Investigation of 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 techniques 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 richer set of equivalences to be developed. The support of a set bulk data type enables much prior work on the optimisation of relational languages to be utilised. Finally, 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.

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 can be shown to hold.

(ii) Some bulk data structures are inherently hard to optimise. For example, lists only readily support the common relational optimisations if the concept of *bag equality* is used [Tri89] (i.e. lists are equal if they contain the same elements, although possibly in different orders), whilst some of the algebraic properties of sets fail for bags [Alb91].

(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 may 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 $A \times B$ (since all tuples of the resulting product will have the same first coordinate).

(v) DBPLs typically support *user-defined data types*, and hence require mechanisms to prove equiva-

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

lences over these data types too.

In this paper we investigate optimisation techniques for DBPLs by addressing some of the above issues. We undertake our investigation in the context of a purely declarative functional language. Since database algebras are functional in nature, this is a particularly natural computational paradigm in which to investigate query optimisation in DBPLs. 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). Finally, 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).

The structure of this paper is as follows. In Section 2 we give the syntax of our language and briefly 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 abstractions* (also known as *set comprehensions* in the literature), 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 briefly compare this work with related research. Finally, in Section 6 we give our conclusions and indicate directions of further work.

2 The Language

The formal foundation of any functional language is the λ-calculus [Hin86]. Expressions in this calculus have the following syntax:

$$\begin{aligned} expr \;=\; & var \mid primitive \mid \text{“}\lambda\text{''} var \text{“.''} expr \mid \\ & expr_1\ expr_2 \mid \text{“(''}\ expr\ \text{“)''} \end{aligned}$$

A variable x is said to be *bound* in an expression e if it occurs in a sub-expression of e of the form $\lambda x.e'$; otherwise it is *free* in e. $FV(e)$ (respectively, $BV(e)$) denotes the set of variables with at least one free (respectively, bound) occurrence in e.

Computation in the λ calculus proceeds by syntactically transforming terms using β reduction. This rewrites a function application of the form $(\lambda x.e)e'$ to the expression $e[e'/x]$ obtained by replacing all free occurrences of x in e by e'. The denotational semantics of the λ-calculus (see [Sch86]) assigns to each expression a value in a semantic domain - this is the meaning of the expression. β reduction is semantically sound in that it does not alter the meaning of an expression.

The language that we will be optimising is the λ-calculus extended with constructors, *let* expressions and pattern-matching λ-abstractions:

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

where tuples $(e_1, ..., e_n)$ are regarded as applications of an n-ary constructor $Tuple_n$ to n arguments $e_1, ..., e_n$. We use v, w, x, y, z for denoting variables, p, q, r for patterns, and e, e' for expressions. This extended λ calculus is straight-forwardly mapped into the (ordinary) λ calculus (see [Pey87]). In particular, $let\, x = e'\, in\, e$ translates into $(\lambda x.e)e'$. The semantic soundness of β reduction thus gives the first equivalence:

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

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 of f is given by the least fixed point of the higher-order (and non-recursive) function $\lambda f.e$ (see [Sch86]). This meaning may just be non-termination for some arguments e.g. for the function $f = \lambda x.not(f\, x)$. Thus, 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 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 $f = \lambda x.not(f\, x)$ 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 *infinite* we mean that its value contains $\perp$; otherwise, we say that e is *finite*. Determining whether e is finite 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* [Hin86]. Furthermore, it may be possible to construct a proof of the finiteness of an expression, perhaps using structural induction (see below) over its definition, 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 functions 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 $f\perp = \perp$ (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* [Cla85].

Continuity of functions. A function f is said to be *continuous* if, for every sequence of values $d_1 \sqsubseteq d_2 \sqsubseteq ...$ in the domain of f, $f(\sqcup\{d_1, d_2, ...\}) = \sqcup\{f\,d_1, f\,d_2, ...\}$. In other words, continuous functions preserve least upper bounds. Any function defined in the λ notation is continuous ([Sch86] 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 a proposition 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 infinite i.e. contains $\perp$ elements. In the terminology of [Sch86] a proposition of the above form is an *inclusive predicate* for which fixpoint induction is valid. [Bir88] gives an accessible discussion of structural induction and uses it to prove equivalences over infinite lists and trees. We similarly use it to prove our equivalences over, possibly infinite, sets below.

Sets are important in our language, so we briefly recall their semantics (see [Pou93] for further details). The least element of the type consisting of sets of values of type t is the set $\{\perp_t\}$. For example, the value of $f = \lambda x.(fx) \cup (fx)$ is $\lambda x.\{\perp\}$, while the value of $nats = \lambda n.\{n\} \cup (nats(n+1))$ is $\lambda n.\{n, n+1, ...\perp\}$, so with $nats$ non-termination arises from the construction of an infinite set.

