Formal foundations for optimising aggregation functions in Database Programming Languages

Alexandra Poulovassilis[†] and Carol Small[‡]

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

[‡]Autosimulations Ltd.
2 Millbanke Court, Millbanke Way, Bracknell, Berkshire, U.K.
carols@autosim.com

Abstract. This paper investigates the optimisation of aggregation functions in the context of computationally complete database programming languages and aims to generalise and provide a unifying formal foundation for previous work. We define a 'fold' operator ϕ over collection types in terms of which operations such as selection, projection, join and group-by can be defined, as well as aggregation functions such as sum, max and min. We introduce two equivalences for ϕ which respectively govern the commuting and coalescing of applications of ϕ. From these two equivalences we then formally derive equivalences governing the commuting and coalescing of iteration operations over collections, the mapping of aggregation functions over grouped collections, the introduction and elimination of aggregation functions, and the promotion of aggregation functions through iteration operations. We also show how some of these equivalences can be used to optimise comprehensions, a high-level query construct supported by many database languages.

1 Introduction

The optimisation of aggregation functions has been the subject of much recent research, prompted by the emergence of new application areas such as data warehousing. This paper contributes to this area by: generalising many previously proposed equivalences and showing that they follow from a very small set of initial equivalences; providing a framework for formally proving existing and new equivalences; and extending previous work to computationally complete database programming languages (DBPLs), in which issues of non-termination, infinite data structures and user-defined data types arise.

This introductory section begins with a summary of the DBPL that we will be considering here in Sections 1.1 and 1.2, including its syntax, semantics and type system, In Section 1.3 we define the notion of a collection type and a generic 'fold' operator, ϕ, over collection types. In Section 1.4 we give an overview of the equivalences that we will be considering, including two equivalences for ϕ from which all the others are derived. We discuss the proof framework for proving these equivalences. Our motivation for starting out with ϕ and two top-level equivalences is that many aggregation functions can be defined in terms of ϕ — and in sections 1.5 and 1.6 we show how several iteration, grouping and clustering operations can be so defined. The equivalences for ϕ then yield equivalences for these operations which need simpler or, in some cases, no provisos. Such equivalences can then be used for practical query optimisation.

Sections 2, 3, 4 and 5 form the core of the paper. In these sections we respectively introduce equivalences governing the mapping of aggregation functions over grouped collections, the introduction and elimination of aggregation functions, the promotion of aggregation functions through iteration operations, and the commuting and coalescing of iteration operations over collections. These kinds of optimisations are important because coalescing two operations into a single one can eliminate the computation of, potentially large, intermediate data structures, while commuting two operations can reduce the size of such structures.

In Section 6 we show how some of the equivalences we derive here can be applied to the optimisation of comprehensions. This is important because comprehensions provide a unifying high-level query formalism for many languages. For example, monoid comprehensions have been used to define the main features of OQL [7], and thus our work can contribute towards practical query optimisers for OQL. In Section 7 we briefly review related work. We give our concluding remarks in Section 8.

1.1 The Language

Following on from our recent work [11, 12], the language that we consider is the λ-calculus extended with constructors, pattern-matching λ-abstractions and *let* expressions. Here we give a necessarily brief description of its syntax and semantics, and full details can be found in [11, 12].

Expressions in the language have the following syntax:

$$\begin{array}{lll} expr & = & var \mid constructor \mid builtin \mid \text{“}\lambda\text{”}pattern\text{“.”}expr \mid expr_1\ expr_2 \mid \\ & & \text{“}let\text{”}\ var_1 \text{“}=\text{”}expr_1\text{“;”}\ \ldots\ \text{“;”}\ var_n\text{“}=\text{”}expr_n\ \text{“}in\text{”}\ expr \mid \\ & & \text{“(”}expr\text{“)”} \\ pattern & = & var \mid constructor\ pattern_1\ \ldots\ pattern_n \end{array}$$

Tuples, $(e_1, \ldots, e_n)$, are regarded as syntactic sugar for applications of the n-ary tuple constructor $Tuple_n$ to n arguments, $e_1, \ldots, e_n$. Functions are defined by equations of the form $f = e$, where f is a variable and e an expression.

An occurrence of a variable x in an expression e is said to be *bound* if it occurs in a sub-expression of e of the form $\lambda p.e'$ or $let\ x_1 = e'_1;\ \ldots; x_n = e'_n\ in\ e'$, and x also occurs in p or $\{x_1, \ldots, x_n\}$, respectively; otherwise the occurrence of x is *free*. $FV(e)$ denotes the set of variables with at least one free occurrence in e.

The semantics of functional languages are typically specified denotationally [13], whereby each expression is denoted by some member of a semantic domain — this is the *value* of the expression. The semantic domain contains for each type t an element $\perp_t$ which denotes any expression of type t whose evaluation fails to terminate and outputs no information (below, we omit the type subscript to $\perp$ unless there is a risk of ambiguity). Our language is *lazy* in that it ensures termination whenever possible by only evaluating the arguments to functions if needed by the function to return a result. An n-ary function f is *strict* in its i^{th} argument if that argument must be evaluated in order for the function to return a result i.e. if the value of $f\ e_1\ \ldots\ e_n$ is $\perp$ whenever the value of e_i is $\perp$.

The usual arithmetic and comparison operators are built into our language and a detailed discussion of their semantics can be found in [11]. The 3-argument conditional function if is built-in and has the following semantics:

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

We should note that here, as in all our definitions of built-in functions, the constants $\perp_{Bool}$, $True$ and $False$ refer to members of the semantic domain, in this case to possible values of expressions of type $Bool$. We should also note that the first equation implies that if is strict in its first argument.

We sometimes use $if\ e_1\ then\ e_2\ else\ e_3$ as syntactic sugar for $if\ e_1\ e_2\ e_3$. Negation can be defined in terms of if, as can conjunction and disjunction:

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

However, *and* and *or* are not commutative; in particular $and \perp False \neq and\ False \perp$ and $or \perp True \neq or\ True \perp$. So we also assume as built-in commutative operators $\vee$ and $\wedge$ with the following semantics:

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

1.2 The type system

Our language is strongly, statically typed and supports a number of primitive types, such as booleans, strings and numbers. Also, $(t_1,\ldots,t_n)$ is an n-product type for any types $t_1,\ldots,t_n$, $List\ t$ is a list type for any type t, and $t_1 \to t_2$ is the type of functions from t_1 to t_2. 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.

Analogously to the unary type constructor, $List$, we also assume as built-in the set and bag type constructors Set and Bag. Enumerated sets, bags and lists are delimited by $\{\ldots\}$, $\langle\ldots\rangle$ and $[\ldots]$ brackets, respectively. The set-union, bag-union and list-append operators are $\cup$, $\cup_B$ and $++$. We note here that the list, bag and set types are *lazy* in the sense that the result of a computation may be an infinite list, bag or set, which can be incrementally computed and output.

The denotation of such infinite structures in the semantic domain contains the $\perp$ element since their computation does not terminate.

We use the notation $e :: t$ to indicate that an expression e has type t. We 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.

1.3 Collection types

A *collection type* is one constructed by a unary type name, T, for which there exist a constant $Empty :: T\ a$ and two functions

$$\begin{array}{l} single :: a \to T\ a \\ \_ \oplus \_ :: T\ a \to T\ a \to T\ a \end{array}$$

such that $\oplus$ has $Empty$ as a left and right identity. We can define an operator, $\phi :: (a \to b) \to (b \to b \to b) \to b \to T\ a \to b$, which 'folds' a binary function op into a collection as follows:

$$\begin{array}{lll} coll/1 & \phi\ f\ op\ e\ \perp & = \perp \\ coll/2 & \phi\ f\ op\ e\ Empty & = e \\ coll/3 & \phi\ f\ op\ e\ (single\ e') & = f\ e' \\ coll/4 & \phi\ f\ op\ e\ (s\ \oplus\ s') & = op\ (\phi\ f\ op\ e\ s)\ (\phi\ f\ op\ e\ s') \end{array}$$

There is of course a different set of functions $single$, $\oplus$ and ϕ for each collection type, T. If there is a risk of ambiguity, we explicitly indicate which particular function is meant by subscripting it with its collection type.

