna, a parallel persistent object system, *Proc. FPCA*, Cambridge, August 1991, pp 569-591. Springer-Verlag LNCS 523.

[Hin86] Hindley, J.R. and Seldin, J.P. *Introduction to combinators and λ-calculus*, C.U.P., 1986.

[Jar84] Jarke, M. and Koch, J. Query optimisation in database systems, *ACM Computing Surveys*, 16(2), 1984, pp 111-152.

[Joh84] Johnsson, T. Efficient compilation of lazy evaluation, *Proc. ACM Conference on Compiler Construction*, Montreal, June 1984, pp 58-69.

[Lie92] Lieuwin, D. and DeWitt, D. A transformation - based approach to optimizing loops in database programming languages, *Proc. ACM SIGMOD*, San Diego, June 1992, pp 91-100.

[Lib93] Libkin, L. and Wong, L. Some Properties of Query Languages for Bags, *Proc. 4th DBPL*, New York, August 1993, pp 97-114. Springer-Verlag Workshops in Computing.

[Oho89] Ohori, A. Buneman, P. and Breazu-Tannen, V. Database programming in Machiavelli - a polymorphic language with static type inference, *Proc. ACM SIGMOD*, Portland, June 1989, pp 46-57.

[Pat90] Paton, N.W. and Gray, P.M.D. Optimising and executing DAPLEX queries using Prolog, *The Computer Journal*, 33(6), 1990, pp 547-555.

[Pau87] Paulson, L.C. *Logic and computation: interactive proof with Cambridge LCF*, C.U.P., 1986.

[Pey87] Peyton-Jones, S.L. *The Implementation of Functional Programming Languages,* Prentice Hall, 1987.

[Plo76] Plotkin G. A powerdomain construction, *SIAM Journal of Computing*, 5(3), 1976, pp 452-487.

[Pou93] Poulovassilis, A. and Small, C. A domain-theoretic approach to integrating logic and functional database languages, *Proc. 19th VLDB*, Dublin, August 1993, pp 416-428.

[Sch86] Schmidt, D.A. *Denotational Semantics*, Allyn and Bacon, 1986.

[Sha89] Shaw, G.B. and Zdonik, S.B. An object oriented query algebra, *Proc. 2nd DBPL*, Oregon, June 1989, pp 103-112.

[Sto91] Stonebraker, M. Managing persistent objects in a multi-level store, *Proc. ACM SIGMOD*, Denver, May 1991, 2-11.

[Tri89] Trinder, P. *A functional database*, D. Phil. Thesis, Oxford University, 1989.

[Ull89] Ullman, J.D. *Principles of database and knowledge-base systems*, Vol. 2, Computer Science Press, 1989.

[Wad90] Wadler, P. Comprehending Monads, *Proc. ACM Conf. on Lisp and Functional Programming*, Nice, June 1990, pp 61-78.

[Van91] Vandenburg, S.L. and DeWitt, D.J. Algebraic support for complex objects with arrays, identity and inheritance, *Proc. ACM SIGMOD*, Denver, May 1991, pp 158-167.