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, $(t_1, ..., t_n)$ is an n-product type for any types $t_1, ..., t_n$, $[t]$ is a list type and $\{t\}$ a set type for any type t, and $t_1 \to t_2$ the type of functions from a type t_1 to a type t_2. Note that the function type constructor $\to$ is right associative, so that $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ 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 $(b \to c) \to (a \to b) \to (a \to c)$, 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 $(:) : a \to [a] \to [a]$ and $[] : [a]$.

2.2 Built-in functions

The usual arithmetic $(+, -, *, /)$ and comparison $(==, != , <, <=, >, >=)$ operators are built-in [1]. These operators may be written either infix or prefix (in which case they are bracketed e.g. $(+)\,1\,2$). These operators 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. The other main

[1] Note that $==$ is the syntactic equality operator as opposed to equality, $=$, in the semantic domain.

comparison operator, <, works in a similar fashion and hinges on an alphanumeric ordering of constants and constructors, with a left-to-right comparison of the arguments of the latter. Functions cannot be meaningfully compared, while sets are converted to lists (by the operation *set_to_list* below) for comparison.

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

$$\begin{aligned} &\mathit{if}\ \bot_{Bool}\ x\ y &&= \bot \\ &\mathit{if}\ \mathit{True}\ x\ y &&= x \\ &\mathit{if}\ \mathit{False}\ x\ y &&= y \end{aligned}$$

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:

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

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

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

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 $\{e_1, ..., e_n\}$ equals $\{e_1\} \cup ... \cup \{e_n\}$.

Breazu-Tannen *et al.* [Bre91] propose a function, Φ, for folding a binary operator *op* into a finite set, requiring *e* and *op* to form a commutative-idempotent monoid on the return type of *f* in order for this definition to have a unique meaning:

$$\begin{aligned} &\Phi\ f\ op\ e\ \{\} &&= e \\ &\Phi\ f\ op\ e\ \{x\} &&= f\ x \\ &\Phi\ f\ op\ e\ (s1 \cup s2) &&= op\ (\Phi\ f\ op\ e\ s1)\ (\Phi\ f\ op\ e\ s2) \end{aligned}$$

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

$$\begin{aligned} &\phi\ f\ op\ e\ \{\} &&= e \\ &\phi\ f\ op\ e\ \{x\} &&= \mathit{if}\ (x = \bot)\ \bot\ (f\ x) \\ &\phi\ f\ op\ e\ (s1 \cup s2) &&= op\ (\phi\ f\ op\ e\ s1)\ (\phi\ f\ op\ e\ s2) \end{aligned}$$

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 continuous. 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 FAD [Ban87] and the 'hom' operator of Machiavelli [Oho89], and [Bre91] gives a comparison of these.

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

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

Thus, *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* only returns *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 deterministic conversion function from sets to lists, where *list_union* is defined in terms of a duplicate-eliminating *sort* function and a list append operator ++:

$$\begin{aligned} &\mathit{set\_to\_list}\ s &&= \phi\ (\lambda x.[x])\ \mathit{list\_union}\ [\,] \\ &\mathit{list\_union}\ xs\ ys &&= \mathit{sort}\ (xs \mathbin{++} ys) \end{aligned}$$

Given *set_to_list* we can determine the cardinality of a set, the sum of its elements etc. Finally note that since *set_to_list* depends on *sort* it will return $\bot$ if applied to higher-order or infinite sets.

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:

$$\begin{aligned} &\mathit{foldl}\ f\ op\ e\ [\,] &&= e \\ &\mathit{foldl}\ f\ op\ e\ (x{:}xs) &&= op\ (f\ x)\ (\mathit{foldl}\ f\ op\ e\ xs) \end{aligned}$$

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

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

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:

sponsors	:	$\{(Country, Str)\}$
comps	:	$\{(Comp, Str, Sex, Country)\}$
events	:	$\{(Event, Category)\}$
registered	:	$\{(Event, \{Comp\})\}$
results	:	$\{(Event, [Comp])\}$

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:

$$\begin{array}{ll} setmap & : (a \to \{b\}) \to \{a\} \to \{b\} \\ setmap\ f\ \{\} & = \{\} \\ setmap\ f\ \{x\} & = if\ (x = \perp)\ \{\perp\}\ (f\ x) \\ setmap\ f\ (s1 \cup s2) & = (setmap\ f\ s1) \cup (setmap\ f\ s2) \end{array}$$

setmap thus distributes a function of type $(a \to \{b\})$ over a set of type $\{a\}$ and returns the union of the results. Notice that *setmap f* is just $\phi\ f\ (\cup)\ \{\}$. [Bre91] similarly gives a finite-set version of *setmap* defined in terms of Φ.

