A Uniform Calculus for Collection Types

Leonidas Fegaras
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
email: *fegaras@cse.ogi.edu*

Abstract

We present a new algebra for collection types based on monoids and monoid homomorphisms. The types supported in this algebra can be any nested composition of collection types, including lists, sets, multisets (bags), vectors, and matrices. We also define a new calculus for this algebra, called *monoid comprehensions*, that captures operations involving multiple collection types in declarative form. This algebra can easily capture the semantics of many object-oriented database query languages that support mixed collection types, such as the OQL language of the ODMG-93 standard. In addition, it is ideal for expressing data parallelism and nested parallelism and can be effectively translated onto many parallel architectures.

We present a normalization algorithm that reduces any expression in our algebra to a canonical form which, when evaluated, generates very few intermediate data structures. These canonical forms are amenable to a higher degree of parallelism than the original programs. The normalization algorithm generalizes some well-known algebraic optimization techniques and heuristics used in relational query optimization. Finally, we extend this algebra by incorporating object-oriented features, such as object identity and destructive updates.

1 Introduction

1.1 Motivation

New research directions in database programming languages are guided by the desire to support multiple collection types in a single framework. It is essential for such a framework to treat collections uniformly in a simple and extensible language. The primary interest of these approaches is the bulk manipulation of collection types. Bulk operations not only need to have enough expressive power to perform complex computations, but they need to be effectively translated into efficient execution plans as well.

There are many proposals for complex object algebras that are aimed at a universal theory of collection types. We classify these approaches into two categories: algebras dealing with *stream forms* and algebras dealing with *decomposed forms* of data. An operator imposes a stream view of a collection if it processes the collection one element at a time. The stream form has been very effective in evaluating set operations in relational databases and it is ideal for expressing pipeline processing. An operator imposes a decomposed view of a collection if it divides the collection into a number of subcollections that are processed independently. Data decomposition supports divide-and-conquer computations, which allow independent parallelism.

Higher-order generic operators, such as map and filter, are an effective way for capturing traversals over collection structures. Each such operator characterizes the control structure of an algorithm, while the instantiations of its functional parameters characterize the "policy" that elaborates the algorithm. By abstracting the policy details from an algorithm, we can express laws about all the algorithms with the same control structure. These laws may not be apparent when we examine these algorithms in their fully instantiated form. By restricting the set of allowable operators to a small, but powerful, set of higher-order operators, we reduce the number of transformation rules needed for optimizing expressions that involve these operators. This restriction results in a system whose correctness is easier to ascertain.

This paper presents a query algebra that includes only one higher-order generic operator, namely the *monoid homomorphism*, which processes collections in decomposed form. In contrast to other approaches [3, 5], we are not so much concerned about the expressive power and the complexity of our language (even though it can be easily proved that our algebra is more expressive than NF^2, the nested relational algebra). Instead, this operator was introduced to provide a precise semantics to the algebra, in such a way that all invalid programs can be easily detected at compile time. Moreover, we wanted an algebra that captures the semantics of most object-oriented database languages, especially the Object Query Language (OQL) of the ODMG-93 standard [8]. Finally, and more importantly, we wanted an algebra that facilitates query optimization and processing. Even though our algebra cannot express non-polynomial time operations, such as powerset and transitive closure, it can easily capture NF^2-style operations that involve different collection types in a uniform way.

1.2 Overview of the Calculus

We have presented previously an expressive and extensible algebra based on *fold* operations [9, 17, 10] that processes data in stream form. Given any user-defined data type expressed by inductive type equations, this system is able to synthesize the definition of the fold operator for this type by simply examining the type details. The fold operator follows a pattern of recursion similar to the pattern of recursion in the type definition. Furthermore, this system is able to generate the necessary transformation theory to optimize expressions that involve the synthesized operator. For example, the *list* type constructor is defined by the following inductive equation:

$$\text{list}(\alpha) \;=\; \text{Nil} \mid \text{Cons}\ \textbf{of}\ \alpha \times \text{list}(\alpha)$$

where α is a type parameter and Nil and Cons are value constructors. The fold operator for lists is defined as follows:

$$\begin{array}{lcl} \mathrm{fold}^{list}(e, acc)\,\mathrm{Nil} & = & e \\ \mathrm{fold}^{list}(e, acc)\,(\mathrm{Cons}(a, x)) & = & acc(a, \mathrm{fold}^{list}(e, acc)\,x) \end{array}$$

Value e in $\mathrm{fold}^{list}(e, acc)$ is the initial seed of the accumulation and acc is a binary function that accumulates each list element to the seed. That is, when $\mathrm{fold}^{list}(e, acc)$ is applied to the list $[a_1, \ldots, a_n]$, it computes the value

$$acc(a_1, acc(a_2, \ldots, acc(a_n, e)))$$

This expression can be evaluated by an incremental computation that captures a *pipeline*: the list elements form a stream of values that flows through the pipe (starting from the last element a_n and proceeding to the first element a_1). For example,

$$\mathrm{append}(x, y) \;=\; \mathrm{fold}^{list}(y, \lambda(a, r).\mathrm{Cons}(a, r))\,x$$

That is, if $x = [a_1, \ldots, a_n]$, then $\mathrm{append}(x, y)$ is

$$\mathrm{Cons}(a_1, \mathrm{Cons}(a_2, \ldots, \mathrm{Cons}(a_n, y)))$$

Pipelining is an effective evaluation technique for stream forms of data. Similarly, *independent parallelism* [15] is effective for decomposed forms of data and for divide-and-conquer computations. One way to capture such computations is to use *structural recursion* [3, 5, 4]. For example, an alternative construction of list values uses three primitives: nil $[\,]$, singleton $[a]$ for an element a, and list append. That way lists are processed in decomposed form. The list structural recursion is defined as follows:

$$\begin{array}{lcl} \mathrm{Hom}^{list}(e, f, acc)\,[\,] & = & e \\ \mathrm{Hom}^{list}(e, f, acc)\,[a] & = & f(a) \\ \mathrm{Hom}^{list}(e, f, acc)\,(\mathrm{append}(x_1, x_2)) & = & acc(\mathrm{Hom}^{list}(e, f, acc)\,x_1, \mathrm{Hom}^{list}(e, f, acc)\,x_2) \end{array}$$

where acc is an associative function (i.e., $\forall x, y, z : acc(acc(x, y), z) = acc(x, acc(y, z))$) with identity e (i.e., $\forall x : acc(e, x) = acc(x, e) = x$). Let T be the output type of the operator $\mathrm{Hom}^{list}(e, f, acc)$. This operator constructs values of type T in the same way as list values are constructed, but with e substituted for $[\,]$, $f(a)$ for $[a]$, and acc for append. In categorical terms, we say that both $(\mathrm{list}, [\,], \mathrm{append})$ and (T, e, acc) are *monoids* (because of the associativity and identity laws), and that Hom^{list} is a *monoid homomorphism* from the former to the latter monoid. This operation captures a divide-and-conquer computation since any list $x = [a_1, \ldots, a_n]$ can be divided into two lists x_1 and x_2 such that $x = \mathrm{append}(x_1, x_2)$. In that case, operation $\mathrm{Hom}^{list}(e, f, acc)\,x$ is equal to $acc(\mathrm{Hom}^{list}(e, f, acc)\,x_1, \mathrm{Hom}^{list}(e, f, acc)\,x_2)$. These two recursive calls can be evaluated in parallel. Since acc is associative, it does not matter how we split the computations into parallel processes and how we accumulate the result values, as long as we accumulate them from left to right. If in addition acc is commutative, as in the case of all set operations, then the results of the computation can be accumulated in any fashion.

One computation of $\mathrm{Hom}^{list}(e, f, acc)\, x$ is $\mathrm{fold}^{list}(e, \lambda(a, r).acc(f(a), r))\, x$, which means that homomorphisms can form pipelines too. But there are better ways of computing homomorphisms. For example, in a Connection Machine [12], we can feed the list $x = [a_1, \ldots, a_n]$ into n processors $X[1], \ldots, X[n]$ and evaluate the program

for all k **in parallel do**
  if $k > n$ **then** $X[k] \leftarrow e$ **else** $X[k] \leftarrow f(X[k])$;
for $i \leftarrow 1$ **to** $\lceil \log_2 n \rceil$ **do**
  for all k **in parallel do**
    if $(k-1) \bmod 2^i = 0$ **then** $X[k] \leftarrow acc(X[k], X[k+2^{i-1}])$;

This computation evaluates $\mathrm{Hom}^{list}(e, f, acc)\, x$ as a balanced binary tree. The tree leaves are the values $f(a_i)$ and the internal nodes are applications of acc. Since the depth of the tree is $\lceil \log_2 n \rceil$ and all computations at each level are performed in parallel, the entire computation is performed in $\lceil \log_2 n \rceil + 1$ steps. This computation is a data parallel computation [2, 18, 1] since each value a_i of x is fed into a different processor $X[i]$. If the number of available processors is less than n, then each processor is assigned more than one element and the homomorphism is evaluated partially as a pipeline and partially as a data parallel computation [26]. If in addition function f is another homomorphism, then we will have a nested parallelism [1].

The focus of this paper is object-oriented database languages based on divide-and-conquer computations. We provide an extensible library of monoids that capture many collection types, such as strings, lists, sets, multisets, vectors, and matrices. In addition, this library includes monoids that capture basic types, such as integers and booleans. A monoid homomorphism in our algebra is from one monoid to another, where both monoids are defined in this library. For example, the homomorphism that maps a list into a set is

$$\mathrm{hom}[list, set](f)\, x \;=\; \mathrm{Hom}^{list}(\emptyset, f, \cup)\, x$$

That is, if x is the list $[a_1, \ldots, a_n]$, then $\mathrm{hom}[list, set](f)\, x$ computes the set $f(a_1) \cup \cdots \cup f(a_n)$.

For example, the following algebraic operation joins the list x with the multiset y and returns the set of all pairs (r, s) that satisfy $r.A = s.A$, where r is an element of x and s is an element of y:

$$\mathrm{hom}[list, set](\lambda r.\, \mathrm{hom}[bag, set](\lambda s.\, \textbf{if } r.A = s.A \textbf{ then } \{(r, s)\} \textbf{ else } \{\})\, y)\, x$$

Another example is the length of a list x computed by:

$$\mathrm{hom}[list, sum](\lambda r.1)\, x$$

where sum is the integer monoid whose merge function is $+$ and whose zero element is 0. That is, if $x = [a_1, \ldots, a_n]$, then this expression computes $1 + \cdots + 1 = n$, which is the length of x. That is, we provide a uniform algebra in which we can express operations that involve different collections as well as aggregates and predicates. This uniform treatment of bulk operators allows us to perform algebraic optimizations that involve complex predicates and

aggregates, such as the translation of a nested SQL query into a join. This translation would not be as straight-forward if we had chosen an algebra in which predicates were treated differently from the other bulk operations.

On top of this algebra we define our calculus, which is based on *monoid comprehensions*. For example, one valid expression in the calculus is

$$set\{\,(a,b)\,|\,a \leftarrow [1,2,3],\ b \leftarrow \{\!\{4,5\}\!\}\,\}$$

This expression joins the list $[1,2,3]$ with the multiset $\{\!\{4,5\}\!\}$ and returns the following set (it is a set because the comprehension is tagged by the word *set*):

$$\{(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)\}$$

Another example is

$$sum\{\,a\,|\,a \leftarrow [1,2,3],\ a \geq 2\,\}$$

which returns 5, the sum of all list elements greater than or equal to 2.

Our calculus is equivalent to our algebra. It can serve as a database language since it does not require higher-order functions, which usually make programs hard to read. Programs expressed in our calculus are far easier to understand than the equivalent algebraic forms. Even though all the program transformation algorithms described in this paper are applied to algebraic forms only, most of our examples are expressed in the calculus.

1.3 Translating Object-Oriented Queries into the Calculus

The monoid comprehension calculus is an ideal calculus for many object-oriented query languages that support multiple collection types. Even though the tuple relational calculus and the relational algebra do not capture commercial relational languages very well (since they do not support aggregates, grouping, and sorting) our calculus and algebra capture precisely the semantics of most database languages. For example, the Object Query Language (OQL) of the ODMG-93 standard [8] can be translated directly into the monoid comprehension calculus. Consider for example the following ODMG-93 schema:

$$\begin{array}{ll}\text{Cities}: & set(\langle \text{name}: string, \\ & \quad \text{hotels}: bag(\langle \text{name}: string, \text{rooms}: list(\langle \text{bed\#}: int, \ldots\rangle), \ldots\rangle), \\ & \quad \text{places\_to\_visit}: list(\langle \text{name}: string, \ldots\rangle)\rangle)\end{array}$$

where '...' represents some omitted details. The following OQL query finds all hotels in Portland which are also interesting places to visit:

```
select distinct h.name
  from c in Cities,
       h in c.hotels,
       p in c.places_to_visit
  where c.name = "Portland" and h.name = p.name
```

Notice that variable h ranges over c.hotels, where c ranges over Cities. Since Cities is a set of tuples where each tuple has an attribute *hotels* which is a bag of hotels, then we have the nested collection of a set of bags. The output of this query returns a set of hotel names. Therefore, this query implicitly flattens this set-of-bags. Notice also that c ranges over a set, h ranges over a bag, and p ranges over a list, while the output of the query is a set. If all collection types in the schema were sets, then this query could be expressed in NF^2-style algebraic form as follows:

$$\begin{array}{l}\textbf{map}(\textbf{flatten}(\textbf{map}(\textbf{filter}(\text{Cities}, \lambda c.\, c.\text{name} = \text{“Portland”}),\\ \qquad\qquad \lambda c.\, \textbf{join}(c.\text{hotels}, c.\text{places\_to\_visit},\\ \qquad\qquad\qquad \lambda(h,p).\, h.\text{name} = p.\text{name}))),\\ \quad \lambda(h,p).\, h.\text{name})\end{array}$$

Even though there are ways of translating SQL-like queries over nested sets into NF^2 form (as is done in the SHORE project [21]), these approaches become infeasible when multiple collection types are introduced. First, the number of required algebraic operators increases dramatically. For example, if we had n collection types we would need n^3 different join operators, since we may need to join two different collection types and return another collection type. Second, we will need n^2 coercion operators to map one collection type to another. Finally, increasing the number of operators will increase the number of transformation rules required for optimizing queries.