'fold' operators and their equivalences have been the subject of much work in the functional programming and database communities, This is work too numerous to fully list here and we refer to [1, 2, 3, 4, 5] as a representative sample. Collection types were first defined by Wadler [18] where their relationship with comprehensions was explored. Our ϕ is an elaboration of the structural recursion over the union presentation (SRU) of [4, 5] to allow for non-terminating computations and infinite collections. Hence the need for $coll/1$ above which ensures that ϕ is monotonic (since $\perp \sqsubseteq Empty$, we need $\phi\ f\ op\ e\ \perp\ \sqsubseteq \phi\ f\ op\ e\ Empty$, i.e. $\phi\ f\ op\ e\ \perp\ \sqsubseteq e$, for all e; so $\phi\ f\ op\ e\ \perp$ must be equal to $\perp$).

The set, bag and list types of our language all satisfy the above definition of a collection type, with the same provisos on op and e as for SRU. For sets, $T = Set$, $Empty = \{\}$, $\oplus = \cup$ and $single = \lambda x.\{x\}$; in order for an application $\phi\ f\ op\ e\ s$ to be well-defined for any set s, op must be commutative, associative and idempotent, and must have e as a left and right identity. For bags, $T = Bag$, $Empty = \langle\rangle$, $\oplus = \cup_B$ and $single = \lambda x.\langle x \rangle$; in order for $\phi\ f\ op\ e\ b$ to be well-defined for any bag b, op must be commutative and associative, and must have e as a left and right identity. For lists, $T = List$, $Empty = []$, $\oplus = ++$ and $single = \lambda x.[x]$; in order for $\phi\ f\ op\ e\ l$ to be well-defined for any list l, op must be associative, and must have e as a left and right identity.

1.4 The Proof Framework

We now define two equivalences over expressions involving the ϕ operator from which all the remaining equivalences in the paper can be derived. Two assumptions that we make for all the equivalences that we state are that all expressions are well-typed and all applications of ϕ are well-defined. Figures 1 and 2 show a 'route-map' through the entire set of equivalences that we explore. For example, Figure 2 indicates that equivalence $ag/3$ is a special case of equivalence $\phi/3$, which is in turn a special case of $\phi/2$.

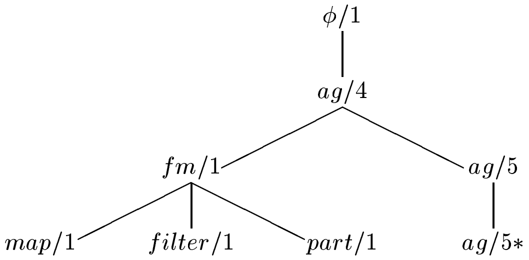

Fig. 1. Equivalences derived from $\phi/1$

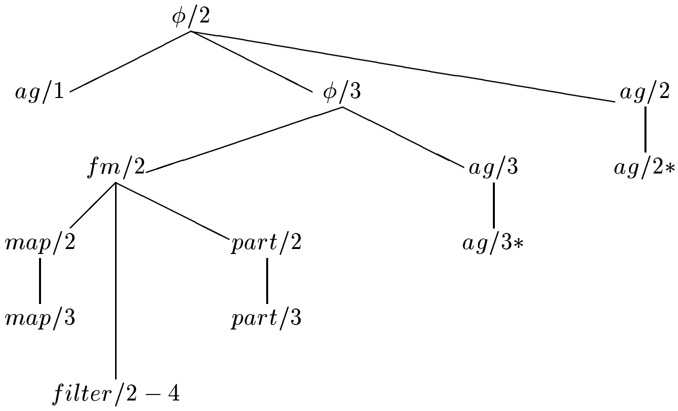

Fig. 2. Equivalences derived from $\phi/2$

The top-level equivalence of Figure 1 governs the commutativity of ϕ:

$$\phi/1 \quad g\ (h\ s)\ =\ h\ (g\ s)\ \ where\ \ g\ =\ \phi\ f\ op\ e\ \ and\ \ h\ =\ \phi\ f'\ op'\ e'$$

We note that in order for this equivalence to be well-typed, the instances of ϕ in g and h must both have type $(a \to T\ a) \to (T\ a \to T\ a \to T\ a) \to T\ a \to$

$T\ a \rightarrow T\ a$ for some collection type T. Clearly $\phi/1$ does not hold in general. In fact, in the Appendix we prove that it holds subject to the following provisos:

Provisos for $\phi/1$: $g\ e' = h\ e$, $g \circ f' = h \circ f$, and $g\ ((h\ s_1)\ op'\ (h\ s_2)) = h\ ((g\ s_1)\ op\ (g\ s_2))$.

Moving now to the top-level equivalence of Figure 2, this governs the replacement of two applications of ϕ by one:

$$\phi/2 \quad \phi\ f\ op\ e\ (\phi\ f'\ op'\ e'\ s) = \phi\ ((\phi\ f\ op\ e) \circ f')\ op''\ (\phi\ f\ op\ e\ e')\ s$$

Here op'' is an arbitrary operation satisfying the proviso stated below. In order for this equivalence to be well-typed, the inner instance of ϕ on the LHS must have type $(a \rightarrow T\ b) \rightarrow (T\ b \rightarrow T\ b \rightarrow T\ b) \rightarrow T\ b \rightarrow T\ a \rightarrow T\ b$ for some T, the outer instance of ϕ on the LHS have type $(b \rightarrow c) \rightarrow (c \rightarrow c \rightarrow c) \rightarrow c \rightarrow T\ b \rightarrow c$ and the outer instance on the RHS have type $(a \rightarrow c) \rightarrow (c \rightarrow c \rightarrow c) \rightarrow c \rightarrow T\ a \rightarrow c$. This equivalence also does not hold in general and has the following associated proviso:

Proviso for $\phi/2$: $(\phi\ f\ op\ e\ s)\ op''\ (\phi\ f\ op\ e\ \ s') = \phi\ f\ op\ e\ (s\ op'\ s')$.

A useful consequence of $\phi/2$ from which a number of further equivalences follow is $\phi/3$ which concerns the *fusion* of two applications of ϕ into one. $\phi/3$ is obtained by setting $op'' = op'$ in $\phi/2$ (giving $c = T\ b$ in the above type signatures):

$$\phi/3 \quad \phi\ f\ op\ e\ (\phi\ f'\ op'\ e'\ s) = \phi\ ((\phi\ f\ op\ e) \circ f')\ op'\ (\phi\ f\ op\ e\ e')\ s$$

Proviso for $\phi/3$: $\phi\ f\ op\ e$ distributes over op'.

In the Appendix we give proofs that $\phi/1$, $\phi/2$ and $\phi/3$ hold subject to their stated provisos. Each lower-level equivalence of Figures 1 and 2 can then be shown by substituting the more general functions in its parent equivalence by the more specific functions of the child equivalence and simplifying the associated provisos. The Appendix gives a representative sample of proofs.