Two functions that frequently appear in algebras are $map : (a \to b) \to \{a\} \to \{b\}$ and $filter : (a \to Bool) \to \{a\} \to \{a\}$ e.g. in [Clu92, Bee92]. 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, where $singleton\ f\ x = \{f\ x\}$:

$$\begin{array}{ll} filter\ f\ s & = setmap\ (\lambda x.if\ (f\ x)\ \{x\}\ \{\})\ s \\ map\ f\ s & = setmap\ (singleton\ f)\ s \end{array}$$

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, $join : ((a,b) \to Bool) \to ((a,b) \to c) \to \{a\} \to \{b\} \to \{c\}$:

$$\begin{array}{lll} join\ f\ g\ r\ s & = & setmap\,(\lambda x.setmap(\lambda y. \\ & & \quad if\ (f\ (x,y))\ \{g\ (x,y)\}\ \{\})\ s)\ r \end{array}$$

join subsumes the various flavours of join and product operations found in relational databases. It can operate upon infinite sets of structured tuples. In particular, for any $a \in r$ and $b \in s$, the value of $g(a,b)$ *in* $(join\ f\ g\ r\ s)$ is *True* provided that $f(a,b)$ is *True*, regardless of the finiteness or otherwise of r and s.

Finally, *join* obeys the following commutativity property, provided r and s are both finite or both infinite, where $f^*\ (x,y) = f(y,x)$:

$$join\ f\ g\ r\ s \quad = \quad join\ f^*\ g^*\ r\ s$$

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*:

$$\begin{array}{ll} \_minus\_ & : \{a\} \to \{a\} \to \{a\} \\ s1\ minus\ s2 & = filter\ (\lambda x.not(x\ in\ s2))\ s1 \end{array}$$

Thus *minus* will terminate if both *s1* and *s2* are finite, or if *s1* is finite and is a subset of *s2*.

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

$$\begin{array}{ll} \_inter\_ & : \{a\} \to \{a\} \to \{a\} \\ s1\ inter\ s2 & = filter\ (\lambda x.x\ in\ s2)\ s1 \end{array}$$

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. To achieve this we can use $\wedge$:

$$\begin{array}{lll} s1\ inter\ s2 & = & filter\ (\lambda x.(x\ in\ s1) \wedge (x\ in\ s2)) \\ & & \quad (s1 \cup s2) \end{array}$$

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 *s1* and *s2* are known to be finite then the original definition can safely be used in its place.

3.2 Equivalences

We now give some equivalences for the functions defined above. Some of these are generalisations of well-known equivalences for relational databases [Jar84, Ull89]. The main class of equivalences which do not have counterparts in our language are the commutative laws for joins and products. However, if records [Oho89] are used instead of tuples, these equivalences also apply, subject to the provisos stated for *join* above regarding the termination of its arguments.

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

∪/1 $s1 \cup s2 = s2 \cup s1$

∪/2 $s1 \cup (s2 \cup s3) = (s1 \cup s2) \cup s3$

∪/3 $s1 \cup s2 = s1 \quad \text{if } s2 \subseteq s1$

∩/1 $s1 \textit{ inter } s2 = s2 \textit{ inter } s1$

∩/2 $s1 \textit{ inter } (s2 \textit{ inter } s3) = (s1 \textit{ inter } s2) \textit{ inter } s3$

∩/3 $(s1 \textit{ inter } s3) \cup (s2 \textit{ inter } s3) = (s1 \cup s2) \textit{ inter } s3$

−/1 $(s1 \textit{ minus } s3) \cup (s2 \textit{ minus } s3) = (s1 \cup s2) \textit{ minus } s3$

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

∩/4 $s1 \textit{ inter } s2 = s2$
if $s2 \subseteq s1$, provided *s1* is finite

−/2 $s1 \textit{ minus } s2 = s1$
if $s1 \cap s2 = \{\}$, provided *s1* and *s2* are finite

−/3 $s1 \textit{ minus } s2 = \{\}$
if $s1 \subseteq s2$, provided *s1* is finite

To illustrate the proviso associated with ∩/4, consider the sets $s1 = \{1,2,\perp\}$ and $s2 = \{1\}$. Then $s2 \subseteq s1$, but $s1 \textit{ inter } s2 = \{1,\perp\} \neq s2$.