Consider now how the above OQL query is expressed in our monoid comprehension syntax:

$$\begin{array}{l}set\{\, h.\text{name} \mid c \leftarrow \text{Cities},\ h \leftarrow c.\text{hotels},\ p \leftarrow c.\text{places\_to\_visit},\\ \qquad c.\text{name} = \text{“Portland”},\ h.\text{name} = p.\text{name}\,\}\end{array}$$

which is directly translated into the following algebraic form:

$$\begin{array}{l}\text{hom}[set, set](\lambda c.\text{hom}[bag, set](\lambda h.\text{hom}[list, set](\lambda p.\ \textbf{if}\ c.\text{name} = \text{“Portland”}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{and}\ h.\text{name} = p.\text{name}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \{h.\text{name}\}\ \textbf{else}\ \{\})\\ \qquad\qquad\qquad\qquad\qquad\qquad c.\text{places\_to\_visit})\\ \qquad\qquad\qquad\qquad c.\text{hotels})\\ \qquad\qquad \text{Cities}\end{array}$$

Even though this form resembles a nested-loop join (if it were to be evaluated naively), we will see that in fact there are ways of translating such forms into efficient joins, such as merge joins and hash joins.

Consider now the following OQL query that finds all hotels in Portland that have at least one room with three beds:

```
select h.name
  from hl in (select c.hotels
                from c in Cities
                where c.name = "Portland"),
        h in hl
  where exists r in h.rooms : (r.bed# = 3)
```

This query is expressed in the comprehension syntax as:

$$bag\{\, h.\text{name} \mid hl \leftarrow bag\{\, c.\text{hotels} \mid c \leftarrow \text{Cities},\ c.\text{name} = \text{“Portland”}\,\},\\ h \leftarrow hl,\ some\{\, r.\text{bed\#} = 3 \mid r \leftarrow h.\text{rooms}\,\}\,\}$$

where *some* is the monoid with *false* as the zero element and $\vee$ (the boolean disjunction operator) as the merge function. That is, if $h.\text{rooms} = [r_1, \ldots, r_n]$, then

$$some\{\, r.bed\# = 3 \mid r \leftarrow h.\text{rooms}\,\} \;=\; (r_1.bed\# = 3) \vee \cdots \vee (r_n.bed\# = 3)$$

We will see that by adopting this uniform syntax for joins, predicates, and aggregates, we can easily optimize nested OQL queries, in which the restriction predicate of a select-from-where query contains another select-from-where query.

The monoid calculus can be easily extended to capture object identity. In fact, object identity is just another monoid. For example, one valid object-oriented comprehension is

$$list\{\, !x \mid x \leftarrow [\text{new}(1), \text{new}(2)],\ x := !x + 1\,\}$$

which first creates a list of two new objects (new(1) and new(2)) of type $list(obj(int))$. Variable x ranges over this list (i.e., x is of type $obj(int)$) and the state of x is incremented by one (by $x := !x + 1$). The result of this computation is the list $[2, 3]$. An object-oriented comprehension is translated into a *state transformer* that will propagate the object heap (which contains bindings from object identities to object states) through all operations in an expression, and will change it only when there is an operation that creates a new object or modifies the state of an object. This translation captures precisely the semantics of object identity without the need of extending the base model. It also provides an equational theory that allows us to do valid optimizations for object-oriented queries. The same technique is used for optimizing destructive updates.

In [24] a normalization algorithm was presented for reducing monad comprehensions to a canonical form. A nice property of canonical forms is that their evaluation generates very few intermediate data structures that regularly need to be garbage-collected after they are used. Following this work, Section 3 presents a normalization algorithm for monoid comprehensions. While there are many algebras with operators that can work on mixed collections, such as general monoid homomorphisms [3, 5, 4] and folds [9], our calculus is the *first calculus* that can do so in a uniform framework. Furthermore, our normalization algorithm can normalize all expressions in the calculus. This is not possible for monoid homomorphisms in general.

The normalization algorithm is used in Section 4 for optimizing queries. Since non-canonical terms are suboptimal in most cases, we ignore all non-canonical forms during database optimization and search the space of equivalent canonical forms only. This heuristic reduces the optimization search space considerably. In addition, Section 4 expresses database storage structures and algorithms for joins, including merge join, hash join, and hash-partitioned join. In this area, monoid homomorphisms are superior to folds, because structures such as sorted lists and hash tables can be represented directly as monoids. Section 5 is focused on vector and matrix manipulations. These data structures deserve special

T	zero[T]	unit[T](a)	merge[T]	C/I
list	$[\,]$	$[a]$	append	
set	$\{\}$	$\{a\}$	$\cup$	CI
bag	$\{\!\{\}\!\}$	$\{\!\{a\}\!\}$	$\uplus$	C
ordered-set	$\{[\,]\}$	$\{[a]\}$	$\cup\!\!\!\cdot$	I
string	""	"a"	concat	

Table 1: Examples of Free Monoids

consideration since they occur very often in scientific applications. Section 6 integrates object-oriented features, such as object identity, into the base language. Section 7 extends the basic algebra with destructive updates and provides operational semantics to these extensions. Finally, Section 8 presents some of the related work.

2 The Theoretical Framework

2.1 Monoids

Definition 1 (Monoid) *Let T be a type,* zero[T] *a value of type T, and* merge[T] *a function of type $(T \times T) \rightarrow T$. The triple* ($T$, zero[$T$], merge[$T$]) *is a* monoid *of type T if* merge[T] *is associative with (left and right) identity* zero[T].

If merge[T] is a commutative (i.e., $\forall x, y :$ merge[T]$(x, y) =$ merge[T](y, x)) and/or idempotent function (i.e., $\forall x :$ merge[T]$(x, x) = x$), then the monoid (T, zero[T], merge[T]) is a *commutative* and/or *idempotent monoid.*

Definition 2 (Free Monoid) *Let $T(\alpha)$ be a type determined by the type parameter α (i.e., T is a type constructor) and* ($T(\alpha)$, zero[T], merge[T]) *be a monoid. The quadruple*

$$(T(\alpha), \text{zero}[T], \text{unit}[T], \text{merge}[T])$$

where unit[T] *is a function of type $\alpha \rightarrow T(\alpha)$, is a* free monoid*.

Table 1 presents some examples of free monoids. The letters C and I indicate that the monoid is a commutative and/or an idempotent monoid. The monoids *list*, *bag*, and *set* capture the well-known collection types for linear lists, multisets, and sets [5]. The monoid *ordered-set* captures lists with no duplicates. The operator $\cup\!\!\!\cdot$ is defined as follows: $x \cup\!\!\!\cdot\, y = \text{append}(x, x - y)$, where $x - y$ is the list x without any elements from y. e.g. $\{[2, 5, 3, 1]\} \cup\!\!\!\cdot \{[3, 2, 6]\} = \{[2, 5, 3, 1, 6]\}$. The monoid *string* captures vectors of any type. It is mostly used for expressing character strings. Section 5 describes a better monoid for complex vectors.

*To be more precise, unit[T] should satisfy the universal mapping property (uniqueness property) that for any $f : \alpha \rightarrow S(\alpha)$ there is a unique function hom[T, S](f) (given in Definition 5) such that hom[T, S](f) $\circ$ unit[T] $= f$.

T	type	zero[T]	unit[T](a)	merge[T]	C/I
sum	int	0	a	$+$	C
prod	int	1	a	$*$	C
max	int	0	a	max	CI
some	bool	false	a	$\vee$	CI
all	bool	true	a	$\wedge$	CI

Table 2: Examples of Simple Monoids

We will use the shorthand $T\{e_1, \ldots, e_n\}$ to represent the construction

$$\text{merge}[T](\text{unit}[T](e_1), \ldots, \text{merge}[T](\text{unit}[T](e_n), \text{zero}[T]))$$

In particular, we will use the following shorthands:

$$\begin{array}{rclrcl} [e_1, \ldots, e_n] & = & list\{e_1, \ldots, e_n\} & \{\!\{e_1, \ldots, e_n\}\!\} & = & bag\{e_1, \ldots, e_n\} \\ \{e_1, \ldots, e_n\} & = & set\{e_1, \ldots, e_n\} & \text{"}c_1 \ldots c_n\text{"} & = & string\{c_1, \ldots, c_n\} \end{array}$$

where c_i is a character.

One monoid that captures natural numbers is $(\text{int}, 0, +)$. Since the *int* type is not parametric, the *int* unit could be a constant value (e.g. 1) instead of a function. In that case, the *int* monoid will be freely generated by the constant value 1. Here though we adopt a different approach (so that monoid comprehensions can return integer values):

Definition 3 (Simple Monoid) *The quadruple* $(T, \text{zero}[T], \text{unit}[T], \text{merge}[T])$, *where* $(T, \text{zero}[T], \text{merge}[T])$ *is a monoid and* $\text{unit}[T]$ *is the identity function* $\text{id} = \lambda x.x$ *of type* $T \rightarrow T$, *is a* simple monoid.

The drawback of this approach is that natural numbers must now be built-in, since they cannot be generated from the *int* primitives alone. Table 2 gives examples of simple monoids. Every simple monoid should be associated with an underlying type (*int* and *bool* here), since instances of simple monoids cannot be generated from the monoid primitives alone; instead they need values of some kind to generate new values. Note also that all simple monoids should be commutative.

In our treatment of programs we will consider only the following types to be valid:

Definition 4 (Monoid Type) *A* monoid type *has one of the following forms:*

T	(T is a simple monoid)
$T(prtype)$	(T is a free monoid and *prtype* is int, bool, or char)
$T(type)$	(T is a free monoid)
$\langle a_1 : t_1, \ldots, a_n : t_n \rangle$	(a record type)

where type and $t_1, \ldots, t_n$ *are monoid types.*

That is, collection types can be freely nested.

Records of type $\langle a_1 : t_1, \ldots, a_n : t_n \rangle$ are constructed by expressions of the form $\langle a_1 = e_1, \ldots, a_n = e_n \rangle$. A component a_i of a record x can be extracted by using either $a_i(x)$ or $x.a_i$. We will use the shorthand

$$t_1 \times t_2 \;=\; \langle \pi_1 : t_1, \pi_2 : t_2 \rangle$$

to capture pairs. For example, set(string(char) $\times$ bag(int)) is a legitimate type.

Theorem 1 *A monoid type is a monoid.*

Proof: (by induction) If $type = T$ is a simple monoid, then the statement is true. If $type = T(prtype)$ where T is a free monoid and $prtype$ is int, bool, or char, then, $\text{zero}[type] = \text{zero}[T]$, $\text{unit}[type] = \text{unit}[T]$, and $\text{merge}[type] = \text{merge}[T]$. If $type = T(t)$ where T is a free monoid, then

$$\begin{array}{lcl} \text{zero}[T(t)] & = & \text{zero}[T] \\ \text{unit}[T(t)](x) & = & \text{unit}[T](\text{unit}[t](x)) \\ \text{merge}[T(t)](x,y) & = & \text{merge}[T](x,y) \end{array}$$

If $type$ is a record, then

$$\begin{array}{l} \text{zero}[\langle a_1 : t_1, \ldots, a_n : t_n \rangle] \;=\; \langle a_1 = \text{zero}[t_1], \ldots, a_n = \text{zero}[t_n] \rangle \\ \text{unit}[\langle a_1 : t_1, \ldots, a_n : t_n \rangle](\langle a_1 = v_1, \ldots, a_n = v_n \rangle) \\ \qquad\qquad = \;\; \langle a_1 = \text{unit}[t_1](v_1), \ldots, a_n = \text{unit}[t_n](v_n) \rangle \\ \text{merge}[\langle a_1 : t_1, \ldots, a_n : t_n \rangle](\langle a_1 = x_1, \ldots, a_n = x_n \rangle, \langle a_1 = y_1, \ldots, a_n = y_n \rangle) \\ \qquad\qquad = \;\; \langle a_1 = \text{merge}[t_1](x_1, y_1), \ldots, a_n = \text{merge}[t_n](x_n, y_n) \rangle \end{array}$$ □

For example, if $type = \text{set(string(char)} \times \text{bag(int))}$, then

$$\begin{array}{lcl} \text{zero}[type] & = & \{\} \\ \text{unit}[type](x,y) & = & \{(\text{"}x\text{"}, \{\!\{y\}\!\})\} \\ \text{merge}[type] & = & \cup \end{array}$$

since pairs are records with $\text{unit}[t_1 \times t_2](a_1, a_2) \;=\; (\text{unit}[t_1](a_1), \text{unit}[t_2](a_2))$.

When there are no ambiguities, we will use the notation T and $T \circ S$ instead of $T(type)$ and $T(S(type))$, where T and S are free monoids, if $type$ can be inferred from context.

We define the mapping ψ from monoid names to the set $\{C, I\}$ as: $C \in \psi(T)$ iff T is commutative and $I \in \psi(T)$ iff T is idempotent. Note that $\psi(T \circ S) = \psi(T)$. The partial order between monoid names $\preceq$ is defined as:

$$S \preceq T \;\Leftrightarrow\; \psi(S) \subseteq \psi(T)$$

For example, 'list' $\preceq$ 'bag' $\preceq$ 'set' since set is commutative-idempotent, bag is commutative but not idempotent, and list is neither commutative nor idempotent.