1.5 Iteration operations

Using ϕ, the maximum and minimum element of a collection s can be found, where id is the identity function $\lambda x.x$:

$$max\ s = \phi\ \ id\ \ (\lambda x.\lambda y.if\ (x > y)\ x\ y)\ \ minNum\ \ s$$
$$min\ s = \phi\ \ id\ \ (\lambda x.\lambda y.if\ (x < y)\ x\ y)\ \ maxNum\ \ s$$

Cardinality and summation functions can be defined over bags and lists (but not sets since $+$ is not idempotent):

$$count\ s = \phi\ \ (\lambda x.1)\ \ (+)\ \ 0\ \ s$$
$$sum\ s\ \ = \phi\ \ id\ \ (+)\ \ 0\ \ s$$

We note that all these functions return $\perp$ for infinite collections. Finite, first-order sets can be counted or summed by converting them to sorted lists and then applying *count* or *sum* (see [11] for details).

Existential and universal quantifiers can be defined over collections, where p is a predicate:

$$\begin{aligned} &some\ p\ s = \phi\ \ p\ \ (\vee)\ \ False\ \ s \\ &all\ p\ s\ \ \ \ = \phi\ \ p\ \ (\wedge)\ \ True\ \ s \end{aligned}$$

We note that *some* $p\ s$ can return $True$ if s is infinite, but not $False$. Similarly *all* $p\ s$ can return $False$ if s is infinite but not $True$.

ϕ_T can be used to define a function $flatmap_T$ that applies a function of type $a \rightarrow T\ b$ to each element of a collection of type $T\ a$ and merges the results using $\oplus_T$:

$$\begin{aligned} &flatmap_T \quad\quad :: (a \rightarrow T\ b) \rightarrow T\ a \rightarrow T\ b \\ &flatmap_T\ f\ s = \phi_T\ f\ \ \oplus_T\ \ Empty_T\ s \end{aligned}$$

We note that $flatmap_T\ f$ is the $ext(f)$ operation of [4]. Using $flatmap_T$, functions map_T, $filter_T$ and $join_T$ can be defined which generalise relational projection, selection and join operations, respectively:

$$\begin{aligned} &map_T \quad\quad :: (a \rightarrow b) \rightarrow T\ a \rightarrow T\ b \\ &map_T\ f\ s \ \ = flatmap_T\ (single_T \circ f)\ s \end{aligned}$$

$$\begin{aligned} &filter_T \quad\quad :: (a \rightarrow Bool) \rightarrow T\ a \rightarrow T\ a \\ &filter_T\ f\ s = flatmap_T\ (\lambda x.if\ (f\ x)\ (single_T\ x)\ Empty_T)\ s \end{aligned}$$

$$\begin{aligned} &join_T \quad\quad :: (a \rightarrow b \rightarrow c) \rightarrow T\ a \rightarrow T\ b \rightarrow T\ c \\ &join_T\ f\ r\ s = flatmap_T\ (\lambda x.flatmap_T\ (\lambda y.f\ x\ y)\ s)\ r \end{aligned}$$

Henceforth we will omit the subscripts to *flatmap*, *map*, *filter* and *join* unless there is a risk of ambiguity. Due to *coll*/1, if s is infinite then so is result of *flatmap* $f\ s$. The *map*, *filter* and *join* operations can similarly operate on infinite collections. [1]

[1] Given that we are exploring equivalences derived from $\phi/1$ and $\phi/2$, and that these equivalences involve iteration over a single collection type, the above definitions of *flatmap*, *map*, *filter* and *join* suffice for the purposes of this paper. In fact, a more general function, $flatmap_{T,T'} :: (a \rightarrow T'\ b) \rightarrow T\ a \rightarrow T'\ b$, can be defined for a pair of collection types T and T': $flatmap_{T,T'}\ f\ s\ =\ \phi_T\ f\ \oplus_{T'}\ Empty_{T'}\ s$.

From the restrictions on *op* and *e* discussed in Section 1.3, in order for $flatmap_{T,T'}$ to be well-defined if $T = List$ then T' can be $List$, Bag or Set, if $T = Bag$ then T' can be Bag or Set, and if $T = Set$ then T' can only be Set. Fegaras and Maier [7] also identify these restrictions on comprehensions over multiple collection types. Functions $map_{T,T'} :: (a \rightarrow b) \rightarrow T\ a \rightarrow T'\ b$, $filter_{T,T'} :: (a \rightarrow Bool) \rightarrow T\ a \rightarrow T'\ a$ and $join_{T,T',T''} :: (a \rightarrow b \rightarrow c) \rightarrow T\ a \rightarrow T'\ b \rightarrow T''\ c$ can be defined using $flatmap_{T,T'}$, with similar restrictions on possible combinations of T, T' and T''.

1.6 Grouping and clustering operations

We now see how grouping and clustering operations can be defined in terms of ϕ, leading the way to identifying equivalences for grouping and clustering in subsequent sections of the paper. We first define an operation $group_{T,T'}$ which takes a collection of type $T\ (t,t')$ and returns a collection of type $T\ (t, T'\ t')$, for some T and T', by grouping together all pairs with the same first component. We next define a clustering operation, $partition_T$, which takes a collection $xs :: T\ t$ and a function g and returns a partition of xs, $xss :: T\ (T\ t)$, such that: (i) for each collection $xs_i \in xss$ and each pair of elements $x, y \in xs_i$, $g\ x = g\ y$, and (ii) for each distinct pair of collections $xs_i, xs_j \in xss$ and each pair of elements $x \in xs_i, y \in xs_j$, $g\ x \neq g\ y$.

group and *partition* are both defined in terms of a binary operator, m, which merges two collections into one by applying a predicate p to pairs of elements x and y drawn from each collection and, if $p\ x\ y$ holds, replacing x and y by $f\ x\ y$ for a given function f:

$$
\begin{array}{ll}
m & :: (a \to a \to Bool) \to (a \to a \to T\ a) \to T\ a \to T\ a \\
m\ p\ f\ Empty\ s & = s \\
m\ p\ f\ (single\ x)\ Empty & = single\ x \\
m\ p\ f\ (single\ x)\ (single\ y) & = if\ (p\ x\ y) \\
& \quad then\ (f\ x\ y) \\
& \quad else\ (single\ x) \oplus (single\ y) \\
m\ p\ f\ (single\ x)\ (s \oplus s') & = if\ (some\ (p\ x)\ s) \\
& \quad then\ (m\ p\ f\ (single\ x)\ s) \oplus s' \\
& \quad else\ if\ (some\ (p\ x)\ s') \\
& \qquad then\ s \oplus (m\ p\ f\ (single\ x)\ s') \\
& \qquad else\ (single\ x) \oplus (s \oplus s') \\
m\ p\ f\ (s \oplus s')\ s'' & = m\ p\ f\ s\ (m\ p\ f\ s'\ s'')
\end{array}
$$

We observe from the fourth equation above that the two arguments to m are assumed to be already "merged" in the sense that, for either argument, if the argument is of the form $s \oplus s'$ then $m\ p\ f\ s\ s' = s \oplus s'$. Subject to this assumption, m has *Empty* as a left and right identity.

$group_{T,T'}$ is then defined as follows:

$$
\begin{array}{ll}
group_{T,T'} & :: T\ (a,a') \to T\ (a, T'\ a') \\
group_{T,T'}\ s & = \phi_T\ nest\ merge\ Empty_T\ s
\end{array}
$$

where:

$$
\begin{array}{ll}
nest\ (x,y) & = single_T\ (x, single_{T'}\ y) \\
merge\ s\ s' & = m\ sameFirst\ mergeSecond\ s\ s' \\
sameFirst\ (x,ys)\ (x',ys') & = x = x' \\
mergeSecond\ (x,ys)\ (x',ys') & = single_T\ (x, ys \oplus_{T'} ys')
\end{array}
$$

Similar grouping operations can be defined which operate on n-tuples or records, and which group over arbitrary combinations of their components, not just the first component. However, for ease of exposition, we confine ourselves to grouping over collections of pairs. The equivalences that we develop for *group* would hold also for grouping operations over collections of n-tuples or records.