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

in/1 $e1 \textit{ in } \{\} = \textit{False}$

in/2 $e1 \textit{ in } \{e2\} = e1 == e2$

in/3 $e \textit{ in } (s1 \cup s2) = (e \textit{ in } s1) \vee (e \textit{ in } s2)$

in/4 $e \textit{ in } (s1 \textit{ inter } s2) = (e \textit{ in } s1) \wedge (e \textit{ in } s2)$
provided *e*, *s1*, *s2* are finite

in/5 $e \textit{ in } (s1 \textit{ minus } s2) = (e \textit{ in } s1) \wedge \textit{not}(e \textit{ in } s2)$
provided *e*, *s1*, *s2* are finite

∪/3, ∩/4 and −/3 also allow us to simplify the following expressions involving the built-in functions *allT*, provided s is finite:

all/1 $s \cup \textit{allT} = \textit{allT} \cup s = \textit{allT}$

all/2 $s \textit{ inter allT} = \textit{allT inter } s = s$

all/3 $s \textit{ minus allT} = \{\}$

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

setmap/1 $\textit{setmap } f\ (s1 \cup s2) = (\textit{setmap } f\ s1) \cup (\textit{setmap } f\ s2)$

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

setmap/3 $\textit{setmap } (\lambda x.\textit{if } (x \textit{ in } s1)\ e1\ e2)\ s = (\textit{setmap } (\lambda x.e1)\ (s \textit{ inter } s1)) \cup (\textit{setmap } (\lambda x.e2)\ (s \textit{ minus } s1))$
provided *s* is finite

setmap/4 $\textit{setmap } (\lambda x.\textit{setmap } (\lambda y.e)\ s2)\ s1 = \textit{setmap } (\lambda y.\textit{setmap } (\lambda x.e)\ s1)\ s2$
provided $x \notin FV(s2)$, $y \notin FV(s1)$, and *s1* and *s2* are both finite or both infinite

setmap/1 states that *setmap* distributes over ∪. 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. Finally setmap/4 states when the nesting of one *setmap* within another commutes. A consequence of setmap/4 is that any two *setmaps* which successively iterate over the *same* set can commute. This is especially important for the optimisation of iteration over recursively defined sets.

These equivalences are 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 s' and s'', from which it remains to show that it holds for $s = s' \cup s''$.

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 $\textit{map } f\ (\textit{map } g\ s) = \textit{map } (f \circ g)\ s$

map/2 $\textit{map } (\lambda q.r)\ (\textit{map } (\lambda p.q)\ s) = \textit{map } (\lambda p.r)\ s$
provided $FV(q) \subseteq FV(p)$

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*, and filter/4 states that selection distributes over difference:

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

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

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

filter/4 $\textit{filter } f\ (s1 \textit{ minus } s2) = (\textit{filter } f\ s1) \textit{ minus } (\textit{filter } f\ s2)$
provided *s1* and *s2* are finite and *(f a)* is finite for finite *a*

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

join/1 $\textit{filter } f1\ (\textit{join } f2\ g\ r\ s) = \textit{join } (\lambda(x,y).f2(x,y) \textit{ and } f1(g(x,y)))\ g\ r\ s$

join/2 *join $(\lambda(x,y).(f1\ x)$ and $(f2\ y))\ g\ r\ s =$*
setmap $(\lambda x.$if $(f1\ x)$ (setmap $(\lambda y.$
if $(f2\ y)\ \{g(x,y)\}\ \{\})\ s)\ \{\})\ r$
provided s finite and $(f\ a)$ finite for finite a

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

3.3 Transformation Principles

Essentially the same principles apply to our language as to relational algebra expressions [Ull89], 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 *setmap* (setmap/4), 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/*.

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 further application of setmap/4 can be made using physical-level knowledge such as expected sizes of sets and availability of indexes. Finally, note that we could also 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 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 very fine-grained and low level. We now examine two additional sets of equivalences at a higher conceptual modelling and querying level: those for set abstractions and those for functions with known inverses. Our reasons for doing so are two-fold. Firstly, both these constructs are commonly found database languages and we wish to extend optimisations identified by others (see Section 5) to our richer computational environment. Secondly, we conjecture that optimising first at this conceptual level is likely to be more efficient than proceeding directly to logical-level optimisations.

4.1 Set abstractions

The syntax of set abstractions is as follows:

set_abstraction	=	"{" *expr* "|" *qualifiers* "}"
qualifiers	=	*qualifier* | *qualifier* ";" *qualifiers*
qualifier	=	*generator* | *filter*
generator	=	*pattern* "∈" *expr*
filter	=	*expr*

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*:

$$\begin{aligned} father &= \{t \mid t \in parent;\ not(t\ in\ mother)\} \\ anc &= parent \cup \{(a,d) \mid (a,d1) \in parent; \\ &\qquad (a1,d) \in anc;\ a1 == d1\} \end{aligned}$$