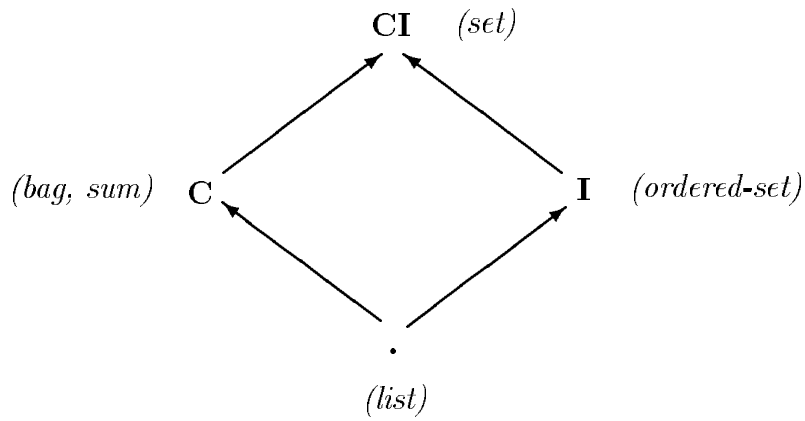

Figure 1: Restriction Lattice for Homomorphisms

2.2 The Algebra and the Calculus

Definition 5 (Homomorphism) *A* homomorphism $\mathrm{hom}[T,S]$ *from the free monoid* $(T(\alpha), \mathrm{zero}[T], \mathrm{unit}[T], \mathrm{merge}[T])$ *to the monoid* $(S, \mathrm{zero}[S], \mathrm{merge}[S])$*, where* $T \preceq S$*, has type* $(\alpha \to S) \to T(\alpha) \to S$ *and it is defined by the following inductive equations:*

$$\begin{array}{lcl} \mathrm{hom}[T,S](f)\,\mathrm{zero}[T] & = & \mathrm{zero}[S] \\ \mathrm{hom}[T,S](f)\,(\mathrm{unit}[T](a)) & = & f(a) \\ \mathrm{hom}[T,S](f)\,(\mathrm{merge}[T](x,y)) & = & \mathrm{merge}[S](\mathrm{hom}[T,S](f)\,x, \mathrm{hom}[T,S](f)\,y) \end{array}$$

The condition $T \preceq S$ is very important. If the free monoid T is a commutative and/or idempotent monoid, then so must be S (see also Figure 1). For example, bag cardinality $\mathrm{hom}[bag, sum](\lambda x.1)$ is a valid homomorphism, while set cardinality $\mathrm{hom}[set, sum](\lambda x.1)$ is not (since $+$ is commutative but not idempotent). Without this restriction we would have (see also [5]):

$$\begin{array}{rcl} 1 & = & \mathrm{hom}[set, sum](\lambda x.1)\{a\} \\ & = & \mathrm{hom}[set, sum](\lambda x.1)(\{a\} \cup \{a\}) \\ & = & \mathrm{hom}[set, sum](\lambda x.1)\{a\} + \mathrm{hom}[set, sum](\lambda x.1)\{a\} \;=\; 2 \end{array}$$

The following are examples of valid monoid homomorphisms:

$$\begin{array}{lcl} \mathrm{image}(f)\,x & = & \mathrm{hom}[set, set](\lambda a.\{f(a)\})\,x \\ e \in x & = & \mathrm{hom}[set, some](\lambda a.(a = e))\,x \\ \mathrm{filter}(p)x & = & \mathrm{hom}[set, set](\lambda a.\mathbf{if}\, p(a)\, \mathbf{then}\, \{a\}\, \mathbf{else}\, \{\})\,x \\ \mathrm{length}(x) & = & \mathrm{hom}[list, sum](\lambda a.1)\,x \end{array}$$

Definition 6 (Monoid Comprehension) *A* comprehension *over the monoid* $(T, \mathrm{zero}[T], \mathrm{unit}[T], \mathrm{merge}[T])$ *is defined by the following inductive equations:*

$$\begin{array}{lcll} T\{\, e \mid \,\} & = & \mathrm{unit}[T](e) & \\ T\{\, e \mid x \leftarrow u, \overline{r}\,\} & = & \mathrm{hom}[S,T](\lambda x.T\{\, e \mid \overline{r}\,\})\, u & \textit{(where } u \textit{ has type } S(\alpha)\textit{)} \\ T\{\, e \mid \mathrm{pred}, \overline{r}\,\} & = & \textbf{if}\ \mathrm{pred}\ \textbf{then}\ T\{\, e \mid \overline{r}\,\}\ \textbf{else}\ \mathrm{zero}[T] & \end{array}$$

where $\overline{r}$ *is a (possibly empty) sequence of terms (of the form* $x \leftarrow u$ *or pred) separated by commas, and* u *is an expression of type* $S(\alpha)$*, where* S *is a free monoid with* $S \preceq T$*.*

While the monoid T of the output is specified explicitly, the free monoid S associated with the expression u in $x \leftarrow u$ is inferred. This process is similar to type inference. Note that the type T that tags a comprehension may be a composition of monoids. In that case, as we have seen earlier, the unit function is the composition of the unit functions of the constituent monoids while the zero and merge functions are the zero and merge functions of the outer monoid. For example, $(\mathrm{set} \circ \mathrm{list})\{\, e \mid e \leftarrow [1,2]\,\}$ returns $\{[1],[2]\}$.

Terms of the form $x \leftarrow u$ in a comprehension (as in the second equation) are called *generators* while terms of the form pred (as in the third equation) are called *guards* or *filters*. We will use the shorthand $T\{\, e \mid x \Leftarrow u, \overline{r}\,\}$ for $T\{\, e \mid x \leftarrow [u], \overline{r}\,\}$, where $[u]$ is the list with one element u. Terms of the form $x \Leftarrow u$ are called *bindings* since they bind the variable x to u. For example, $\mathit{some}\{\, x = y \mid x \Leftarrow 3,\ y \Leftarrow 4\,\}$ is false.

The monoid comprehensions for $T = \mathit{set}$ are similar to the tuple relational calculus queries. For example, the join of two sets x and y is defined as:

$$\mathrm{join}(f,p)(x,y) \ = \ \mathit{set}\{\, f(a,b) \mid a \leftarrow x,\ b \leftarrow y,\ p(a,b)\,\}$$

and it is equal to:

$$\mathrm{hom}[\mathit{set},\mathit{set}](\lambda a.\mathrm{hom}[\mathit{set},\mathit{set}](\lambda b.\textbf{if}\ p(a,b)\ \textbf{then}\ \{f(a,b)\}\ \textbf{else}\ \{\})\, y)\, x$$

But we can also use comprehensions to join different collection types. For example,

$$\mathit{set}\{\, (x,y) \mid x \leftarrow [1,2],\ y \leftarrow \{3,4,5\}\,\}$$

returns $\{(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)\}$.

Another example is $\mathrm{nest}^T(k)\, x$:

$$\mathit{set}\{\, (k(e), s) \mid e \leftarrow x,\ s \Leftarrow T\{\, a \mid a \leftarrow x,\ k(e) = k(a)\,\}\,\}$$

For example,

$$\mathrm{nest}^{\mathit{list}}(\mathrm{id})\,[1,2,2,3,1] \ = \ \{(1,[1,1]),(2,[2,2]),(3,[3])\}$$

If x is a set, then $\mathrm{nest}^{\mathit{set}}(k)\, x$ is equivalent to the nesting operator for nested relations. Similarly†

$$\mathrm{unnest}^T(x) \ = \ T\{\, e \mid (k,s) \leftarrow x,\ e \leftarrow s\,\}$$

†We allow generators to take the form $\langle a_1 = e_1, \ldots, a_n = e_n\rangle \leftarrow u$:

$$T\{\, e \mid \langle a_1 = e_1, \ldots, a_n = e_n\rangle \leftarrow u, \overline{r}\,\} \ = \ T\{\, e \mid v \leftarrow u,\ e_1 \Leftarrow v.a_1, \ldots, e_n \Leftarrow v.a_n, \overline{r}\,\}$$

For example, $T\{\, e \mid (a,b) \leftarrow u, \overline{r}\,\} \ = \ T\{\, e \mid v \leftarrow u,\ a \Leftarrow \pi_1 v,\ b \Leftarrow \pi_2 v, \overline{r}\,\}$.

Other examples:

$$
\begin{array}{lcll}
\text{filter}(p)(x) & = & set\{\, e \,|\, e \leftarrow x,\, p(e) \,\} & (x \textit{ is a set, a list, or a bag}) \\
\text{flatten}(x) & = & set\{\, e \,|\, s \leftarrow x,\, e \leftarrow s \,\} & (x \textit{ is a set of sets}) \\
x \cap y & = & set\{\, e \,|\, e \leftarrow x,\, e \in y \,\} & (x \textit{ and } y \textit{ are sets, lists, or bags}) \\
\text{length}(x) & = & sum\{\, 1 \,|\, e \leftarrow x \,\} & (x \textit{ is a list or a bag}) \\
\text{sum}(x) & = & sum\{\, e \,|\, e \leftarrow x \,\} & (x \textit{ is a list or a bag of int}) \\
\text{count}(x, a) & = & sum\{\, 1 \,|\, e \leftarrow x,\, e = a \,\} & (x \textit{ is a list or a bag}) \\
a \in x & = & some\{\, a = e \,|\, e \leftarrow x \,\} & (x \textit{ is a set, a list, or a bag}) \\
\exists a \in x : e & = & some\{\, e \,|\, a \leftarrow x \,\} & (x \textit{ is a set, a list, or a bag}) \\
\forall a \in x : e & = & all\{\, e \,|\, a \leftarrow x \,\} & (x \textit{ is a set, a list, or a bag})
\end{array}
$$

Expression $\text{sum}(x)$ adds the elements of any x of type $T(\text{int})$, where $T \preceq$ 'int'. For example, $\text{sum}([1,2,3]) = 6$. Expression $\text{count}(x, a)$ counts the number of occurrences of a in the bag x. For example, $\text{count}(\{\!\{1,2,1\}\!\}, 1) = 2$.

Monoid comprehensions are more expressive than monad comprehensions [23]. For example, $set\{\, x \,|\, x \leftarrow [1,2,1] \,\}$ is a valid monoid comprehension that translates the list $[1,2,1]$ into the set $\{1,2\}$, but it is not a valid monad comprehension. This is a consequence of the fact that a monoid homomorphism is a generalization of the flattenmap (the monad extension operator [23]), since $\text{flattenmap}^T(f) = \text{hom}[T,T](f)$. In contrast to flattenmap, a monoid homomorphism can map one type to another type, allowing us to mix different types in the same computation. Notice also that $list\{\, x \,|\, x \leftarrow \{1,2\} \,\}$ is not a valid monoid comprehension while $sum\{\, x \,|\, x \leftarrow \{\!\{1,2\}\!\} \,\}$ is. That is, if one of the generators in a monoid comprehension is over a commutative and/or idempotent monoid, then this comprehension must be over a commutative and/or idempotent monoid. This condition could be checked by the type inference system and report errors to users, since the commutativity and idempotence properties of a monoid are specified explicitly when this monoid is defined (by the ψ function).

Definition 7 (Monoid Algebra) *A* monoid term *is defined in Figure 2, where v is a variable name, c is a constant (i.e.,* true, false, *an integer, or a character $'c'$), names starting with e represent monoid terms, and names starting with t represent monoid types. The notation $\sigma \vdash e : t$ indicates that the monoid term e has type t under the substitution σ. The substitution list σ binds variable names to types. Equations 9, 10, and 11 are for a free monoid $T(\alpha)$ but similar equations can be expressed for a simple monoid (by substituting T for $T(\alpha)$, $T(t)$, and t).*

Definition 8 (Monoid Calculus) *A* comprehension term *is similar to a monoid term but with the Equation 12 of Figure 2 replaced by the equation*

$$
\frac{\sigma_{n+1} \vdash e : t}{\sigma \vdash T\{\, e \,|\, a_1, \ldots, a_n \,\} : T(t)}
$$

for a free monoid $T(\alpha)$, where $n \geq 0$, $\sigma_1 = \sigma$, and each a_i is either

$$\sigma \vdash c : T \quad (1)$$

$$\sigma \vdash v : \sigma(v) \quad (2)$$

$$\frac{\sigma \vdash e_1 : t,\ \sigma \vdash e_2 : t}{\sigma \vdash e_1 = e_2 : \text{bool}} \quad (3)$$

$$\frac{\sigma \vdash e_1 : t_1, \ldots, \sigma \vdash e_n : t_n}{\sigma \vdash \langle a_1 = e_1, \ldots, a_n = e_n \rangle : \langle a_1 : t_1, \ldots, a_n : t_n \rangle} \quad (4)$$

$$\frac{\sigma \vdash e : \langle a_1 : t_1, \ldots, a_n : t_n \rangle}{\sigma \vdash e.a_i : t_i} \quad (5)$$

$$\frac{\sigma \vdash e_1 : \text{bool},\ \sigma \vdash e_2 : t,\ \sigma \vdash e_3 : t}{\sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t} \quad (6)$$

$$\frac{\sigma[t_1/v] \vdash e : t_2}{\sigma \vdash \lambda v^{t_1}.e : t_1 \rightarrow t_2} \quad (7)$$

$$\frac{\sigma \vdash e_1 : t_1 \rightarrow t_2,\ \sigma \vdash e_2 : t_1}{\sigma \vdash e_1(e_2) : t_2} \quad (8)$$

$$\sigma \vdash \text{zero}[T] : T(\alpha) \quad (9)$$

$$\frac{\sigma \vdash e : t}{\sigma \vdash \text{unit}[T](e) : T(t)} \quad (10)$$

$$\frac{\sigma \vdash e_1 : T(t),\ \sigma \vdash e_2 : T(t)}{\sigma \vdash \text{merge}[T](e_1, e_2) : T(t)} \quad (11)$$

$$\frac{\sigma \vdash e_2 : T(t),\ \sigma[t/v] \vdash e_1 : S(t),\ T \preceq S}{\sigma \vdash \text{hom}[T, S](\lambda v.e_1)\, e_2 : S(t)} \quad (12)$$