Finally, $partition_T$ is defined as follows:

$$
\begin{array}{ll}
partition_T & :: (a \rightarrow b) \rightarrow T\ a \rightarrow T\ (T\ a) \\
partition_T\ g\ s = \phi\ nestP\ (mergeP\ g)\ Empty\ s &
\end{array}
$$

where:

$$
\begin{array}{ll}
nestP\ x & = single\ (single\ x) \\
mergeP\ g\ s\ s' & = m\ (\lambda s.\lambda s'.allSame\ g\ (s \oplus s'))\ mergeNest\ s\ s' \\
allSame\ g\ s & = (join\ (\lambda x.\lambda y.if\ (g\ x \neq g\ y)\ (single\ (x,y))\ Empty)\ s\ s) \\
 & \quad = Empty \\
mergeNest\ s\ s' & = single\ (s \oplus s')
\end{array}
$$

The comprehension notation discussed in Section 6 below gives a perhaps clearer definition of *allSame*: $allSame\ g\ s\ =\ [(x,y) \mid x \leftarrow s;\ y \leftarrow s;\ g\ x \neq g\ y]\ =\ Empty$.

Henceforth we will omit the subscripts to *group* and *partition*, unless there is a risk of ambiguity. We note that in order for $m\ p\ f\ s$ not to lose information, $p\ x\ y$ must terminate for all pairs of elements x and y of s. In the case of $group\ s$, this means that *sameFirst* must terminate for all arguments, which requires the first component of each tuple in s to be terminating and first-order (its second component need not be). In the case of $partition\ g\ s$, $allSame\ g\ s'$ must terminate for all finite subcollections s' of s, which requires $g\,x$ to terminate for all elements of s. Subject to these provisos, *merge* and $mergeP\ g$ have the same idempotence, commutatitivy and associativity properties as $\oplus$.

2 Mapping aggregation functions over grouped collections

The application of *group* to a collection of pairs followed by the mapping of an aggregation function *aggr* over the second component of each resulting pair can be optimised into a single application of ϕ. In particular, let *nest*, *merge* and *group* be as defined in Section 1.6 and *aggr*, h and $merge'$ be defined as follows for some *op* and e:

$$
\begin{array}{ll}
aggr & = \phi\ id\ op\ e \\
h & = map\ (\lambda(x,ys).(x, aggr\ ys)) \\
merge'\ s\ s' & = merge\ (flatmap\ nest\ s)\ (flatmap\ nest\ s')
\end{array}
$$

Then the following equivalence holds:

$$ag/1 \qquad h\ (group\ s) = \phi\ (h \circ nest)\ (h \circ merge')\ Empty\ s$$

For example, a function *groupSum* can be defined which performs a group operation, followed by a summation operation:

$$
\begin{aligned}
groupSum\ s &= h\ (group\ s)\\
&\quad \textit{where } h\ =\ map\ (\lambda(x, ys).(x, sum\ ys))
\end{aligned}
$$

However, this definition undertakes two applications of ϕ and can be optimised to the following definition which undertakes only one application of ϕ:

$$
\begin{aligned}
groupSum\ s &= \phi\ (h \circ nest)\ (h \circ merge')\ Empty\ s\\
&\quad \textit{where } h\ =\ map\ (\lambda(x, ys).(x, sum\ ys))
\end{aligned}
$$

More generally, a generic 'group-and-compute' function, gc, that takes an aggregation function, $aggr = \phi\ id\ op\ e$, as an argument can be defined:

$$
\begin{aligned}
gc\ aggr\ s &= h\ (group\ s)\\
&\quad \textit{where } h\ =\ map\ (\lambda(x, ys).(x, aggr\ ys))
\end{aligned}
$$

and this definition can be optimised to the following definition which undertakes a single application of ϕ:

$$
\begin{aligned}
gc\ aggr\ s &= \phi\ (h \circ nest)\ (h \circ merge')\ Empty\ s\\
&\quad \textit{where } h\ =\ map\ (\lambda(x, ys).(x, aggr\ ys))
\end{aligned}
$$

3 Introducing/eliminating aggregation functions

The following equivalence can be shown for any function j and collection s of the appropriate type by appealing to $\phi/2$, where $f\ =\ \phi\ f_1\ op\ (f_1\ e)$ and $g\ =\ \phi\ f_2\ op\ e$:

$$ag/2 \quad f\ (map\ (g \circ j)\ s) = f_1 \circ g\ (flatmap\ j\ s)$$

Provisos for $ag/2$: f_1 is strict and distributes over op.

The following equivalences are instances of $ag/2$ (and all have $f_1 = j = id$):

$$
\begin{array}{lll}
ag/21 & max\ (map\ max\ xss) & = max\ (flatmap\ id\ xss)\\
ag/22 & min\ (map\ min\ xss) & = min\ (flatmap\ id\ xss)\\
ag/23 & sum\ (map\ sum\ xss) & = sum\ (flatmap\ id\ xss)\\
ag/24 & sum\ (map\ count\ xss) & = count\ (flatmap\ id\ xss)\\
ag/25 & some\ id\ (map\ (some\ p)\ xss) & = some\ p\ (flatmap\ id\ xss)\\
ag/26 & all\ id\ (map\ (all\ p)\ xss) & = all\ p\ (flatmap\ id\ xss)
\end{array}
$$

The following equivalence follows from $\phi/3$ and concerns the idempotence of group-and-compute operations:

$$ag/3 \quad (gc\ aggr) \circ (gc\ aggr') = gc\ aggr'$$

where $aggr\ =\ \phi\ id\ op\ e$ and $aggr'\ =\ \phi\ f\ op\ e$ for some f, op and e. Instances of $ag/3$ are:

$$\begin{array}{lll} ag/31 & (gc\ max) \circ (gc\ max) & = gc\ max \\ ag/32 & (gc\ min) \circ (gc\ min) & = gc\ min \\ ag/33 & (gc\ sum) \circ (gc\ sum) & = gc\ sum \\ ag/34 & (gc\ sum) \circ (gc\ count) & = gc\ count \end{array}$$

4 Promoting aggregation functions

The following equivalence governs the promotion of a function $g = \phi\ f\ op\ e$ through an iteration over a collection s:

$$ag/4 \quad g\ (flatmap\ j\ s) = flatmap\ j\ (g\ s)$$

j can be any function of the appropriate type, subject to the following.

Provisos for $ag/4$**:** $flatmap\ j\ e = e$, $g \circ j = (flatmap\ j) \circ f$, and $flatmap\ j$ distributes over op.

A useful instance of $ag/4$ is $ag/5$ which governs the promotion of a group-and-compute operation $gc\ aggr$, where $aggr = \phi\ id\ op\ e$:

$$ag/5 \quad gc\ aggr\ (flatmap\ j\ s) = flatmap\ j\ (gc\ aggr\ s)$$

Provisos for $ag/5$**:** $(gc\ aggr) \circ j = j$ and $flatmap\ j$ distributes over $h \circ merge'$ where $h = map\ (\lambda(x, ys).(x, aggr\ ys))$.

For example, the following instances of $ag/5$ govern the promotion of a gc operation through a product, a selection and an equijoin, respectively:

$$\begin{array}{ll} ag/51 & gc\ aggr\ (join\ f\ r\ s) \ = \ join\ f\ (gc\ aggr\ r)\ s \\ & where\ f \ = \ \lambda(a, c).\lambda b.single\ ((a, b), c) \\ ag/52 & gc\ aggr\ (filter\ (\lambda(a, b).f\ a)\ r) \ = \ filter\ (\lambda(a, b).f\ a)\ (gc\ aggr\ r) \\ ag/53 & gc\ aggr\ (join\ f\ r\ s) \ = \ join\ f\ (gc\ aggr\ r)\ s \\ & where\ f \ = \ \lambda(b, c).\lambda(a, b').if\ (b = b')\ (single\ ((a, b), c))\ Empty \end{array}$$

We note that for $ag/51$ and $ag/53$ the first of the two provisos associated with $ag/5$ holds only if there are no duplicates in the collection s e.g. if s is a set or is a bag or list without duplicates.