Optimisation of set abstractions is important since these provide a unifying query formalism for relational, functional, and deductive languages. For example, the head of a set abstraction corresponds to the SELECT clause of an SQL query, the generators are correspond to the FROM clause, and the filters to the WHERE clause. Also, Trinder [Tri89] gives a translation of the relational calculus into list (as opposed to set) abstractions, [Pat90] notes that DAPLEX queries are easily translated into set abstractions, and in previous papers e.g. [Pou93] we have observed the syntactic and semantic correspondence between set-valued functions such as *father* and *anc* and the analogous Datalog predicates. However, set abstractions 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 simple variables: the interested reader can find the full translation scheme in [Pou93]. In the translation equations below Q denotes a sequence of zero or more qualifiers:

$$\begin{aligned} T\lceil\{e \mid\}\rceil &= \{T\lceil e\rceil\} \\ T\lceil\{e_1 \mid e_2; Q\}\rceil &= if\ (T\lceil e_2\rceil)\ (T\lceil\{e_1 \mid Q\}\rceil)\ \{\} \\ T\lceil\{e_1 \mid x \in e_2; Q\}\rceil &= setmap\ (\lambda x.T\lceil\{e_1 \mid Q\}\rceil)\ (T\lceil e_2\rceil) \end{aligned}$$

For example, *father* above translates into:

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

Three classes of equivalences can be identified for set abstractions, those for qualifier interchange, for qualifier introduction/elimination, and for moving qualifiers into nested set abstractions. These equivalences can be proved by translating expressions into the extended λ calculus and using structural induction.

The following equivalences for interchanging the qualifiers in set abstractions have well-known counterparts for list abstractions with bag equality [Tri89]:

abs/1 $\{e|\ ...;\ p1 \in s1;\ p2 \in s2;\ Q\} =$
$\{e|\ ...;\ p2 \in s2;\ p1 \in s1;\ Q\}$
provided $FV(p1) \cap FV(s2) = FV(p2) \cap FV(s1) = \{\}$

abs/2 $\{e|\ ...;\ p \in s;\ f;\ Q\} = \{e|\ ...;\ f;\ p \in s;\ Q\}$
provided $FV(p) \cap FV(f) = \{\}$

abs/3 $\{e|\ ...;\ f;\ g;\ Q\} = \{e|\ ...;\ g;\ f;\ Q\}$

abs/1 states that generators can be interchanged. It follows directly from setmap/4 and has the same proviso that s1 and s2 are both finite or both infinite. abs/2 states that a generator and a filter can be interchanged, and it requires both that s is finite and that f terminates, otherwise non-termination may be introduced. 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 finite, and in a left-to-right direction if f is known to terminate. abs/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:

abs/4 $\{e|\ ...;\ f;\ g;\ Q\} = \{e|\ ...;\ f\ \mathit{and}\ g;\ Q\}$

abs/5 $\{e|\ ...;\ x \in \{e'\};\ Q\} = \{e[e'/x]|\ ...;\ Q[e'/x]\}$
provided $x \notin BV(Q)$

abs/6 $\{e|\ ...;\ p \in s1;\ p\ \mathit{in}\ s2;\ Q\} =$
$\{e|\ ...;\ p \in (s1\ \mathit{inter}\ s2);\ Q\}$

abs/4 states that two filters can be compressed into one: its proof follows directly from if/1. abs/5 states that a generator over a singleton can be eliminated: its proof follows from the semantic soundness of β reduction. abs/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 finite.

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

abs/7 $\{e|\ ...;\ p \in s;\ Q\} = \{e|\ ...;\ p \in \{p \mid p \in s\};\ Q\}$

abs/8 $\{e|\ ...;\ p \in \{p|Q\};\ f;\ Q'\} =$
$\{e|\ ...;\ p \in \{p|Q;\ f\};\ Q'\}$
provided $FV(f) \subseteq FV(p)$

More sophisticated forms of these are

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

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 abstractions

The following transformation principles can be applied successively, moving inwards from the outermost set abstraction to nested set abstractions c.f. the lower-level transformations of 3.3:

(i) use abs/4 in a right to left direction to split up complex filter conditions;

(ii) perform filters as early as possible by interchanging them with generators and other filters using abs/2 and abs/3;

(iii) interchange groups consisting of a generator and its dependent filters by using abs/1-3, according to the expected efficiency of evaluating the group (based on physical-level knowledge such as expected sizes of sets and availability of indexes);

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

(v) at any stage during the above steps, simplify set unions, intersections and differences whenever possible by using the equivalences of Section 3;

(vi) pass filters into preceding, nested, set abstractions using abs/8.

Finally, abstract common subexpressions using let/1 in a right-to-left direction.