Figure 2: The Monoid Algebra

- *a generator* $v_i \leftarrow e_i$*, where* $\sigma_i \vdash e_i : S_i(t_i)$*,* $S_i \preceq T$*, and* $\sigma_{i+1} = \sigma_i[t_i/v_i]$*, or*
- *a filter* e_i*, where* $\sigma_i \vdash e_i : \text{bool}$ *and* $\sigma_{i+1} = \sigma_i$*.*

Similarly, a comprehension of a simple monoid T *is*

$$\frac{\sigma_{n+1} \vdash e : T}{\sigma \vdash T\{\, e \,|\, a_1, \ldots, a_n \,\} : T}$$

From the equivalence

$$\text{hom}[T, S](f)\, x \;=\; S\{\, a \,|\, v \leftarrow x,\ a \leftarrow f(v) \,\}$$

if S is a free monoid and

$$\text{hom}[T, S](f)\, x \;=\; S\{\, f(v) \,|\, v \leftarrow x \,\}$$

if S is a simple monoid, and from Definition 6 we have the following theorem:

Theorem 2 *The monoid algebra is equivalent to the monoid calculus.*

3 Normalization of Monoid Terms

In this section we are concerned with normalizing the monoid algebra into a simplier canonical form. There are four reasons for doing this. First, it is easier to prove general theorems

$$
\begin{array}{rcll}
(\lambda v.e_1)\, e_2 & \leadsto & e_1[e_2/v] & \textit{(beta reduction)} \quad (13)\\
\langle a_1 = e_1, \ldots, a_n = e_n \rangle.a_i & \leadsto & e_i & (14)\\
(\mathrm{hom}[T, \langle a_1 : t_1, \ldots, a_n : t_n \rangle](f)\, e).a_i & \leadsto & \mathrm{hom}[T, t_i](a_i \circ f)\, e & (15)\\
g(\mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3) & \leadsto & \mathbf{if}\, e_1\, \mathbf{then}\, g(e_2)\, \mathbf{else}\, g(e_3) & (16)\\
 & & \textit{(where } g \textit{ is either a projection or a hom)} & \\
\mathrm{hom}[T, S](f)\, \mathrm{zero}[T] & \leadsto & \mathrm{zero}[S] & (17)\\
\mathrm{hom}[T, S](f)\, (\mathrm{unit}[T](e)) & \leadsto & f(e) & (18)\\
\mathrm{hom}[T, S](f)\, (\mathrm{merge}[T](e_1, e_2)) & \leadsto & \mathrm{merge}[S](\mathrm{hom}[T, S](f)\, e_1,\ \mathrm{hom}[T, S](f)\, e_2) & (19)\\
\mathrm{hom}[S, R](f)\, (\mathrm{hom}[T, S](g)\, e) & \leadsto & \mathrm{hom}[T, R](\mathrm{hom}[S, R](f) \circ g)\, e & (20)
\end{array}
$$

Figure 3: The Normalization Algorithm for Homomorphisms

about properties of algebraic programs if these programs are reduced to canonical forms. Second, non-canonical terms are suboptimal in most cases. Therefore, a good heuristic for database optimization is to ignore all non-canonical forms and search the space of equivalent canonical forms for the one with the minimum estimated cost. Normalization reduces the optimization search space considerably. Third, the evaluation of canonical forms generates less intermediate data structures, which need to be garbage-collected after they are used. Finally, normalized programs are amenable to a higher degree of parallelism.

The normalization algorithm in Figure 3 maps the monoid algebra into a canonical monoid algebra. This canonical algebra is similar to the monoid algebra described in Figure 2 with the exception of Equation 12: a monoid homomorphism takes the following form:

$$\mathrm{hom}[S, T](f)\, path$$

where *path* is either *name* (the name of a bound variable) or *path.name* (where *name* is an attribute name of a record).

It is trivial to prove the soundness of the normalization algorithm (i.e., that all transformations are meaning preserving). For example, Rule 20 can be proved by induction over e (e can only take three forms: a zero, a unit, or a merge). It is also easy to see why there are no monoid homomorphisms after normalization other than those described above: if a monoid homomorphism is applied to a form other than a path, then this term is reduced to a simplier form by the normalization algorithm. We can also prove that this rewrite system is terminating and confluent (i.e., it satisfies the Church-Rosser property). That is, it does not matter which sequence of rules are applied to a term as they all result to the same canonical form.

The focus of the normalization algorithm is the last rule. It fuses two piped monoid homomorphisms into a nested one. This transformation facilitates parallel processing since

piped homomorphisms require that the two monoid homomorphisms are performed in sequence, one after the other, even though each one may be evaluated in parallel. That is, the inner homomorphism $\text{hom}[T, S](g)$ is computed first, constructing an intermediate result, and then the outer homomorphism $\text{hom}[S, R](f)$ is computed by consuming this intermediate result. These two computations can be fused into one parallel computation (i.e., into one monoid homomorphism). This is achieved by the normalization algorithm.

For example, $\text{filter}(p)(\text{filter}(q)\,x)$ is

$$
\begin{aligned}
& \text{hom}[set, set](\lambda b.\textbf{if}\, p(b)\,\textbf{then}\,\{b\}\,\textbf{else}\,\{\})\\
& \qquad (\text{hom}[set, set](\lambda a.\textbf{if}\, q(a)\,\textbf{then}\,\{a\}\,\textbf{else}\,\{\})\,x)\\
\rightsquigarrow\ & \text{hom}[set, set](\lambda a.(\text{hom}[set, set](\lambda b.\textbf{if}\, p(b)\,\textbf{then}\,\{b\}\,\textbf{else}\,\{\})\\
& \qquad (\textbf{if}\, q(a)\,\textbf{then}\,\{a\}\,\textbf{else}\,\{\}))\,x && \textit{(by Rule 20)}\\
\rightsquigarrow\ & \text{hom}[set, set](\lambda a.\textbf{if}\, q(a)\,\textbf{then}\,(\text{hom}[set, set](\lambda b.\textbf{if}\, p(b)\,\textbf{then}\,\{b\}\,\textbf{else}\,\{\})\,\{a\})\\
& \qquad \textbf{else}\,(\text{hom}[set, set](\lambda b.\textbf{if}\, p(b)\,\textbf{then}\,\{b\}\,\textbf{else}\,\{\})\,\{\}))\,x && \textit{(by Rule 16)}\\
\rightsquigarrow\ & \text{hom}[set, set](\lambda a.\textbf{if}\, q(a)\,\textbf{then}\,\textbf{if}\, p(a)\,\textbf{then}\,\{a\}\,\textbf{else}\,\{\}\,\textbf{else}\,\{\})\,x && \textit{(by Rules 17 and 18)}\\
=\ & \text{filter}(\lambda a.q(a) \wedge p(a))\,x
\end{aligned}
$$

The normalization algorithm improves program performance in most cases. There are two cases where the normalized programs may be less efficient than the programs before normalization. The first case is when we have a monoid homomorphism applied to an idempotent merge. For example, $\text{hom}[set, set](f)\,(\{a\} \cup \{a\}) = f(a)$ is more efficient than its normalized form $f(a) \cup f(a)$. This problem can be solved by using the rule: if T is idempotent, then $\text{merge}[T](e, e) \rightsquigarrow e$, and by requiring that the input of a hom be normalized before the hom itself (such as in a bottom-up rewriting strategy). The other case is when a variable occurs more than once in a term (i.e., the term is nonlinear). In that case, the beta reduction rule (Rule 13) will replace this variable with its bound term, producing a duplicated computation. This problem can be solved by generating DAGs instead of trees during beta reduction.

The normalization of the input of a homomorphism is a very important transformation because it eliminates intermediate data structures. However, another use of canonical forms is to recognize familiar patterns of expressions to translate them into efficient algorithms. One such pattern is a select-from-where SQL query whose 'where' clause contain another SQL query (i.e., this is a nested SQL query). We would like to transform such queries into joins since joins can be evaluated very efficiently. To do this transformation in our framework, we need to normalize if-then-else statements (since the 'where' clause of an SQL statement is captured in our algebra by an if-then-else statement). Figure 4 extends the algorithm of Figure 3 with rules that normalize if-then-else statements. It works for idempotent monoids only (such as sets). Canonical if-the-else statements take one of the following forms:

$$\textbf{if}\ path\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

$$\textbf{if}\ e_1 = e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4$$

that is, the condition of the if-then-else statement is normalized into a path or an equality (or a comparison in general).

$$\mathbf{if}\ \text{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_1 \tag{21}$$

$$\mathbf{if}\ \text{false}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_2 \tag{22}$$

$$\mathbf{if}\ e_1 \vee e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \rightsquigarrow \mathbf{if}\ e_1\ \mathbf{then}\ e_3\ \mathbf{else}\ \mathbf{if}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \tag{23}$$

$$\mathbf{if}\ e_1 \wedge e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \rightsquigarrow \mathbf{if}\ e_1\ \mathbf{then}\ \mathbf{if}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4\ \mathbf{else}\ e_4 \tag{24}$$

$$\mathbf{if}\ (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5 \rightsquigarrow \mathbf{if}\ e_1\ \mathbf{then}\ \mathbf{if}\ e_2\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5\ \mathbf{else}\ \mathbf{if}\ e_3\ \mathbf{then}\ e_4\ \mathbf{else}\ e_5 \tag{25}$$

$$\mathbf{if}\ \text{hom}[T, \mathit{some}](f)\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ \text{zero}[S] \rightsquigarrow \text{hom}[T, S](\lambda x.\mathbf{if}\ f(x)\ \mathbf{then}\ e_1\ \mathbf{else}\ \text{zero}[S])\ e \quad \textit{(if } S \textit{ is idempotent)} \tag{26}$$

$$\mathbf{if}\ \text{hom}[T, \mathit{all}](f)\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow \mathbf{if}\ \text{hom}[T, \mathit{some}](\mathit{not} \circ f)\ e\ \mathbf{then}\ e_2\ \mathbf{else}\ e_1 \tag{27}$$

Figure 4: The Normalization Algorithm for Conditionals

For example, consider the following nested OQL query:

```
select distinct r
   from r in R
   where r.B in (select distinct s.D
                 from s in S
                 where r.C = s.C)
```

which is expressed in the monoid calculus as:

$$\begin{aligned}
& \mathit{set}\{\, r \mid r \leftarrow R,\ \mathit{some}\{\, x = r.B \mid x \leftarrow \mathit{set}\{\, s.D \mid s \leftarrow S,\ r.C = s.C \,\}\,\}\,\} \\
= \ & \text{hom}[\mathit{set}, \mathit{set}](\lambda r.\mathbf{if}\,(\text{hom}[\mathit{set}, \mathit{some}](\lambda x.\, x = r.B) \\
& \qquad (\text{hom}[\mathit{set}, \mathit{set}](\lambda s.\mathbf{if}\, r.C = s.C\ \mathbf{then}\ \{s.D\}\ \mathbf{else}\ \{\})\, S)) \\
& \qquad \mathbf{then}\ \{r\}\ \mathbf{else}\ \{\})\, R \\
\rightsquigarrow \ & \text{hom}[\mathit{set}, \mathit{set}](\lambda r.\mathbf{if}\,(\text{hom}[\mathit{set}, \mathit{some}](\lambda s.\mathbf{if}\, r.C = s.C\ \mathbf{then}\ s.D = r.B\ \mathbf{else}\ \text{false})\, S) \\
& \qquad \mathbf{then}\ \{r\}\ \mathbf{else}\ \{\})\, R \qquad \textit{(by Rules 20 and 16)} \\
\rightsquigarrow \ & \text{hom}[\mathit{set}, \mathit{set}](\lambda r.\text{hom}[\mathit{set}, \mathit{set}](\lambda s.\mathbf{if}\, r.C = s.C\ \mathbf{then}\ \mathbf{if}\ s.D = r.B\ \mathbf{then}\ \{r\}\ \mathbf{else}\ \{\} \\
& \qquad \mathbf{else}\ \{\})\, S)\, R \qquad \textit{(by Rule 26)} \\
= \ & \mathit{set}\{\, r \mid r \leftarrow R,\ s \leftarrow S,\ r.C = s.C,\ s.D = r.B \,\}
\end{aligned}$$

Notice that this is a well-known transformation in relational systems that converts a nested select-from-where statement into a join.

The normalization algorithm yields canonical programs that generate very few intermediate data structures. But there are other important program transformations that explore the commutativity properties of monoids. In particular, if T is a commutative monoid, then

the following transformations are valid:

$$\begin{array}{lcl} \mathrm{hom}[S,T](\lambda v_2.\mathrm{hom}[R,T](\lambda v_1.e)\,e_1)\,e_2 & = & \mathrm{hom}[R,T](\lambda v_1.\mathrm{hom}[S,T](\lambda v_2.e)\,e_2)\,e_1 \\ \mathrm{hom}[S,T](\lambda v.\mathbf{if}\,e_1\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3)\,e & = & \mathbf{if}\,e_1\,\mathbf{then}\,\mathrm{hom}[S,T](\lambda v.e_2)\,e\,\mathbf{else}\,\mathrm{hom}[S,T](\lambda v.e_3)\,e \end{array}$$

The first equation is valid only if term e_1 does not depend on v_2 and e_2 does not depend on v_1, and the second equation is valid only if e_1 does not depend on v. The first equation is not a reduction. It corresponds to the join commutativity rule:

$$T\{\,e\,|\,v_1 \leftarrow e_1,\, v_2 \leftarrow e_2,\, \overline{n}\,\} \;=\; T\{\,e\,|\,v_2 \leftarrow e_2,\, v_1 \leftarrow e_1,\, \overline{n}\,\}$$

These rules can be used for generating alternatives during optimization. An optimizer should generate and estimate the costs of all these alternatives.