5 Commuting and coalescing applications of *flatmap*

$fm/1$ governs the commuting of two applications of $flatmap$ and holds subject to the stated proviso while equivalence $fm/2$ states that two successive applications of $flatmap$ can always be coalesced into one application with a second nested within it:

$$\begin{array}{ll} fm/1 & flatmap\ f\ (flatmap\ g\ s) = flatmap\ g\ (flatmap\ f\ s) \\ fm/2 & flatmap\ f\ (flatmap\ g\ s) = flatmap\ ((flatmap\ f) \circ g)\ s \end{array}$$

Proviso for $fm/1$**:** $(flatmap\ f) \circ g = (flatmap\ g) \circ f$.

The following equivalences are instances of $fm/1$ and respectively state that two applications of *map*, *filter* and *partition* commute:

$$
\begin{array}{lll}
map/1 & map\ f\ (map\ g\ s) & = map\ g\ (map\ f\ s) \\
filter/1 & filter\ f\ (filter\ g\ s) & = filter\ g\ (filter\ f\ s) \\
part/1 & \multicolumn{2}{l}{flatmap\ (partition\ g)\ (flatmap\ (partition\ g')\ xss) =} \\
 & \multicolumn{2}{l}{flatmap\ (partition\ g')\ (flatmap\ (partition\ g)\ xss)}
\end{array}
$$

They have the following provisos associated with them:

Proviso for $map/1$**:** $f \circ g = g \circ f$.

Proviso for $filter/1$**:** For all x, $f\ x = \bot \iff g\ x = \bot$.

Proviso for $part/1$**:** $\oplus$ is commutative.

Two equivalences derived from $fm/2$ are:

$$
\begin{array}{ll}
part/2 & flatmap\ (partition\ g)\ (flatmap\ (partition\ g')\ xss)\ = \\
 & flatmap\ ((flatmap\ (partition\ g)) \circ (partition\ g'))\ xss \\
part/3 & flatmap\ (partition\ g)\ (flatmap\ (partition\ g')\ xss)\ = \\
 & flatmap\ (partition\ (\lambda x.(g\ x, g'x))\ xss
\end{array}
$$

$part/2$ is an instance of $fm/2$. $part/3$ follows from $part/2$ by showing that $(flatmap\ (partition\ g)) \circ (partition\ g') = partition\ (\lambda x.(g\ x, g'x))$ and has the same proviso as $part/1$. Other equivalences that can be derived from $fm/2$ include $map/2$ and $map/3$ which govern the combining of two successive applications of *map* into one (c.f. combining cascades of projections), $filter/2$ which is a generalised cascade of selections, and $filter/3$ and $filter/4$ which combine successive applications of *filter* and *flatmap* into a single *flatmap*:

$$
\begin{array}{lll}
map/2 & map\ f\ (map\ g\ s) & = map\ (f\ \circ\ g)\ s \\
map/3 & map\ (\lambda p'.e)\ (map\ (\lambda p.p')\ s) & = map\ (\lambda p.e)\ s \\
filter/2 & filter\ f\ (filter\ g\ s) & = filter\ (\lambda x.(g\ x)\ and\ (f\ x))\ s \\
filter/3 & flatmap\ f\ (filter\ g\ s) & = flatmap\ (\lambda x.if\ (g\ x)\ (f\ x)\ Empty)\ s \\
filter/4 & filter\ f\ (flatmap\ g\ s) & = flatmap\ (\lambda x.filter\ f\ (g\ x))\ s
\end{array}
$$

In [11] we have already explored some of the above equivalences for sets, namely $fm/2$, $map/2-3$, and $filter/2-4$, and in an extended version of that paper [12] we extended the treatment to bags. In those papers we also explored several equivalences that do not slot into the framework of Figures 1 and 2, for example equivalences governing iteration over more than one collection (we note that our top-level equivalences here, $\phi/1$ and $\phi/2$, are concerned with iteration over a single collection).

6 Optimising comprehensions

We now consider how the equivalences developed above can be used to optimise comprehensions. We recall that the syntax of comprehensions is $[e \mid Q_1;\ \ldots;\ Q_n]$. e is an expression, termed the *head* of the comprehension, and Q_1 to Q_n are *qualifiers*, where $n \geq 0$. Each qualifier is either a *filter* or a *generator*. A filter is a boolean-valued expression. A generator has syntax $p \leftarrow s$, where p is a pattern and s is a collection-valued expression. For the purposes of this paper we assume that all the generators in a comprehension iterate over collections of the same collection type, T. Overall, a comprehension returns a collection of type $T\ t$, where t is the type of its head expression.

Developing optimisation techniques for comprehensions is important because comprehensions provide a unifying query formalism for functional, relational and object-oriented languages. Wadler [18] defines many equivalences for comprehensions. Paton and Gray [10] translate DAPLEX queries into set comprehensions and discuss their optimisation. Trinder [16, 17] gives a translation of the relational calculus into list comprehensions. Buneman *et al.* [4] discuss expressing SQL in the comprehension language CL and review previous work on the expressiveness of the comprehension syntax. Fegaras and Maier [7] define a calculus based on monoid comprehensions and use it to model the essential features of OQL (monoid comprehensions are more general than conventional comprehensions in that they can iterate over multiple collection types). Grust *et al.* [8] discuss optimisation of this calculus and propose a method whereby subqueries expressed in the calculus are translated into algebraic expressions for more efficient evaluation.

Comprehensions add no extra expressiveness to our language and they translate into the syntax of Section 1.1 as follows [2]:

$$\begin{array}{ll}
[e \mid p \leftarrow s; Q] & \equiv flatmap_T\ (\lambda p.[e \mid Q])\ s \\
[e \mid e'; Q] & \equiv if\ e'\ then\ [e \mid Q]\ else\ Empty_T \\
[e \mid\ ] & \equiv single_T\ e
\end{array}$$

In [11, 12] we explored three categories of equivalences for comprehensions, in the context of our computationally complete DBPL: those for interchanging pairs of qualifiers, for introducing/eliminating a qualifier, and for moving qualifiers into/out of nested comprehensions. Some of these equivalences could slot into the framework of Figures 1 and 2 while others do not, for example those governing iteration over multiple collections. Here, we will be needing two of the equivalences from [11, 12], $c/1$ and $c/2$ below, and also $c/3$ which follows from the fact that $flatmap\ single\ s\ =\ s$:

[2] For the purposes of this paper we assume all patterns p in generators $p \leftarrow s$ are *irrefutable* i.e. consist of tuple constructors and variables only, so that $(\lambda p.[e \mid Q])\ e'$ does return not $Fail$ for any element e' of s. If this were not the case, then the RHS of the first equation would be:

$$flatmap_T\ (\lambda x.if\ ((\lambda p.[e \mid Q])\ x)\ \neq Fail\ then\ ((\lambda p.[e \mid Q])\ x)\ else\ Empty_T)\ s$$

We also note that it is the use of $flatmap_T$ in the translation scheme that restricts comprehensions to iterate over a single collection type. If $flatmap_{T,T'}$ were used this restriction could be relaxed, subject to the restrictions on possible combinations of T and T' stated in Footnote 1. Fegaras and Maier [7] also identify these restrictions on comprehensions over multiple collection types.