We illustrate these principles 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 | c ∈ allCountry; (ev,cat) ∈ events;
(ev′,nums) ∈ results;
(num,name,sex,compc) ∈ comps;
ev′ == ev and cat == Alpine and num ==
head(nums) and sex == Female and compc == c}

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

{c | c ∈ allCountry; (ev,cat) ∈ events; cat == Alpine;
(ev′,nums) ∈ results; ev′ == ev;
(num,name,sex,compc) ∈ comps;
num == head(nums); sex == Female; compc == c}

Interchange of groups of generators and their dependent filters gives:

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (\mathit{num},\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\quad \mathit{num} == \mathit{head}(\mathit{nums});\ \mathit{sex} == \mathit{Female};\\
\quad c \in \mathit{allCountry};\ \mathit{compc} == c\}
\end{array}$$

or, alternatively:

$$\begin{array}{l}
\{c \mid (\mathit{num},\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\quad \mathit{sex} == \mathit{Female};\ c \in \mathit{allCountry};\ \mathit{compc} == c;\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{num} == \mathit{head}(\mathit{nums});\\
\quad (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\ \mathit{ev}' == \mathit{ev}\}
\end{array}$$

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

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (\mathit{num},\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\quad \mathit{num} == \mathit{head}(\mathit{nums})\ \mathit{and}\ \mathit{sex} == \mathit{Female};\\
\quad c \in \{\mathit{compc}\}\}
\end{array}$$

Finally, using abs/5 gives:

$$\begin{array}{l}
\{\mathit{compc} \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (\mathit{num},\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\quad \mathit{num} == \mathit{head}(\mathit{nums})\ \mathit{and}\ \mathit{sex} == \mathit{Female}\}
\end{array}$$

A similar process for the second alternative gives:

$$\begin{array}{l}
\{\mathit{compc} \mid (\mathit{num},\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\quad \mathit{sex} == \mathit{Female};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{num} == \mathit{head}(\mathit{nums});\\
\quad (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine}\ \mathit{and}\ \mathit{ev}' == \mathit{ev}\}
\end{array}$$

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:

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ (\mathit{ev}',\mathit{nums}) \in \mathit{results};\\
\quad (c,\mathit{spname}) \in\\
\qquad \{(c,\mathit{spname}) \mid (c,\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\qquad (\mathit{spname},\mathit{spc}) \in \mathit{sponsors};\ \mathit{compc} == \mathit{spc}\}\\
\quad \mathit{spname} == \text{“Atomic”}\ \mathit{and}\ \mathit{ev}' == \mathit{ev}\ \mathit{and}\\
\quad \mathit{cat} == \mathit{Alpine}\ \mathit{and}\ c == \mathit{head}(\mathit{nums})\}
\end{array}$$

Applying abs/4 in a right to left direction followed by filter promotion gives two alternatives, one being:

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (c,\mathit{spname}) \in\\
\qquad \{(c,\mathit{spname}) \mid (c,\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\qquad (\mathit{spname},\mathit{spc}) \in \mathit{sponsors};\ \mathit{compc} == \mathit{spc}\}\\
\quad \mathit{spname} == \text{“Atomic”};\ c == \mathit{head}(\mathit{nums})\}
\end{array}$$

Passing the last two filter conditions into the preceding set abstraction using abs/8 gives:

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (c,\mathit{spname}) \in\\
\qquad \{(c,\mathit{spname}) \mid (c,\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\qquad (\mathit{spname},\mathit{spc}) \in \mathit{sponsors};\ \mathit{compc} == \mathit{spc};\\
\qquad \mathit{spname} == \text{“Atomic”};\ c == \mathit{head}(\mathit{nums})\}\}
\end{array}$$

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

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (c,\mathit{spname}) \in\\
\qquad \{(c,\mathit{spname}) \mid (c,\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\qquad c == \mathit{head}(\mathit{nums});\\
\qquad (\mathit{spname},\mathit{spc}) \in \mathit{sponsors};\\
\qquad \mathit{compc} == \mathit{spc}\ \mathit{and}\ \mathit{spname} == \text{“Atomic”}\}\}
\end{array}$$

or

$$\begin{array}{l}
\{c \mid (\mathit{ev},\mathit{cat}) \in \mathit{events};\ \mathit{cat} == \mathit{Alpine};\\
\quad (\mathit{ev}',\mathit{nums}) \in \mathit{results};\ \mathit{ev}' == \mathit{ev};\\
\quad (c,\mathit{spname}) \in\\
\qquad \{(c,\mathit{spname}) \mid (\mathit{spname},\mathit{spc}) \in \mathit{sponsors};\\
\qquad \mathit{spname} == \text{“Atomic”};\\
\qquad (c,\mathit{name},\mathit{sex},\mathit{compc}) \in \mathit{comps};\\
\qquad \mathit{compc} == \mathit{spc}\ \mathit{and}\ c == \mathit{head}(\mathit{nums})\}\}
\end{array}$$

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 programmer [Har92]. In the case of extensional functions, inverses typically correspond to fast access paths provided by indexes. An implicit assumption for the equivalences we give below is that the functions and their inverses are strict, which is necessary in order for the equivalences to hold.