4 Mapping Monoid Terms into Physical Algorithms

Monoids and monoid terms can capture the data types and operations in both the logical and physical layers of a database. If a logical data type T is mapped into the physical storage structure S, then there must be a function $\mathcal{R}_{S,T}$ of type $S \rightarrow T$ that maps any instance of the storage structure S to an instance of type T. That is, the mapping from logical to physical must always be one-to-one or one-to-many. For example, sets can be implemented as lists with $\mathcal{R}_{list,set}(x) = set\{\,e\,|\,e \leftarrow x\,\}$ but not vice-versa. This mapping between types is called *type transformation* [11].

If a logical operation $f : (T_1 \times \cdots \times T_n) \rightarrow T_0$ is implemented by a physical algorithm $F : (S_1 \times \cdots \times S_n) \rightarrow S_0$, then the following equation must be true:

$$\mathcal{R}_{S_0,T_0} \circ F \;=\; f \circ (\mathcal{R}_{S_1,T_1} \times \cdots \times \mathcal{R}_{S_n,T_n})$$

where the product of functions is $(g_1 \times g_2)(a,b) = (g_1(a), g_2(b))$. If $\mathcal{R}_{S,T}(x) = T\{\,e\,|\,e \leftarrow x\,\}$ and $\mathcal{R}_{T,S}(x) = S\{\,e\,|\,e \leftarrow x\,\}$, then $F(x_1,\ldots,x_n)$ is equal to

$$S_0\{\,e\,|\,e \leftarrow f(T_1\{\,e\,|\,e \leftarrow x_1\,\},\ldots,T_n\{\,e\,|\,e \leftarrow x_n\,\})\,\}$$

That is, the implementation of any logical operation f expressed as a monoid term is exactly this monoid term with S_i substituted for T_i. (This also means that we substitute operators like merge$[S_i]$ for merge$[T_i]$.) This approach requires that $\psi(S_i) = \psi(T_i)$ and that the mapping between S_i and T_i is one-to-one.

Storage structures at the physical layer of a database can also be represented as monoids. For example, Figure 5 defines the monoids for sorted lists and hash tables. The monoid sorted$[f](\alpha)$, where type β in $f : \alpha \rightarrow \beta$ is an ordered type (i.e., a type with a partial order $\leq$, such as the integer type), denotes a list of elements of type α ordered by f. It represents a family of monoids, one for each function f. list_merge$(f)(x,y)$ merges two ordered lists x and y (both of type sorted$[f](\alpha)$) into an ordered list of type sorted$[f](\alpha)$. The monoid hashed$[f,n](\alpha)$, where $f : \alpha \rightarrow$ int and $n :$ int, denotes a hash table of size n and hash key f. Instances of this type could be constructed as vectors of n elements of type list(α). Each

free monoid T	zero^T	$\text{unit}^T(a)$	$\text{merge}^T(x,y)$	C/I
sorted$[f]$	$[\,]$	$[a]$	list_merge$(f)(x,y)$	CI
hashed$[f,n]$	empty_vector(n)	hash_unit$(f,n)(a)$	hash_merge$(f,n)(x,y)$	CI

Figure 5: Monoids for Physical Storage Structures

element e in the ith list satisfies $i = f(e) \bmod n$. hash_unit$(f,n)(a)$ creates a vector of n lists, which all of them are empty except the $f(a) \bmod n$ list which is the singleton list $[a]$. Finally, hash_merge$(f,n)(x,y)$ merges two hash tables x and y into a new hash table that contains elements from both x and y.

If we implement sets as sorted$[k]$ structures, then the implementation of join$(f,p)(x,y) = set\{\, f(a,b) \mid a \leftarrow x,\, b \leftarrow y,\, p(a,b) \,\}$ is

$$sorted[k]\{\, f(a,b) \mid a \leftarrow X,\, b \leftarrow Y,\, p(a,b) \,\}$$

where both X and Y are sorted$[k]$ monoids. The resulting algorithm has the same behavior as a merge join (even though its direct interpretation is still a nested loop). That is, it joins two sorted lists into a sorted list. However, as we will see next, we only care about the behavior and the cost model of a physical algorithm. A physical algorithm may implemented by a very complex program written in a language like C, but the only information needed for optimization is the algorithm's functionality expressed by a monoid term (to get consistent translations) and the cost model (to compare physical plans). The previous comprehension has exactly the functionality of a merge join since for the same inputs they both yield the same output.

Another implementation of join$(f,p)(x,y)$ is

$$hashed[k,n]\{\, f(a,b) \mid a \leftarrow X,\, b \leftarrow Y,\, p(a,b) \,\}$$

where now both X and Y are hashed$[k,n]$ monoids. The resulting algorithm has the same behavior as a hash-join. This ability of directly extracting the implementation of a logical operation is characteristic of monoid terms. It makes database implementation very effective. Lets now make these observations more formal:

Definition 9 (Physical Algorithm) *A* physical algorithm *F is a program (typically written in an imperative language, such as C) associated with the monoid type $T \rightarrow T'$, with the* behavior *f, where f is a monoid term of type $T \rightarrow T'$, and with a* cost function *that expresses the cost of this program in terms of the costs of its domain values.*

We will not give any formal definition of costs and of cost functions here. The physical layer consists of an extensible library of physical algorithms. We will assume that any physical algorithm F maps each value v in T into the value $F(v) = f(v)$ in T'. The physical layer is a trusted system where all physical algorithms are consistent with their behavior.

For example, merge_join$[k_1,k_2]$ is a physical algorithm with type

$$(\text{sorted}[k_1](T_1) \times \text{sorted}[k_2](T_2)) \rightarrow \text{sorted}[\lambda(a,b).k_1(a)](T_1 \times T_2)$$

The behavior of merge_join$[k_1, k_2](x, y)$ is

$$sorted[\lambda(a,b).k_1(a)]\{\,(a,b)\,|\,a \leftarrow x,\ b \leftarrow y,\ k_1(a) = k_2(b)\,\}$$

and its implementation could be a C procedure that evaluates a merge join. Another example of a physical algorithm is hash_join$[k_1, k_2, n]$ of type

$$(\text{hashed}[k_1, n](T_1) \times \text{hashed}[k_2, n](T_2)) \rightarrow \text{hashed}[\lambda(a,b).k_1(a) + k_2(b), n](T_1 \times T_2)$$

Its behavior is equal to

$$hashed[\lambda(a,b).k_1(a) + k_2(b), n]\{\,(a,b)\,|\,a \leftarrow x,\ b \leftarrow y,\ k_1(a) = k_2(b)\,\}$$

We will assume that all algorithm behaviors are reduced to canonical form by the normalization algorithm before optimization. Query translation in our framework can be viewed as a mapping from the monoid term $f(x_1, \ldots, x_n)$ that represents a query to the program $F(X_1, \ldots, X_n)$, where each physical structure X_i is the implementation of the logical structure x_i. The program $F(X_1, \ldots, X_n)$ consists of calls to physical algorithms in such a way that the behavior of this program is equivalent to

$$S_0\{\,e\,|\,e \leftarrow f(T_1\{\,e\,|\,e \leftarrow X_1\,\}, \ldots, T_n\{\,e\,|\,e \leftarrow X_n\,\})\,\}$$

Query optimization consists of the following steps. First, the above comprehension is normalized to a canonical form. Then, alternatives are generated from the commutative monoids (i.e., from join commutativity). Finally, the resulting canonical terms are pattern-matched with the canonical behaviors of the physical algorithms, substituting subterms with the proper calls to algorithms. Pattern-matching may sound inefficient but, because all terms are in canonical form, it can be performed very fast. This is a crucial point that makes the normalization process very important for query optimization. Note also that type checking narrows the matching process further. The search process should consider all matching physical algorithms each time. Cost functions can be used for guiding and pruning the search as well as for selecting the final best plan.

For example, we will map the collection type $T(\alpha)$ into the type $S(\alpha, \beta) = \text{set}(\beta \times T(\alpha))$ using

$$\begin{array}{rcll} \mathcal{R}_{T,S}(x) & = & \text{nest}^T(k)\,x & \textit{(for a function } k : \alpha \rightarrow \beta\textit{)} \\ \mathcal{R}_{S,T}(x) & = & \text{unnest}^T(x) & \end{array}$$

where nest and unnest were defined on Page 13. This type transformation basically partitions any collection into sets (equivalent classes) of collections whose elements have the same image under k. That is, k is the partition function. Under this type transformation, an operation $T\{\,a\,|\,a \leftarrow x,\ b \leftarrow y,\ k_1(a) = k_2(b)\,\}$ has the following implementation:

$$\text{nest}^T(k)(T\{\,a\,|\,a \leftarrow \text{unnest}^T(X),\ b \leftarrow \text{unnest}^T(Y),\ k_1(a) = k_2(b)\,\})$$

where X and Y are the implementations of x and y (i.e., they are already partitioned by k_1 and k_2) and the result will be partitioned by k. If there is an algorithm partitioned$(f)(x, y)$ with behavior

$$set\{\,(k, f(r,s))\,|\,(k,r) \leftarrow x,\ (m,s) \leftarrow y,\ m = k\,\}$$

$$
\begin{array}{rcll}
\text{subseq}(x,n,l) & = & vec[l]\{\, a[i-n] \mid a[i] \leftarrow x,\, i \geq n \,\} & \textit{(i.e. } x[i],\, i = n,\ldots,l+n-1\textit{)} \\
\text{reverse}(x) & = & vec[n]\{\, a[n-i-1] \mid a[i] \leftarrow x \,\} & \textit{(i.e. } x[i],\, i = n, n-1,\ldots,2,1\textit{)} \\
\text{zip}(x,y) & = & vec[n]\{\, (a,b)[i] \mid a[i] \leftarrow x,\, b[j] \leftarrow y,\, i = j \,\} & \textit{(i.e. } (x[i], y[i]),\, i = 1,\ldots,n\textit{)} \\
\text{permute}(x,p) & = & vec[n]\{\, a[b] \mid a[i] \leftarrow x,\, b[j] \leftarrow p,\, i = j \,\} & \textit{(i.e. } x[p[i]],\, i = 1,\ldots,n\textit{)} \\
\text{concat}(x,y) & = & \text{merge}[T](vec[n+m]\{\, a[i] \mid a[i] \leftarrow x \,\}, & \\
 & & \qquad vec[n+m]\{\, b[n+i] \mid b[i] \leftarrow y \,\}) & \\
mat & = & vec[n](vec[m](sum)) & \textit{(mat is the type of } n \times m \textit{ integer matrices)} \\
\text{map}(f)\, x & = & mat\{\, (f\, b)[i,j] \mid a[i] \leftarrow x,\, b[j] \leftarrow a \,\} & \\
\text{transpose}(x) & = & mat\{\, b[j,i] \mid a[i] \leftarrow x,\, b[j] \leftarrow a \,\} & \\
\text{inner}(x,y) & = & sum\{\, a \mid a[i] \leftarrow x * y \,\} & \\
\text{multiply}(x,y) & = & mat\{\, v[i,j] \mid a[i] \leftarrow x,\, b[j] \leftarrow \text{transpose}(y),\, v \Leftarrow \text{inner}(a,b) \,\} & \\
\text{determinant}(x) & = & sum\{\, v \mid a[k] \leftarrow x,\, b[l] \leftarrow a,\, v \Leftarrow ct(x,k,l) \,\} & \\
 & & \textbf{where } ct(x,k,l) \;=\; (-1)^{k*l} * prod\{\, b \mid a[i] \leftarrow x,\, i \neq k,\, b[j] \leftarrow a,\, j \neq l \,\} & \\
\text{inverse}(x) & = & mat\{\, v[k,l] \mid a[k] \leftarrow x,\, b[l] \leftarrow a,\, v \Leftarrow (ct(x,k,l)/\text{determinant}(x)) \,\} &
\end{array}
$$

Figure 6: Examples of Vector Operations

then the above operation can be evaluated by the following algorithm:

$$\text{partitioned}(\text{hash\_join}[k_1, k_2, n])(X, Y)$$

which actually implements a hash-partitioned join.

5 Vectors and Matrices

Vectors are very important collection types [13]. In contrast to other approaches, we will not express vectors as monoids whose merge function is a simple vector concatenation. Instead we will require that each vector has a fixed predefined size. We believe that such a view considerably simplifies the vector manipulation programs and facilitates parallel processing. Our work is closely related to Buneman's work on arrays [6]. (He uses transpose and reshape as array primitives.) We believe that our approach is simplier and more consistent with the basic theory for the other collection types.

We introduce a new free monoid vec$[n](T)$, for some monoid T and some constant integer n, to denote all vectors of size n with elements of type T. The monoid vec$[n](T)$ has the following primitives: (The unit function here is binary, but in functional languages any n-ary

function can be considered as a unary applied to a tuple.)

$$
\begin{array}{lcll}
\text{zero}[vec[n](T)] & = & (|\text{zero}[T], \ldots, \text{zero}[T]|) & (n \textit{ times}) \\
\text{unit}[vec[n](T)](a,k) & = & (|e_0, \ldots, e_{n-1}|) & \textit{where } e_i = \begin{cases} \text{unit}[T](a) & \text{if } i = (k \bmod n) \\ \text{zero}[T] & \text{otherwise} \end{cases} \\
\text{merge}[vec[n](T)]((|a_0, \ldots, a_{n-1}|), (|b_0, \ldots, b_{n-1}|)) & & & \\
 & = & (|\text{merge}[T](a_0, b_0), \ldots, \text{merge}[T](a_{n-1}, b_{n-1})|) &
\end{array}
$$

where $(|e_0, \ldots, e_{n-1}|)$ constructs a vector of n elements. That is, the zero element of $\text{vec}[n](T)$ is a vector of n zeros; the unit takes two values $a : T$ and $k : int$ and constructs a vector of n zeros, except of the kth element which is set to a; the merge function uses $\text{merge}[T]$ to merge the two input vectors element-wise. For example,

$$
\begin{array}{lcl}
\text{zero}[vec[4](int)] & = & (|0,0,0,0|) \\
\text{unit}[vec[4](int)](8,2) & = & (|0,0,8,0|) \\
\text{merge}[vec[4](int)]((|0,1,2,0|),(|3,0,2,1|)) & = & (|0+3, 1+0, 2+2, 0+1|) \;=\; (|3,1,4,1|)
\end{array}
$$

When there is no ambiguity, we will use $\text{merge}[T]$ for merging vectors instead of $\text{merge}[vec[n](T)]$. For example,

$$
\begin{array}{lcl}
(|1,2|) + (|3,4|) & = & (|4,6|) \\
\text{append}((|[1,2],[3]|),(|[4],[5,6]|)) & = & (|[1,2,4],[3,5,6]|)
\end{array}
$$

The matrix addition of two matrices x and y of type $mat = \text{vec}[m](\text{vec}[n](\text{sum}))$ is $x + y$, since $\text{merge}[mat]$ is $\text{merge}[vec[n](sum)]$, which is $\text{merge}[sum]$, equal to $+$.