$$
\begin{array}{lll}
c/1 & [e \mid Q;\ f;\ g;\ Q'] & = [e \mid Q;\ g;\ f;\ Q'] \\
 & & \quad \text{provided } f\ = \bot \iff g\ =\ \bot \\
c/2 & [e \mid Q;\ p \leftarrow [p \mid Q'];\ f;\ Q''] & = [e \mid Q;\ p \leftarrow [p \mid Q';\ f];\ Q''] \\
 & & \quad \text{provided } FV(f) \subseteq FV(p) \\
c/3 & [p \mid p \leftarrow s] & = s
\end{array}
$$

Several of the equivalences that we have stated in this paper, namely $ag/2$, $ag/4$ and their descendants, can be directly reformulated for comprehensions, with the same associated provisos:

$$
\begin{array}{lll}
ag/2 & f\ [g\ (j\ x)|x \leftarrow s] & = f_1 \circ g\ [y|x \leftarrow s; y \leftarrow j\ x] \\
ag/21 & max\ [max\ xs|xs \leftarrow xss] & = max\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/22 & min\ [min\ xs|xs \leftarrow xss] & = min\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/23 & sum\ [sum\ xs|xs \leftarrow xss] & = sum\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/24 & sum\ [count\ xs|xs \leftarrow xss] & = count\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/25 & some\ id\ [some\ p\ xs|xs \leftarrow xss] & = some\ p\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/26 & all\ id\ [all\ p\ xs|xs \leftarrow xss] & = all\ p\ [x|xs \leftarrow xss; xs \leftarrow xss] \\
ag/4 & g\ [y|x \leftarrow s; y \leftarrow j\ x] & = [y|x \leftarrow g\ s; y \leftarrow j\ x] \\
ag/5 & gc\ aggr\ [y|x \leftarrow s; y \leftarrow j\ x] & = [y|x \leftarrow gc\ aggr\ s; y \leftarrow j\ x] \\
ag/51 & gc\ aggr\ [((a,b),c)|(a,c) \leftarrow r; b \leftarrow s] = & \\
 & [((a,b),c)|(a,c) \leftarrow gc\ aggr\ r; b \leftarrow s] & \\
ag/52 & gc\ aggr\ [(a,b)|(a,b) \leftarrow s; f\ a] & = [(a,b)|(a,b) \leftarrow gc\ aggr\ s; f\ a] \\
ag/53 & gc\ aggr\ [((a,b),c)|(b,c) \leftarrow r; (a,b') \leftarrow s; b = b'] = & \\
 & [((a,b),c)|(b,c) \leftarrow gc\ aggr\ r; (a,b') \leftarrow s; b = b'] &
\end{array}
$$

We illustrate the practical use of some of these equivalences by means of an example query over a database containing information about music sales. The following extensionally-defined collections are maintained in the database:

$$
\begin{array}{l}
categories :: Bag\ (Category, Group) \\
recordings :: Bag\ (RecNo, Group, Margin, Sales)
\end{array}
$$

These detail for each group with whom the company has a contract the category (e.g. Folk, Rock, Jazz) of the group, and for each recording a recording number, the group that made the recording, the profit margin and the total sales.

Our example query involves finding the total sales per category and group, for those categories which satisfy a stated predicate, p. A naive formulation of this query is to project the group and sales attributes from *recordings*, join the result with *categories*, select the tuples which satisfy p, and then perform a *gc sum*:

$$
\begin{array}{l}
let\ r1\ =\ [(g,s) \mid (r,g,m,s)\ \leftarrow\ recordings]; \\
\quad r2\ =\ [((c,g),s) \mid (g,s)\ \leftarrow\ r1;\ (c,g')\ \leftarrow\ categories;\ g = g'] \\
in\ \ gc\ sum\ [((c,g),s) \mid ((c,g),s)\ \leftarrow\ r2;\ p\ c]
\end{array}
$$

Using $ag/52$ we can push *gc sum* before the selection $p\ c$:

$$\begin{aligned} let\, r1 &= [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ r2 &= [((c,g),s) \mid (g,s) \leftarrow r1;\ (c,g') \leftarrow categories;\ g = g'] \\ in\ &[((c,g),s) \mid ((c,g),s) \leftarrow gc\ sum\ r2;\ p\ c] \end{aligned}$$

Using $ag/53$ we can then push $gc\ sum$ into the join, to obtain an expression which projects the group and sales attributes from *recordings*, computes for each group their total sales, joins the result with *categories*, and finally selects the tuples which satisfy p:

$$\begin{aligned} let\, r1 &= [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ r2 &= [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ g = g'] \\ in\ &[((c,g),s) \mid ((c,g),s) \leftarrow r2;\ p\ c] \end{aligned}$$

Using $c/2$ and then $c/1$ we can push the filter $(p\ c)$ into the second comprehension and then promote it past $g = g'$:

$$\begin{aligned} let\, r1 &= [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ r2 &= [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ p\ c;\ g = g'] \\ in\ &[((c,g),s) \mid ((c,g),s) \leftarrow r2] \end{aligned}$$

Finally, using $c/3$ we can eliminate the third comprehension:

$$\begin{aligned} let\, r1 &= [(g,s) \mid (r,g,m,s) \leftarrow recordings]; \\ r2 &= [((c,g),s) \mid (g,s) \leftarrow gc\ sum\ r1;\ (c,g') \leftarrow categories;\ p\ c;\ g = g'] \\ in\ &r2 \end{aligned}$$

7 Related Work

Sheard and Fegaras [14] investigate the optimisation of a fold operator over types that are sums-of-products (e.g. lists and trees) drawing together and generalising previous work in this area in the context of functional programming languages. They state and prove a *Fold Promotion Theorem* which is the analogue of our equivalence $\phi/3$.

Steenhagen *et al.* [15] use the following instance of $fm/2$ to replace the nested selection expression on the RHS by the *nestjoin* expression on the LHS:

$$\begin{aligned} &[f\ x \mid (x, ys) \leftarrow [(x, [g\ x\ y \mid y \leftarrow s';\ q\ x\ y]) \mid x \leftarrow s];\ p\ x\ ys] \ = \\ &[f\ x \mid x \leftarrow s;\ p\ x\ [g\ x\ y \mid y \leftarrow s';\ q\ x\ y]] \end{aligned}$$

Chaudhuri and Shim [6] investigate the optimisation of queries containing group-and-compute operations. They give three equivalences: $ag/53$, $eq/1$ below which introduces/eliminates a redundant group-and-compute operation, and $eq/2$ below which allows a group of tuples to be coalesced into a single tuple together with a count of the number of tuples which were coalesced:

$$\begin{aligned} eq/1 \quad & gc\ aggr\ (flatmap\ f\ s) \ = \ gc\ aggr\ (flatmap\ f\ (gc\ aggr\ s)) \\ eq/2 \quad & gc\ aggr\ [(a,(n,c)) \mid (a,n) \leftarrow gc\ count\ s;\ (a',c) \leftarrow s';\ a = a'] \ = \\ & gc\ aggr'\ [(a,c) \mid (a,b) \leftarrow s;\ (a',c) \leftarrow s';\ a = a'] \end{aligned}$$

$eq/1$ has $aggr = \phi\ id\ op\ e$ and holds provided $(gc\ aggr) \circ (flatmap\ f)$ distributes over $h \circ merge'$, where $h = map\ (\lambda(x, ys).(x, aggr\ ys))$. Superficially, $eq/1$ would seem to be captured by applying $ag/3$ followed by $ag/5$ to the left-hand-side. However, the latter step generates the proviso $(gc\ aggr) \circ f = f$, which is not needed due to the retention of the outer $gc\ aggr$ operation on the right-hand-side. In fact, $eq/1$ can be accommodated by a third top-level equivalence governing the replacement of three applications of ϕ by two. $eq/2$ has $aggr = \phi\ f\ op\ e$ and $aggr' = \phi\ f'\ op\ e$, and holds provided $aggr'\ (replicate\ n\ x) = aggr\ (single\ (n, x))$, where $replicate\ n\ x$ returns a collection containing n instances of x. This too cannot be derived from $\phi/1$ and $\phi/2$ but would be accommodated by the same top-level equivalence as required for $eq/1$.