Given two functions $f : s \rightarrow t$ and $f^{-1} : t \rightarrow s$ such that $f^{-1}(f\,e) = e$ for all expressions $e : s$ and $f(f^{-1}\,e) = e$ for all expressions $e : t$, then:

inv/1	$(f\ \mathit{e1}) == \mathit{e2}$	=	$\mathit{e1} == (f^{-1}\ \mathit{e2})$
inv/2	$(f\ \mathit{e1})\ \mathit{in}\ s$	=	$\mathit{e1}\ \mathit{in}\ (\mathit{map}\ f^{-1}\ s)$

Given a function $f : s \rightarrow t$ which is many-to-one and a function $f^{-1} : t \rightarrow \{s\}$ such that for all $e : s$, $e \in f^{-1}(f\,e)$ and for all $e : t$, $\forall u \in (f^{-1}e).f\,u = e$, then:

inv/3	$(f\ \mathit{e1}) == \mathit{e2}$	=	$\mathit{e1}\ \mathit{in}\ (f^{-1}\ \mathit{e2})$
inv/4	$(f\ \mathit{e1})\ \mathit{in}\ s$	=	$\mathit{e1}\ \mathit{in}\ (\mathit{setmap}\ f^{-1}\ s)$

Clearly inv/3 and inv/4 are also applicable given a one-to-many function $f : s \to \{t\}$ and its inverse $f^{-1} : t \to s$, since in this case the roles of f and f^{-1} above are reversed. Finally, given a many-to-many function $f : s \to \{t\}$ and a function $f^{-1} : t \to \{s\}$ such that for all $e : s$, $\forall u \in (f\, e).e \in (f^{-1} u)$ and for all $e : t$, $\forall u \in (f^{-1} e).e \in (f\, u)$, then:

inv/5 $\quad$ *e2 in (f e1)* $\quad = \quad$ *e1 in (f^{-1} e2)*

4.4 Transformation of functions with inverses

The following transformation principles should be applied at successive levels of nesting, and common subexpressions should be abstracted out at the end of the process:

(i) use abs/4 in a right to left direction to break up complex filter conditions;

(ii) use inv/1-5 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;

(iii) apply transformation principles (ii) - (v) for set abstractions from Section 4.2.

We again illustrate these principles 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* $\to$ *Country*
team_of	: *Country* $\to$ {*Comp*} //= *country_of*$^{-1}$
competes_in	: *Comp* $\to$ {*Event*}
comps_in	: *Event* $\to$ {*Comp*} //= *competes_in*$^{-1}$
winner	: *Event* $\to$ *Comp*
sex	: *Comp* $\to$ *Sex*
sex^{-1}	: *Sex* $\to$ {*Comp*}
category	: *Event* $\to$ *Category*
events	: *Category* $\to$ {*Event*} //= *category*$^{-1}$

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* | *c* ∈ *allCountry; comp* ∈ *allComp; ev* ∈ *allEvents;*
  Alpine== (category ev) and Female== (sex comp)
  and comp== (winner ev)
  and c== (country_of comp)}

Phase (i), which repeatedly applies abs/4 to break up the filter condition, gives:

{*c* | *c* ∈ *allCountry; comp* ∈ *allComp; ev* ∈ *allEvents;*
  Alpine== (category ev); Female== (sex comp);
  comp== (winner ev); c== (country_of comp)}

During phase (ii) inv/1-5 are applied, and we reach the point:

{*c* | *c* ∈ *allCountry; comp* ∈ *allComp; ev* ∈ *allEvents;*
  ev in (events Alpine); comp in (sex^{-1} *Female);*
  comp== (winner ev); 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 phase (iii), where the filters are removed using abs/6:

{*c* | *ev* ∈ *allEvents inter (events Alpine);*
  comp ∈ *allComp inter (sex*$^{-1}$ *Female)*
    inter {*winner ev*};
  c ∈ *allCountry inter* {*country_of comp*}}

Finally, the lower level optimisations (in particular, all/2) reduce the size of the sets over which we iterate, giving us our first query plan:

{*c* | *ev* ∈ *events Alpine;*
  comp ∈ *(sex*$^{-1}$ *Female) inter* {*winner ev*};
  c ∈ {*country_of comp*}}

Alternatively, applying inv/1 again gives:

{*c* | *c* ∈ *allCountry; comp* ∈ *allComp; ev* ∈ *allEvents;*
  ev in (events Alpine); comp in (sex^{-1} *Female);*
  comp== (winner ev); comp in (team_of c)}

and removing the filters using abs/6 gives:

{*c* | *c* ∈ *allCountry;*
  ev ∈ *allEvents inter (events Alpine);*
  comp ∈ *allComp inter (sex*$^{-1}$ *Female) inter*
    {*winner ev*} *inter (team_of c)*}

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

{*c* | *c* ∈ *allCountry; ev* ∈ *(events Alpine);*
  comp ∈ *(sex*$^{-1}$ *Female) inter* {*winner ev*}
    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 Related Work

Our framework draws considerably from Breazu-Tannen *et al.* [Bre91] 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, this research is in the context of terminating functions over finite sets and a single level at which equivalences can be expressed. In contrast, we have richer semantic domains which include $\perp$, and can thus address termination issues; we also allow for the possibility that sets may be infinite; and we have investigated equivalences expressed at several levels of abstraction.

Cluet and Delobel [Clu92] 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 [Clu92] retains a separation of DML and programming language; thus, the possibility of non-termination or infinite data structures are not considered.

Several other groups have introduced object algebras and strategies for their optimisation [Dem94, Lie92, Sha89, Sto91, Van91] 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.

Finally, the optimisation of functional database languages has been examined by several other researchers e.g. [Tri89, Bee90, Erw91, Pat90, Hey91]. Trinder [Tri89] advocates analogues of abs/1-3 for list (as opposed to set) abstractions. These equivalences are justified by assuming bag equality over lists. Implicit assumptions made are that lists are finite (otherwise, the equivalent of abs/1 would not hold for example) and that functions over their elements are terminating (otherwise abs/3 would not hold). Trinder also proposes *filter hiding*, which corresponds to a combination of our abs/7 and abs/8. [Hey91] too is concerned with the optimisation of list-valued expressions, and in this context proposes 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. [Pat90] discusses the optimisation of DAPLEX queries, essentially using abs/1-3 and inv/1-5. [Bee90, Erw91] discuss the optimisation of FP-like functional database languages, also highlighting the suitability of using functional language for DBPL optimisation. In particular, [Erw91] develops equivalences over map, filter and a Φ-like aggregation operator in the context of strict functions over finite sets; a set of inverse equivalences are also given, including inv/1, and others that we have not discussed here.

6 Conclusions

We have investigated algebraic query optimisation techniques 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. Conversely, we can incorporate equivalences discovered by others into our formalism. Finally, although we have concentrated upon showing equivalences relevant to the set data type, the same approach can be used for other, possibly user-defined, data types: see for example the equivalences shown in [Bir88] for list and tree data types.

We have not yet considered heuristics to reduce the query search space. Also, in a computationally complete language the user can easily 'program around' predefined functions and thus thwart the optimisation process. One ongoing issue is the optimisation of retrieval from recursive sets. We discussed this in [Pou93] where we addressed the optimisation of a class of set-valued functions called *selectors*. Selectors 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 the definitions of intentional selectors, given specific look-up patterns: we are now implementing these ideas. Finally, it is clear that the user must be provided with sophisticated tools if they are to aid the optimisation process. Such tools have already been developed for functional languages, examples being strictness analysis [Cla85] and Cambridge LCF [Pau87], which can be used to prove properties of functions such as equivalence and termination.

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 48818).

References

[Alb91] Albert, J. *Algebraic properties of bag data types*, Proc. 17th VLDB Conference, 1991.

[Ban87] Bancilhon, F. et al. *FAD, a powerful and simple database language*, Proc. 13th VLDB Conference, 1991.

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

[Bee92] Beeri, C. and Milo, T. *Functional and predicative programming in OODBs*, Proc. ACM PODS, 1992.

[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, Morgan-Kaufman, 1991.

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

[Clu92] Cluet, S. and Delobel, C. *A General Framework for the Optimization of Object-Oriented Queries*, Proc. ACM SIGMOD, 1992.

[Dem94] Demuth, B. Geppert, A. Gorchs, T. *Algebraic query optimisation in the CoOMS structurally object-oriented database system*, in 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, Morgan-Kaufman, 1991.

[Har92] Harrison, P. and Khoshafian, S. *The mechanical transformation of data types*, The Computer Journal, 35(2), 1992.

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

[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.

[Lie92] Lieuwin, D. and Dewitt, D. *A transformation - based approach to optimizing loops in database programming languages*, Proc. ACM SIGMOD, 1992.

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

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

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

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

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

[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, Morgan-Kaufman, 1989.

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

[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.

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