Figure 6 presents some examples of vector operations. Here we use the shorthand $a[i]$ for (a,i) and $a[i,j]$ for $((a,j),i)$. To understand map, notice that $(f\,b)[i,j] = ((f\,b,j),i)$, which is lifted twice by the comprehension into the matrix $\text{unit}[vec[n](T)](\text{unit}[vec[m](T)](f\,b,j),i)$, since this comprehension is over a composition of monoids, and, therefore, the unit is a composition of units. That is, the composed unit is a matrix with all zeroes, except the ij^{th} element, which is $f\,b$. In a parallel architecture, this computation could be distributed into a matrix of processors (since it is a nested homomorphisms, which requires nested parallelism), where each processor is dedicated to one matrix element.

6 Incorporating Object Identity

In this section we are concerned with incorporating impure features, such as object identity, into our base language. To give operational semantics to such extensions, we need the ability to pass the state (such as the object store) through all operations in a program. That is, each value of type T must be *lifted* into a function $S \rightarrow (T \times S)$ that will propagate (and possibly change) the state s of type S to a new state s'. This function is called a *state transformer*. That way, impure features can be simulated directly in a pure functional language. The approach described here is inspired by Wadler's work on monads and monad state transformers [22, 23]. Our state transformers are equivalent to monad state transformers, but they are better suited for monoids[‡].

[‡]In particular, $\text{merge}[\Phi(T)](f,g) = \text{bindS}(f, \lambda x.\text{bindS}(g, \lambda y.\lambda s.(\text{merge}[T](x,y), s)))$, where $\text{bindS}(m,k)\,s = \textbf{let val}\,(a,s') = (m\,s)\,\textbf{in}\,k\,a\,s'\,\textbf{end}$ is the monad state transformer [22].

$$
\begin{array}{rcl}
[\![e]\!] & = & \lambda s.(e, s) \qquad \textit{(if } e \textit{ is first-order with no 'new' or ':=')} \\
[\![f]\!] & = & \lambda a.\lambda s.(f(a), s) \quad \textit{(if } f \textit{ is second-order with no 'new' or ':=')} \\
[\![(e_1, e_2)]\!] & = & \mathrm{distr} \circ (\mathrm{id} \times [\![e_2]\!]) \circ [\![e_1]\!] \\
[\![f(e)]\!] & = & (\lambda(a, s).[\![f]\!]\, a\, s) \circ [\![e]\!] \\
[\![\lambda v.e]\!] & = & \lambda v.[\![e]\!] \\
[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & (\lambda(a, s).\mathbf{if}\ a\ \mathbf{then}\ [\![e_2]\!]\, s\ \mathbf{else}\ [\![e_3]\!]\, s) \circ [\![e_1]\!] \\
[\![\mathrm{hom}[R, T](\lambda v.e)]\!] & = & \mathrm{hom}[R, \Phi(T)](\lambda v.[\![e]\!]) \\
[\![\mathrm{new}(s)]\!] & = & (\lambda(a, (L, H)).(\mathrm{ref}^T(n), (\mathrm{extend}^T(L, \mathrm{ref}^T(n), a), H - \{n\}))) \circ [\![s]\!] \\
 & & \textit{(for some } n \in H\textit{)} \\
[\![e := s]\!] & = & (\lambda(p, (a, (L, H))).(\mathrm{true}, (\mathrm{extend}^T(L, p, a), H))) \circ (\mathrm{id} \times [\![s]\!]) \circ [\![e]\!] \\
[\![!e]\!] & = & (\lambda(p, (L, H)).(\mathrm{lookup}^T(L, p), (L, H))) \circ [\![e]\!]
\end{array}
$$

Figure 7: Translation of the Monoid Algebra Extended with Object Identity

Definition 10 (Monoid State Transformer) *Let* $(T, \mathrm{zero}[T], \mathrm{merge}[T])$ *be a monoid. The triple* $(\Phi(T), \mathrm{zero}[\Phi(T)], \mathrm{merge}[\Phi(T)])$ *in which*

$$
\begin{array}{lcl}
\Phi(T) & = & S \rightarrow (T \times S) \\
\mathrm{zero}[\Phi(T)] & = & \lambda s.(\mathrm{zero}[T], s) \\
\mathrm{merge}[\Phi(T)](f, g) & = & (\mathrm{merge}[T] \times \mathrm{id}) \circ \mathrm{distr} \circ (\mathrm{id} \times g) \circ f
\end{array}
$$

where $\mathrm{distr}(a, (b, c)) = ((a, b), c)$*, is called a* monoid state transformer*.*

Function $\mathrm{merge}[\Phi(T)](f, g)$ returns an $\Phi(T)$ value by chasing the following diagram:

$$
S \xrightarrow{f} T \times S \xrightarrow{id \times g} T \times (T \times S) \xrightarrow{distr} (T \times T) \times S \xrightarrow{merge[T] \times id} T \times S
$$

In ML, $\mathrm{merge}[\Phi(T)](f, g)$ can be computed as follows:

$$
\mathrm{merge}[\Phi(T)](f, g)\, s \;=\; \left\{ \begin{array}{l}
\mathbf{let}\ \mathbf{val}\ (a, s_1) = f(s) \\
\quad\ \ \mathbf{val}\ (b, s_2) = g(s_1) \\
\mathbf{in}\ (\mathrm{merge}[T](a, b), s_2)\ \mathbf{end}
\end{array} \right.
$$

Theorem 3 *A monoid state transformer is a monoid.*

The proof of this theorem is given in the appendix. This theorem is crucial for guaranteeing the validity of the resulting programs. Note that T being a commutative and/or idempotent monoid does not necessarily imply that $\Phi(T)$ is too. Note also that while values of type T are lifted into functions of type $\Phi(T)$, functions of type $R \rightarrow T$ are lifted into $R \rightarrow \Phi(T)$. These are the only cases needed since we do not support functions of higher order.

The rest of this section is concerned with capturing object identity using monoid state transformers. In the following analysis we mix a pure functional language, namely the monoid algebra, with object manipulation operations. That way, one may use the pure functional core of the language for most of the operations, leaving the rest for the object-oriented part of the language. Our approach is based on Ohori's work on representing object identity using monads [14]. In his work he gives operational semantics to the lambda calculus extended with object identity in terms of the pure lambda calculus. But not any subset of the lambda calculus, such as a query algebra, extended with object identity can be expressed in terms of the pure algebra itself. The most notable contribution of our approach is that the resulting programs (after state transformation) are valid terms in our algebra and, therefore, are amenable to the same treatment as our basic algebra.

We extend the monoid algebra with the following impure operations:

$$\begin{array}{lcl} \text{new} & : & T \rightarrow \text{obj}(T) \\ ! & : & \text{obj}(T) \rightarrow T \\ := & : & \text{obj}(T) \times T \rightarrow \text{bool} \end{array}$$

Type obj(T) is a new primitive type that depends on the monoid T. It represents all objects with state T. new(s) creates a new object with state s, $!e$ dereferences the object e (returns the state of e), and $e := s$ changes the state of the object e into s (it returns true). Expressions of the form $e := s$ should appear only as guards in a comprehension (they do not filter out any comprehension values, since they are always true, but they side-effect the object store). For a detailed explanation of these operators, the reader is referred to Ohori's paper [14]. The state s of the state transformer is the pair (L, H), called the *object heap*. Value L is called the *object store*. It contains pairs of the form (p, v) that bind an object p (i.e., the *object identity* of p) to its state v. Value H is called the *Oid pool*. It is a set of integers that correspond to the available object identifiers. Typically, H is the set of natural numbers. It is highly desirable to have the Oid pool as a set where we can randomly choose numbers instead of using a counter (an approach adopted in [14]), because, as we will see, it makes the $\Phi(T)$ monoid compatible with T.

The new extended algebra is translated by the rules in Figure 7. $[\![t]\!]$ translates a term t in the extended algebra into a term in the monoid algebra. The first two rules translate terms that do not contain any *new* or := subterms. If there is a $!x$ subterm in e or f, it is translated into lookup($\pi_1 s, x$). Note that e in $f(e)$ in the fourth rule is evaluated before f is applied (i.e., this is a call-by-value), that is, the state changes in e are performed before the state changes in f.

The first seven rules in Figure 7 cover most operations in our algebra (records can be handled like pairs). For example, $[\![\text{merge}[T](e_1, e_2)]\!] = (\lambda(a, s).[\![\text{merge}[T]]\!]\, a\, s) \circ [\![(e_1, e_2)]\!] = (\lambda(a, s).(\text{merge}[T]\, a, s)) \circ \text{distr} \circ (\text{id} \times [\![e_2]\!]) \circ [\![e_1]\!] = \text{merge}[\Phi(T)]([\![e_1]\!], [\![e_2]\!])$. Only the last three rules in Figure 7 manipulate the state. All the other rules simply propagate the state unchanged. Type T in the last three rules is inferred from context. The eighth rule that translates new(s) selects an integer n from the pool H in a non-deterministic way.

We used five constructs to manipulate the object heap: operation *ref* translates an integer into an Oid of type T, *emptyStore* is the initial store value, *OidPool* is the initial Oid pool,

operation *extend* updates the store, and operation *lookup* accesses the store. They have the following types:

$$\begin{array}{lcl} \text{ref}^T & : & \text{int} \rightarrow \text{obj}(T) \\ \text{emptyStore} & : & \text{store} \\ \text{OidPool} & : & \text{set(int)} \\ \text{extend}^T & : & (\text{store} \times \text{obj}(T) \times T) \rightarrow \text{store} \\ \text{lookup}^T & : & (\text{store} \times \text{obj}(T)) \rightarrow T \end{array}$$

Operations of this form can be simplified by the following rules:

$$\begin{array}{lcll} \text{ref}^T(n) = \text{ref}^T(m) & \leadsto & n = m & \\ \text{extend}^T(\text{extend}^T(s, x, v), y, u) & = & \text{extend}^T(\text{extend}^T(s, y, u), x, v) & \textit{if } x \neq y \\ \text{extend}^T(\text{extend}^T(s, x, v), x, u) & \leadsto & \text{extend}^T(s, x, u) & \\ \text{lookup}^T(\text{extend}^T(s, x, v), y) & \leadsto & \textbf{if } x = y \textbf{ then } v \textbf{ else } \text{lookup}^T(s, y) & \end{array}$$

The only rules needed for optimizing object-oriented comprehensions are the above reduction rules, the rules of Figure 7, and the normalization algorithm. In fact, most parts of the resulting programs can be beta-reduced to first-order programs that look very similar to the programs one might write using the four store manipulation functions directly. The resulting programs can be implemented very efficiently if the store is a global array. All the rules except the last one are *single-threaded* [16]. Roughly speaking, an expression that involves a state is single-threaded if there is no need of undoing state modifications at any point, that is, if there is a strict sequencing of state updates. In that case there will always be only one pointer to the store, and therefore, this store can be implemented as a global store and all updates as in-place (destructive) updates. Since no object creation is undone at any point, the object store is single-threaded. The last rule can enforce a single array pointer by fetching the state first (using *lookup*) and then by returning the pointer to L. Any operation on object store can be done destructively. For example, $\text{extend}^T(s, x, v)$ will ignore the parameter s, it will modify the global variable that contains the object store (by inserting the binding from x to v), and it will return a dummy value (e.g. 0). This means that we can evaluate the resulting programs after state transformation as efficiently as the real object-oriented programs. Single-threadedness here is enforced by language constraints, instead of using expensive compile-time analysis to detect it, as is done in some functional languages.

For example, we will normalize the following object-oriented comprehension:

$$\begin{array}{l} sum\{\, !x \mid x \Leftarrow \text{new}(1),\ x := !x + 1 \,\} \\ \qquad\qquad = \text{hom}[list, sum](\lambda x.\textbf{if}\,(x := !x + 1)\,\textbf{then}\,!x\,\textbf{else}\,0)\ [\text{new}(1)] \end{array}$$

We have $[\![x := !x + 1]\!] = \lambda(L, H).(\text{true}, (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H))$. Therefore,

$$
\begin{aligned}
& [\![\textbf{if}\,(x := !x + 1)\,\textbf{then}\,!x\,\textbf{else}\,0]\!] \\
= \;& (\lambda(a, (L, H)).\textbf{if}\,a\,\textbf{then}\,(\text{lookup}^{int}(L, x), (L, H))\,\textbf{else}\,(0, (L, H))) \\
& \circ(\lambda(L, H).(\text{true}, (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H))) \\
= \;& \lambda(L, H).\textbf{if}\,\text{true}\,\textbf{then}\,(\text{lookup}^{int}(\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), x), \\
& \qquad (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H))\,\textbf{else}\,\ldots \\
= \;& \lambda(L, H).(\text{lookup}^{int}(\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), x), \\
& \qquad (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H)) \\
= \;& \lambda(L, H).(\text{lookup}^{int}(L, x) + 1, (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H))
\end{aligned}
$$

In addition, $[\![[\text{new}(1)]]\!] = \lambda(L, H).([\text{ref}^{int}(n)], (\text{extend}^{int}(L, \text{ref}^{int}(n), 1), H - \{n\}))$. Therefore,

$$
\begin{aligned}
& [\![sum\{\,!x \mid x \Leftarrow \text{new}(1),\ x := !x + 1\,\}]\!] \\
= \;& \text{hom}[list, \Phi(sum)](\lambda x.\lambda(L, H).(\text{lookup}^{int}(L, x) + 1, (\text{extend}^{int}(L, x, \text{lookup}^{int}(L, x) + 1), H))) \\
& \qquad [\text{ref}^{int}(n)]\ (\text{extend}^{int}(L, \text{ref}^{int}(n), 1), H - \{n\}) \\
= \;& \lambda(L, H).(R + 1, (\text{extend}^{int}(\text{extend}^{int}(L, \text{ref}^{int}(n), 1), \text{ref}^{int}(n), R + 1), H - \{n\})) \\
& \quad where\ R = \text{lookup}^{int}(\text{extend}^{int}(L, \text{ref}^{int}(n), 1), \text{ref}^{int}(n)) = 1 \\
= \;& \lambda(L, H).(2, (\text{extend}^{int}(L, \text{ref}^{int}(n), 2), H - \{n\}))
\end{aligned}
$$

That is, the resulting value is 2 and the state is augmented by the binding from a new OID to 2.