Gupta *et al.* [9] examine the optimisation of SQL query trees built from selection, product and 'generalised projection' nodes. They consider aggregation where the aggregate over a set s can be computed from aggregates over subsets of s and identify $ag/21-3$. Other equivalences they develop allow a group-and-compute to be promoted through a product and a selection i.e. $ag/51$ and $ag/52$. A further equivalence they give governs the removal of redundant group-by operations:

$$gc\ aggr\ [(a, b) \mid (c, d) \leftarrow gc\ aggr\ s] = gc\ aggr\ [(a, b) \mid (c, d) \leftarrow s]$$

provided $FV(a, b) \subseteq FV(c)$ and $\oplus$ is idempotent. This can be shown to be an instance of $eq/1$ above.

Yan and Larson [19] examine the optimisation of decomposable aggregation functions over bags, where an aggregation function f is *decomposable* if there exist functions f_1 and f_2 such that $f\ (s_1 \oplus s_2) = f_2\ (f_1\ s_1)\ (f_1\ s_2)$. They identify *sum*, *max*, *min* and *count* as decomposable and use combinations of $eq/1$ and $eq/2$ to push/pull group-and-compute operations through one or both branches of a join.

8 Conclusion

We have defined a 'fold' operator ϕ over collection types such as lists, bags and sets, and have investigated the optimisation of aggregation functions defined in terms of ϕ. We defined two equivalences for ϕ. The first, $\phi/1$, governs the commutativity of ϕ, and the second, $\phi/2$, governs the coalescing of two applications of ϕ into one. Many of the equivalences proposed by other researchers turn out to be special cases of $\phi/1$ and $\phi/2$. Conversely, we have discovered new equivalences such as $ag/25$, $ag/26$ and $part/1-3$, and we conjecture that many other new equivalences based upon $\phi/1$ and $\phi/2$ remain to be discovered. We are currently exploring equivalences such as $eq/1$ and $eq/2$ above that can be derived from a third top-level equivalence which governs the replacement of three applications of ϕ by two.

We see our work as having application in two main areas. Firstly, it provides a rich, yet simple, theoretical framework for exploring and formally proving new equivalences for aggregation functions. Secondly, the equivalences that we derive for our functional DBPL clearly also hold for less expressive sub-languages, for example the monoid comprehension calculus, OQL and SRU, and could thus be used in practical query optimisers for such languages, as well as for our own DBPL.

Acknowledgements

This work has been supported by the U.K. Engineering and Physical Sciences Research Council (grant no. GR/L 26872). We thank the referees for their helpful comments on earlier versions of this paper.

Appendix - Proofs of the key equivalences

We give proofs of $\phi/1, \phi/2, \phi/3, ag/1$ and $ag/2$. The remaining equivalences of Figures 1 and 2 follow straight-forwardly from their parent equivalence.

(a) Proof of $\phi/1$

$\phi/1$ states that for all collections s, $g\ (h\ (s)) \ = \ h\ (g\ (s))$, where $g \ = \ \phi\ f\ op\ e$ and $h \ = \ \phi\ f'\ op'\ e'$, provided that:

(1) $g\ e' \ = \ h\ e$,
(2) $g \circ f' \ = \ h \circ f$,
(3) $g\ ((h\ s_1)\ op'\ (h\ s_2)) \ = \ h\ ((g\ s_1)\ op\ (g\ s_2))$.

We show this equivalence by structural induction over s (this is a valid proof technique even for infinite s because all functions in our language are continuous — see [13]):
$\underline{\text{If } s = Empty}$ then

$LHS\ Empty = g\ (h\ Empty) = g\ e'$
$RHS\ Empty = h\ (g\ Empty) = h\ e$

and these are equal by (1).
$\underline{\text{If } s = \perp}$ then $LHS \perp \ = \ RHS \perp \ = \ \perp$ by $coll/1$.
$\underline{\text{If } s = single\ a}$ for some expression a, then

$LHS\ s = g\ (h\ (single\ a)) = g\ (f'\ a)$
$RHS\ s = h\ (g\ (single\ a)) = h\ (f\ a)$

and these are equal by (2).

If $s = s_1 \oplus s_2$ then

$$\begin{aligned} LHS\ s &= g\ (h\ (s_1 \oplus s_2)) \\ &= g\ ((h\ s_1)\ op'\ (h\ s_2)) \quad \text{by } coll/4 \\ &= h\ ((g\ s_1)\ op\ (g\ s_2)) \quad \text{by } (3) \\ &= h\ ((g\ (s_1 \oplus s_2)) \quad \text{by } coll/4 \\ &= RHS\ s \end{aligned}$$

(b) Proof of $\phi/2$

$\phi/2$ states that $\phi\ f\ op\ e\ (\phi\ f'\ op'\ e'\ s) \ = \ \phi\ ((\phi\ f\ op\ e) \circ f')\ op''\ (\phi\ f\ op\ e\ e')\ s$ provided that:

(1) $(\phi\ f\ op\ e\ s)\ op''\ (\phi\ f\ op\ e\ \ s') \ = \ \phi\ f\ op\ e\ (s\ op'\ s')$

We show this equivalence by structural induction over s:
If $s = Empty$ then $LHS\ Empty \ = \ RHS\ Empty \ = \ \phi\ f\ op\ e\ e'$.
If $s = \bot$ then $LHS\ \bot \ = \ RHS\ \bot \ = \ \bot$ by $coll/1$.
If $s = single\ a$ for some expression a, then

$$\begin{aligned} LHS\ s &= \phi\ f\ op\ e\ (\phi\ f'\ op'\ e'\ (single\ a)) \\ &= \phi\ f\ op\ e\ (f'\ a) \\ &= RHS\ s \end{aligned}$$

If $s = s_1 \oplus s_2$ then

$$\begin{aligned} LHS\ s &= \phi\ f\ op\ e\ (\phi\ f'\ op'\ e'\ (s_1 \oplus s_2)) \\ &= \phi\ f\ op\ e\ ((\phi\ f'\ op'\ e'\ s_1)\ op'\ (\phi\ f'\ op'\ e'\ s_2)) \quad \text{by } coll/4 \end{aligned}$$

while

$$\begin{aligned} RHS\ s &= \phi\ ((\phi\ f\ op\ e) \circ f')\ op''\ (\phi\ f\ op\ e\ e')\ (s_1 \oplus s_2) \\ &= (\phi\ ((\phi\ f\ op\ e) \circ f')\ op''\ (\phi\ f\ op\ e\ e')\ s_1))\ op'' \\ &\quad\ (\phi\ ((\phi\ f\ op\ e) \circ f')\ op''\ (\phi\ f\ op\ e\ e')\ s_2)) \quad \text{by } coll/4 \\ &= (\phi\ f\ op\ e\ ((\phi\ f'\ op'\ e'\ s_1))\ op'' \\ &\quad\ (\phi\ f\ op\ e\ ((\phi\ f'\ op'\ e'\ s_2)) \quad \text{by the induction hypothesis} \\ &= LHS\ s \quad \text{by } (1) \end{aligned}$$

(c) Proof of $\phi/3$

Substituting op' for op'' in $\phi/2$ transforms the proviso stated for $\phi/2$ into the proviso stated for $\phi/3$.