Similarly, the following comprehensions are normalized into values when they applied to some state:

$$
\begin{array}{lcl}
some\{\,x = y \mid x \Leftarrow \text{new}(1),\ y \Leftarrow \text{new}(1)\,\} & \rightsquigarrow & \text{false} \\
some\{\,!x = !y \mid x \Leftarrow \text{new}(1),\ y \Leftarrow \text{new}(1)\,\} & \rightsquigarrow & \text{true} \\
some\{\,x = y \mid x \Leftarrow \text{new}(1),\ y \Leftarrow x,\ y := 2\,\} & \rightsquigarrow & \text{true} \\
sum\{\,!x \mid x \Leftarrow \text{new}(1),\ y \Leftarrow x,\ y := 2\,\} & \rightsquigarrow & 2 \\
set\{\,e \mid x \Leftarrow \text{new}([\,]),\ x := [1, 2],\ e \leftarrow !x\,\} & \rightsquigarrow & \{1, 2\} \\
list\{\,!x \mid x \leftarrow [\text{new}(1), \text{new}(2)],\ x := !x + 1\,\} & \rightsquigarrow & [2, 3] \\
list\{\,!x \mid x \Leftarrow \text{new}(0),\ e \leftarrow [1, 2, 3, 4],\ x := !x + e\,\} & \rightsquigarrow & [1, 3, 6, 10]
\end{array}
$$

Because of the non-determinism introduced when we choose an arbitrary number $n \in H$ for the OID of a new object, we can support sets of objects without any inconsistency. For example, $\{\text{new}(1), \text{new}(2)\}$ is a valid expression and it is equal to $\{\text{new}(2), \text{new}(1)\}$:

$$
\begin{aligned}
& [\![\{\text{new}(1), \text{new}(2)\}]\!] \\
= \;& \lambda(L, H).(\{\text{ref}^{int}(n), \text{ref}^{int}(m)\}, \\
& \qquad (\text{extend}^{int}(\text{extend}^{int}(L, \text{ref}^{int}(n), 1), \text{ref}^{int}(m), 2), H - \{n, m\})) \\
= \;& \lambda(L, H).(\{\text{ref}^{int}(m), \text{ref}^{int}(n)\}, \\
& \qquad (\text{extend}^{int}(\text{extend}^{int}(L, \text{ref}^{int}(m), 2), \text{ref}^{int}(n), 1), H - \{m, n\})) \\
= \;& [\![\{\text{new}(2), \text{new}(1)\}]\!]
\end{aligned}
$$

However, there are still problems with assignments. For example, $\{sum\{\,1 \mid x := 1\,\},\ sum\{\,2 \mid x := 2\,\}\}$ will bind x to 2 while $\{sum\{\,2 \mid x := 2\,\},\ sum\{\,1 \mid x := 1\,\}\}$ will bind x to 1. Similarly,

$set\{\, a \,|\, a \leftarrow A,\ x := !x + a \,\}$ is not valid since $+$ in $!x + a$ is not idempotent. Here we present a syntactic restriction to programs (reminiscent to $T \preceq S$ for $\mathrm{hom}[T, S]$) that always rejects inconsistent programs. We believe that this is a sufficient (even though not a necessary) condition for a program to be valid. The proof of this conjecture is a topic for future work.

Conjecture 1 *An object-oriented algebraic form is valid if it does not contain subterms of the following forms (for a commutative or idempotent monoid T):*

1. $\mathrm{merge}[T](\ldots x := e_1 \ldots, \ldots x := e_2 \ldots)$, *where x was bound outside this merge and terms e_1 and e_2 are not of the form* $\mathrm{merge}[S](!x, e)$, *where $T \preceq S$ and the term e does not contain a term $!y$ where y was bound outside this merge;*

2. $\mathrm{hom}[R, T](\lambda v. \ldots x := e_1 \ldots)\, e_2$, *where x was bound outside this hom and the term e_1 is not of the form* $\mathrm{merge}[S](!x, e)$, *where $T \preceq S$ and the term e does not contain a term $!y$ where y was bound outside this hom.*

For example, $\{sum\{\, 1 \,|\, x := 1 \,\},\ sum\{\, 2 \,|\, x := 2 \,\}\}$ is not valid because it is of the first form, and $set\{\, a \,|\, a \leftarrow A,\ x := !x + a \,\}$ is not valid because $!x + a$ is constructed as a *sum* monoid, which is not idempotent (i.e., $set \not\preceq sum$). However, $set\{\, a \,|\, a \leftarrow A,\ a := !a + 1 \,\}$ is a valid expression since a is defined locally in the comprehension, and $set\{\, a \,|\, a \leftarrow A,\ x := !x \cup a \,\}$ is valid since $\cup$ is idempotent.

7 Destructive Updates

The state transformer introduced in the previous section can also be used for expressing destructive updates in terms of the basic algebra. That way database transactions and queries can be optimized in a single framework. Let *db* be the current database state with type *DB* (a monoid type). Typically, *db* is the aggregation of all persistent objects in a program. A database update in an applicative language can be expressed as a function of type $DB \rightarrow DB$ that consumes the current database and constructs a new database. If this function was evaluated naively, it would result into a lot of unnecessary copying, which could be avoided if this update was evaluated directly in an imperative language. That is, we have a tension here: on one hand we want an equational theory for updates that would allow us to do algebraic optimization and reasoning, and on the other hand we want the resulting programs to be evaluated doing destructive updates to the database. That is, following the analysis of the previous section, we want to translate updates into the monoid algebra in such a way that the propagated state (i.e., the database) is single-threaded.

The state transformer used here is

$$\Phi(T) \;=\; DB \rightarrow (T \times DB)$$

that is, the state propagated through all expressions is *db*, the actual database. Object identity can be handled by the above state transformer by incorporating the object heap into *db*.

$$
\begin{array}{rcl}
\mathcal{T}[\![e]\!]\sigma & = & \lambda s.(e[db/s], s) \qquad \textit{(if } e \textit{ is first-order with no updates)} \\
\mathcal{T}[\![f]\!]\sigma & = & \lambda a.\lambda s.(f(a), s) \qquad \textit{(if } f \textit{ is second-order with no updates)} \\
\mathcal{T}[\![(e_1, e_2)]\!]\sigma & = & \mathrm{distr} \circ (\mathrm{id} \times (\mathcal{T}[\![e_2]\!]\sigma)) \circ (\mathcal{T}[\![e_1]\!]\sigma) \\
\mathcal{T}[\![f(e)]\!]\sigma & = & (\lambda(a, s).(\mathcal{T}[\![f]\!]\sigma)\, a\, s) \circ (\mathcal{T}[\![e]\!]\sigma) \\
\mathcal{T}[\![\lambda v.e]\!]\sigma & = & \lambda v.(\mathcal{T}[\![e]\!]\sigma) \\
\mathcal{T}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\sigma & = & (\lambda(a, s).\mathbf{if}\ a\ \mathbf{then}\ \mathcal{T}[\![e_2]\!]\sigma\, s\ \mathbf{else}\ \mathcal{T}[\![e_3]\!]\sigma\, s) \circ (\mathcal{T}[\![e_1]\!]\sigma) \\
\mathcal{T}[\![\mathrm{hom}[R, T](\lambda v_1.e)\,(\rho v_2)]\!]\sigma & = & \left\{ \begin{array}{l} \lambda s.\mathrm{hom}[R, \Phi(T)](\lambda v_1.\mathcal{T}[\![e]\!]\sigma')\,(\rho v_2)\, s \\ \mathbf{where}\ \sigma' = [(\lambda(a, s).(\sigma v_2)(\mathcal{P}[\![\rho]\!](a, v_2), s))/v_1] \end{array} \right. \\
\mathcal{T}[\![\mathrm{update}(\rho v, u, e)]\!]\sigma & = & (\lambda(a, s).((\lambda v.a)\,(\mathcal{P}[\![\rho]\!](u, v)), (\sigma v)(\mathcal{P}[\![\rho]\!](u, v), s)) \circ (\mathcal{T}[\![e]\!]\sigma)
\end{array}
$$

Figure 8: Translation of the Monoid Algebra Extended with Updates

We extend Definition 6 with the following destructive operation:

$$T\{\, e \,|\, \rho v \Leftarrow u,\ \overline{r}\,\} \quad = \quad \mathrm{update}(\rho v, u, T\{\, e \,|\, \overline{r}\,\})$$

Expression ρv denotes a path (i.e., ρ is a sequence of projections $a_{i_1} \circ a_{i_2} \circ \cdots \circ a_{i_n}$, where each a_{i_k} is a record attribute). Expression $\rho v \Leftarrow u$ destructively replaces ρv with u. Operation *update* is an impure operation that modifies the database. We will also use the following shorthands:

$$
\begin{array}{rcl}
\rho v \Leftarrow + \ u & = & \rho v \Leftarrow \mathrm{merge}[S](\mathrm{unit}[S](u), \rho v) \\
\rho v \Leftarrow - \ u & = & \rho v \Leftarrow S\{\, a \,|\, a \leftarrow \rho v,\ a \neq u\,\}
\end{array}
$$

That is, $\rho v \Leftarrow + \ u$ inserts u into ρv (ρv is a monoid of type S) and $\rho v \Leftarrow - \ u$ deletes all elements of ρv equal to u.

For example, if $DB = \mathrm{set}(\mathrm{int})$, then

$$sum\{\, 1 \,|\, a \leftarrow db,\ a > 10,\ a \Leftarrow + \ 1\,\}$$

increments every database element greater than 10 by one. It returns the number of elements updated.

Figure 8 gives the translation $\mathcal{T}[\![e]\!]\sigma$ of the algebraic expression e extended with updates. Operation $\mathcal{P}[\![\rho]\!]$ is defined as follows:

$$
\begin{array}{rcl}
\mathcal{P}[\![\mathrm{id}]\!] & = & \lambda(e, s).e \\
\mathcal{P}[\![\rho \circ a_i]\!] & = & \lambda(e, s).\langle a_1 = s.a_1, \ldots, a_i = \mathcal{P}[\![\rho]\!](e, s.a_i), \ldots, a_n = s.a_n \rangle \\
 & & \text{where } s \text{ is of type } \langle a_1 : t_1, \ldots, a_n : t_n \rangle
\end{array}
$$

For example, $\mathcal{P}[\![\pi_2 \circ \pi_1]\!] = \lambda(e, s).((\pi_1 \pi_1 s, e), \pi_2 s)$. That is, $\mathcal{P}[\![\rho]\!](e, s)$ is equal to a copy of s but with the ρs part of s replaced by e.

A *generalized path* of a variable v in a database update e is a sequence of paths $[\rho_1, \ldots, \rho_n]$ such that there exist a sequence of variables $[a_0, \ldots, a_n]$ with $a_0 = db$ and $a_n = v$ such that

$\mathrm{hom}[T,S](\lambda a_i.e_i)\,(\rho_i a_{i-1})$ is a term in e. This definition basically says that v is generated by a comprehension that looks like this:

$$T\{\, u \mid a_1 \leftarrow \rho_1 db,\ a_2 \leftarrow \rho_2 a_1, \ldots, v \leftarrow \rho_n a_{n-1} \,\}$$

The function $\mathcal{GP}[\![v]\!]$ is defined inductively for any such generator $a_i \leftarrow \rho_i a_{i-1}$:

$$\mathcal{GP}[\![a_i]\!] \;=\; \lambda(n,s).(\mathcal{GP}[\![a_{i-1}]\!])\,(\mathcal{P}[\![\rho_i]\!](n,a_{i-1}),s)$$

with $\mathcal{GP}[\![db]\!] = \lambda(n,s).n$. The idea behind $\mathcal{GP}[\![v]\!]$ is that the expression $\mathrm{unit}^{DB}(\mathcal{GP}[\![v]\!](n,db))$ (at the point v is defined) will return a slice of the database db but with v replaced by n. The binding list σ in the algorithm of Figure 8 binds a variable v to $\mathcal{GP}[\![v]\!]$. That is, $\sigma v = \mathcal{GP}[\![v]\!]$. For example, if a comprehension includes the generators $a \leftarrow \pi_1 db$ and $b \leftarrow \pi_2 a$, then $\sigma = [(\lambda(e,s).(e,\pi_2 s))/a, (\lambda(e,s).((\pi_1 a, e),\pi_2 s))/b]$. Initially $\sigma = [(\lambda(e,s).e)/db]$.