(d) Proof of $ag/1$

The equivalence is easily seen to be an instance of $\phi/2$: substitute $\phi\ f\ op\ e$ in $\phi/2$ by h, $\phi\ f'\ op'\ e'$ by $group$ and op'' by $h \circ merge'$. It therefore only remains to show that the proviso holds i.e. that $h\ (merge'\ (h\ s)\ (h\ s'))\ =\ h\ (merge\ s\ s')$ or, substituting for the definition of $merge'$, that

$$h\ (merge\ (flatmap\ nest\ (h\ s))\ (flatmap\ nest\ (h\ s'))) = h\ (merge\ s\ s')$$

Now every tuple (x, ys) of s or s' results in a tuple $(x, aggr\ ys)$ when h is applied to s and s' in the LHS of this equivalence. This tuple in turn results in a tuple $(x, single\ (aggr\ ys))$ when $flatmap\ nest$ is applied to $h\ s$ and $h\ s'$. Since $merge$ depends on the first components of tuples only, the collections $merge\ s\ s'$ and $merge\ (flatmap\ nest\ (h\ s))\ (flatmap\ nest\ (h\ s'))$ are thus identical, except that every tuple (x, ys) in the former corresponds to a tuple $(x, single\ (aggr\ ys))$ in the latter. Applying h to these two collections transforms (x, ys) to $(x, aggr\ ys)$ and $(x, single\ (aggr\ ys))$ to $(x, aggr\ (single\ (aggr\ ys)))$. Since first argument to ϕ in the definition of $aggr$ is id, $aggr \circ single = id$. The second tuple thus simplifies to $(x, aggr\ ys)$, and the two tuples are identical.

(e) Proof of $ag/2$

$ag/2$ states that $f\ (map\ (g \circ j)\ s)\ =\ f_1 \circ g\ (flatmap\ j\ s)$, where $f\ =\ \phi\ f_1\ op\ (f_1\ e)$ and $g\ =\ \phi\ f_2\ op\ e$, provided that:

(1) f_1 is strict.
(2) f_1 distributes over op.

To show this, we first apply $\phi/2$ to the LHS of $ag/2$, setting op'' in $\phi/2$ to be op above, and obtaining:

$$\phi\ (f_1 \circ g \circ j)\ op\ (f_1\ e)\ s$$

We need to check that the proviso to $\phi/2$ holds i.e. that $(f\ s)\ op\ (f\ s')\ =\ f\ (s \oplus s')$, which is indeed the case, by $coll/4$.

We next apply $\phi/2$ to the RHS of $ag/2$, again setting op'' in $\phi/2$ to be op above, and obtaining:

$$f_1(\phi\ (g \circ j)\ op\ e\ s)$$

We again need to check that the proviso to $\phi/2$ holds i.e. that $(g\ s)\ op\ (g\ s')\ =\ g\ (s \oplus s')$, which again is the case, by $coll/4$.

$ag/2$ has thus been simplified to

$$\phi\ (f_1 \circ g \circ j)\ op\ (f_1\ e)\ s\ =\ f_1(\phi\ (g \circ j)\ op\ e\ s)$$

This can now be shown by structural induction on s:
$\underline{\text{If } s = Empty}$ then $LHS\ Empty\ =\ RHS\ Empty\ =\ f_1\ e$.
$\underline{\text{If } s = \bot}$ then $LHS\ \bot\ =\ \bot$ and $RHS\ \bot\ =\ f_1\ \bot\ =\ \bot$ by (1).
$\underline{\text{If } s = single\ a}$ then $LHS\ s\ =\ RHS\ s\ =\ f_1(g(j\ a))$.
$\underline{\text{If } s = s_1 \oplus s_2}$ then

$$
\begin{aligned}
LHS\ s &= \phi\ (f_1 \circ g \circ j)\ op\ (f_1\ e)\ (s_1 \oplus s_2)) \\
&= (\phi\ (f_1 \circ g \circ j)\ op\ (f_1\ e)\ s_1)\ op\ (\phi\ (f_1 \circ g \circ j)\ op\ (f_1\ e)\ s_2) \\
&= (f_1(\phi\ (g \circ j)\ op\ e\ s_1))\ op\ (f_1(\phi\ (g \circ j)\ op\ e\ s_2)) \\
&\quad \text{by the induction hypothesis} \\
&= f_1((\phi\ (g \circ j)\ op\ e\ s_1)\ op\ (\phi\ (g \circ j)\ op\ e\ s_2)) \quad \text{by (2)} \\
&= f_1(\phi\ (g \circ j)\ op\ e\ (s_1 \oplus s_2)) \\
&= RHS\ s
\end{aligned}
$$

References

1. F.Bancilhon, T.Briggs, S.Khoshafian and P.Valduriez, FAD, a powerful and simple database language, *Proc. 13th VLDB Conference*, Brighton, September 1987, pp 97-106.
2. C.Beeri and Y.Kornatzky, Algebraic optimization of object-oriented query languages, *Proc. 3rd ICDT*, Paris, December 1990, pp 72-88. Springer-Verlag LNCS 470.
3. R.Bird and P.Wadler, *Introduction to functional programming*, Prentice-Hall, 1988.
4. P.Buneman, L.Libkin, D.Suciu, V.Tannen and L.Wong, Comprehension Syntax, *ACM SIGMOD Record*, 23(1), pp 87-96, 1994.
5. P.Buneman, S.Naqvi,V.Tannen and L.Wong, Principles of programming with complex objects and collection types, *Theoretical Computer Science*, 149(1), pp 3-48, 1995.
6. S.Chaudhuri and K.Shim, Including group-by in query optimisation, *Proc. 20th VLDB Conference*, Santiago, September 1994, pp 354-366.
7. L.Fegaras and D.Maier, Towards an effective calculus for Object Query Languages, *Proc. ACM SIGMOD Conference*, May 1995, pp 47-58.
8. T.Grust, J.Kroger, D.Gluche, A.Heuer and M.Scholl, Query evaluation in CROQUE - Calculus and Algebra coincide, *Proc. 15th British National Conference on Databases*, London, July 1997, pp 84-100. Springer-Verlag LNCS 1271.
9. A.Gupta, V.Harinarayan and D.Quass, Aggregate query-processing in Data Warehousing Environments, *Proc. 21st VLDB Conference*, Zurich, 1995, pp 358-369.
10. N.W.Paton and P.M.D.Gray, Optimising and executing DAPLEX queries using Prolog, *The Computer Journal*, 33(6), pp 547-555, 1990.
11. A.Poulovassilis and C.Small, Investigation of algebraic query optimisation for database programming languages, *Proc. 20th VLDB Conference*, Santiago, 1994, pp 415-426.
12. A.Poulovassilis and C.Small, Algebraic query optimisation for database programming languages, *The VLDB Journal*, 5(2), pp 119-132, 1996.
13. D.A.Schmidt, *Denotational Semantics*, Allyn and Bacon, 1986.
14. T.Sheard and L.Fegaras, A fold for all seasons. *Proc. ACM Conference on Functional Programming and Computer Architectures (FPCA)*, 1993, pp 233-242.
15. H.J.Steenhagen, P.M.G.Apers, H.M.Blanken and R.A.de By, From nested-loop to join queries in OODB, *Proc. 20th VLDB Conference*, Santiago, 1994, pp 618-629.
16. P.Trinder, *A functional database*, D.Phil Thesis, Oxford University, 1989.
17. P.Trinder, Comprehensions: a query notation for DBPLs. *Proc. 3rd Int. Workshop on Database Programming Languages*, Nafplion, August 1991, pp 55-68.

18. P.Wadler, Comprehending Monads, *Proc. ACM Conference on Lisp and Functional Programming*, Nice, June 1990, pp 61-78.
19. W.Yan and P.Larson, Eager aggregation and lazy aggregation, *Proc. 21st VLDB Conference*, Zurich, 1995, pp 345-357.