We assume that the algebraic operation e has been normalized to canonical form before the evaluation of $\mathcal{T}[\![e]\!]\sigma$. That is, all monoid homomorphisms are over paths of the form ρv. Variables v_2 and v in the last two rules are assumed different than db. Otherwise, s is substituted for db. The rule before the last, which translates a normalized monoid homomorphism, is very similar to the one in Figure 7. Here though σ is extended to include the binding from v_1 to $\mathcal{GP}[\![v_1]\!]$. The last rule translates a destructive update. It returns the pair of a value and a new state. The value is e but with any occurrence of variable v replaced by $\mathcal{P}[\![\rho]\!](u,v)$ (i.e., this a copy of v in which ρv is replaced by u). The new state is the current state but with ρv replaced by u.

For example, $set\{\, a \mid a \leftarrow db,\ a \Leftarrow\!+ 1 \,\}$ is translated into

$$\mathrm{hom}[set,\Phi(set)](\lambda a.\lambda s.(a+1,s+1))\,db\,db$$

As another example, consider $DB = \mathrm{set}(\mathrm{int} \times \mathrm{list}(\mathrm{int} \times \mathrm{int})) \times \mathrm{int}$. Then

$$set\{\, \pi_1 b \mid a \leftarrow \pi_1 db,\ b \leftarrow \pi_2 a,\ \pi_1 b \Leftarrow\!+ \pi_2 db \,\}$$

is translated into

$$\begin{aligned}&\mathrm{hom}[set,\Phi(set)](\lambda a.\mathrm{hom}[list,\Phi(list)](\lambda b.\lambda s.(\pi_1 b+\pi_2 s,((\pi_1 a,(\pi_1 b+\pi_2 s,\pi_2 b)),\pi_2 s)))\,(\pi_2 a))\\ &\qquad(\pi_1 db)\,db\end{aligned}$$

Notice that $\mathrm{unit}[DB]((a,(b,c)),d) = (\{(a,[(b,c)])\},d)$. Therefore, the value$(\pi_1 b+\pi_2 s,((\pi_1 a,(\pi_1 b+\pi_2 s,\pi_2 b)),\pi_2 s))$ will be lifted into an $\mathrm{int} \times DB$ value in which the integer will hold the resulting value while the DB will be the new database state.

We believe that the state transformer for destructive updates is single-threaded. That is, no database update is undone at any point. Of course, in real database systems, transactions are not single-threaded because of the transaction atomicity: some database updates need to be undone in case of failure. There are many ways to overcome this problem and all these ways constitute the database recovery mechanism (e.g. by copying the database state before a transaction or by using shadowing). The resulting programs after state transformation can be evaluated using destructive updates if we make the operation $\mathcal{P}[\![\rho]\!](e,s)$ destructive. That

is, instead of returning a copy of s with ρs replaced by ϵ, we destructively update ρs into ϵ. Operations $\mathcal{P}[\![\rho]\!]$ though are unfolded during program optimization. Since recognition of destructive updates should come after optimization, we would need the inverse of $\mathcal{P}[\![\rho]\!]$ to generate calls of the form $\mathcal{P}[\![\rho]\!](\epsilon, s)$ by pattern-matching path expressions.

The above analysis can be extended to include destructive updates on vectors. The vector updates supported are either vector- or element-based. For example, $v[i] \Leftarrow + 1$ increments the ith element of v by one. A path ρv can now include vector accesses. For example, if the path is $v.A[i].C$, then $\rho = C \circ (\lambda x.x[i]) \circ A$. We extend the $\mathcal{P}[\![\rho]\!]$ function as follows:

$$\mathcal{P}[\![\rho \circ (\lambda v.v[k])]\!] \;=\; \lambda(\epsilon, s).(\mathrm{vec}[n](T))\{\,(\mathbf{if}\, i = k\, \mathbf{then}\, \mathcal{P}[\![\rho]\!](\epsilon, a)\, \mathbf{else}\, a)[i] \mid a[i] \leftarrow s\,\}$$

where the type of s is $\mathrm{vec}[n](T)$. That is, we reconstruct the vector s by copying all its elements except the kth element $s[k]$ whose $\rho(s[k])$ part is set to ϵ.

Furthermore, it is easy to see what constitute a valid updatable view in our framework. For a set of variables V we define the following term form:

$$\mathcal{E}[V] \;:==\; \rho v \mid \langle a_1 : \mathcal{E}[V], \ldots, a_n : \mathcal{E}[V]\rangle$$

where $v \in V$. An updatable view is either $\mathcal{E}[\{db\}]$ or $T\{\,\mathcal{E}[\{x_1, \ldots, x_n\}] \mid \overline{r}\,\}$, where each x_i is a generalized path defined in $\overline{r}$. It is easy to prove that if a view v is defined as a term of this form, then v can be used in place of db and any view update can be compiled into a database update by simply unfolding the view definition.

8 Related Work

Monoid homomorphisms as a database query language were first introduced by V. Tannen and P. Buneman [3, 5, 4]. Their forms of monoid homomorphisms (also called structural recursion over the union presentation – SRU) are more expressive than ours. In particular, the Hom operator presented on Page 3 is similar to the SRU operation for lists. Operations of the form $\mathrm{Hom}^{set}(e, f, acc)$, though, require that acc be associative, commutative, and idempotent with a zero element ϵ. These properties are very hard to check by a compiler [5], which makes the SRU operation impractical. They first recognized that there are some special cases where these conditions are automatically satisfied, such as for the $\mathrm{ext}(f)$ operator (which is equivalent to flattenmap). In our view, SRU is too expressive, since inconsistent programs cannot always be detected in that form. Moreover, the SRU operator can capture non-polynomial operations, such as the powerset, which complicate query optimization. In fact, to our knowledge, there is no normalization algorithm for SRU forms in general. That is, SRU forms cannot be put in canonical form. On the other hand, $\mathrm{ext}(f)$ is not expressive enough, since it cannot capture operations that involve different collections and it cannot express predicates and aggregates. We believe that our monoid homomorphism algebra is the most expressive subset of SRU where inconsistencies can always be detected at compile time, and, more importantly, where all programs can be put in canonical form.

Monad comprehensions were first introduced by P. Wadler [23] as a generalization of list comprehensions (which have already been used in some functional languages). Monoid comprehensions are related to monad comprehensions but they are considerably more expressive.

In particular, monoid comprehensions can mix inputs from different collection types and may return output of a different type. This is not possible for monad comprehensions, since they restrict the inputs and the output of a comprehension to be of the same type. Monad comprehensions were first proposed as a convenient database language by P. Trinder [20, 19] who also presented many algebraic transformations over these forms as well as methods for converting comprehensions into joins. The monad comprehension syntax was also adopted by P. Buneman and V. Tannen [7] as an alternative syntax to monoid homomorphisms. The comprehension syntax was used for capturing operations that involve collections of the same type while structural recursion was used for expressing the rest of the operations (such as converting one collection type to another, predicates, and aggregates).

Our normalization algorithm is highly influenced by L. Wong's work on normalization of monad comprehensions [24, 25]. He presented some very powerful rules for flattening nested comprehensions into canonical comprehension forms whose generators are over simple paths. These canonical forms are equivalent to our canonical forms for monoid homomorphisms. He used these canonical forms to prove a very important theorem about the expressiveness of query languages that possess the conservative extension property (i.e., languages in which the nesting of the input and output collections in a function cannot exceed a certain level). The theorem indicates that this maximum level of nesting is independent of the nesting level of intermediate data structures used by a function. That is, adding more nesting level to the intermediate data structures does not increase the expressiveness of the language. This theorem is a consequence of the fact that nested intermediate structures can be flattened out from monad comprehensions by the normalization algorithm.

Our object-oriented comprehensions are based on Ohori's work on capturing object identity in a functional language [14]. However, our work is the first one that incorporates this theory into a database calculus. (Ohori applied this theory in the general framework of lambda calculus.) Consequently, we can optimize the resulting second-order programs by using the same optimization techniques used for regular programs without object identity.

9 Conclusion

We have introduced a new calculus for capturing operations over collection types. This calculus is based upon the theory of monoid homomorphisms. It can be used as a database query language as well as a language for expressing data and nested parallelism. It supports an efficient normalization algorithm that improves program performance, eliminates garbage structures, and increases the degree of parallelism. This uniform treatment of collection types and their operations may lead into a new unified theory that generalizes the nested relational algebra.

We have experimented with a prototype query optimizer for the OQL language of the ODMG-93 standard. Our optimizer translates OQL syntax into the monoid calculus syntax, which in turn is translated into an algebraic form. The resulting algebraic expressions are normalized by the optimizer to a canonical form, which is passed to a plan generator that searches the space of alternative access paths and algorithms to retrieve an efficient plan. In addition, the normalizer was extended to support object identity, as it is described in

Section 6.

Acknowledgements: The author is grateful to Dave Maier, Tim Sheard, Val Tannen, Bennet Vance, and Limsoon Wong for helpful comments on the paper. This work is supported by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518.

References

[1] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Carnegie Mellon University, April 1993. CMU-CS-93-129.

[2] G. Blelloch and G. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[3] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 9–19. Morgan Kaufmann Publishers, Inc., August 1991.

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

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

[6] P. Buneman. The Fast Fourier Transform as a Database Query. Technical report, University of Pennsylvania, March 1993. MS-CIS-93-37/L&C 60.

[7] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[8] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[9] L. Fegaras. Efficient Optimization of Iterative Queries. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City*, pp 200–225. Springer-Verlag, Workshops on Computing, August 1993.

[10] L. Fegaras, T. Sheard, and T. Zhou. Improving Programs which Recurse over Multiple Inductive Structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, pp 21–32, June 1994.

[11] L. Fegaras and D. Stemple. Using Type Transformation in Database System Implementation. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 337–353. Morgan Kaufmann Publishers, Inc., August 1991.

[12] W. Hillis and G. Steele. Data Parallel Algorithms. *Communications of ACM*, 29(12), pp 1170–1183, December 1986.

[13] D. Maier and B. Vance. A Call to Order. *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC*, pp 1–16, May 1993.

[14] A. Ohori. Representing Object Identity in a Pure Functional Language. In *International Conference on Database Theory, Paris, France*, pp 41–55. Springer-Verlag, December 1990. LNCS 470.

[15] D. Parker, E. Simon, and P. Valduriez. SVP – a Model Capturing Sets, Streams, and Parallelism. In *Proceedings of the 18th VLDB Conference, Vancouver, Canada*, pp 115–126, August 1992.

[16] D. Schmidt. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[17] T. Sheard and L. Fegaras. A Fold for All Seasons. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 233–242, June 1993.

[18] J. Sipelstein and G. Blelloch. Collection-Oriented Languages. Technical report, Carnegie Mellon University, March 1991. CMU-CS-90-127.

[19] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 55–68. Morgan Kaufmann Publishers, Inc., August 1991.

[20] P. Trinder and P. Wadler. Improving List Comprehension Database Queries. In *in Proceedings of TENCON'89, Bombay, India*, pp 186–192, November 1989.

[21] S. Vandenberg and D. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Denver, Colorado*, 20(2), May 1991.

[22] P. Wadler. The Essence of Functional Programming. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, pp 1–14, January 1992.

[23] P. Wadler. Comprehending Monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pp 61–78, June 1990.

[24] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC*, pp 26–36, May 1993.

[25] L. Wong. *Querying Nested Collections*. PhD thesis, Univerity of Pennsylvania, March 1994. Also appeared as a Univerity of Pennsylvania technical report IRCS Report 94-09.

[26] B. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. Do&Merge: Integrating Parallel Loops and Reductions. In *Sixth International Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon*, pp 169–183. Springer-Verlag, August 1993. LNCS 768.

A The Proof of Theorem 3

Theorem 2 *If* $(T, \mathrm{zero}[T], \mathrm{merge}[T])$ *is a monoid, then* $(\Phi(T), \mathrm{zero}[\Phi(T)], \mathrm{merge}[\Phi(T)])$ *is also a monoid, where*

$$\begin{array}{lcl} \Phi(T) & = & S \rightarrow (T \times S) \\ \mathrm{zero}[\Phi(T)] & = & \lambda s.(\mathrm{zero}[T], s) \\ \mathrm{merge}[\Phi(T)](f, g) & = & (\mathrm{merge}[T] \times \mathrm{id}) \circ \mathrm{distr} \circ (\mathrm{id} \times g) \circ f \end{array}$$

Proof: First we need to prove that $\mathrm{zero}[\Phi(T)]$ is the left and right identity of $\mathrm{merge}[\Phi(T)]$ and that $\mathrm{merge}[\Phi(T)]$ is associative:

$$\begin{array}{ll} 1) & \mathrm{merge}[\Phi(T)](\mathrm{zero}[\Phi(T)], g) \\ & = (\mathrm{merge}[T] \times \mathrm{id}) \circ \mathrm{distr} \circ (\mathrm{id} \times g) \circ \mathrm{zero}[\Phi(T)] \\ & = \lambda s.(\mathrm{merge}[T] \times \mathrm{id})(\mathrm{distr}((\mathrm{id} \times g)(\mathrm{zero}[T], s))) \\ & = \lambda s.(\mathrm{merge}[T] \times \mathrm{id})(\mathrm{distr}(\mathrm{zero}[T], g\, s)) \\ & = \lambda s.(\mathrm{merge}[T] \times \mathrm{id})((\mathrm{zero}[T], \pi_1(g\, s)), \pi_2(g\, s)) \\ & = \lambda s.(\mathrm{merge}[T](\mathrm{zero}[T], \pi_1(g\, s)), \pi_2(g\, s)) \\ & = \lambda s.(\pi_1(g\, s), \pi_2(g\, s)) = \lambda s.(g\, s) = g \\ 2) & \mathrm{merge}[\Phi(T)](f, \mathrm{zero}[\Phi(T)]) \\ & = (\mathrm{merge}[T] \times \mathrm{id}) \circ \mathrm{distr} \circ (\mathrm{id} \times \mathrm{zero}[\Phi(T)]) \circ f \\ & = (\mathrm{merge}[T] \times \mathrm{id}) \circ \mathrm{distr} \circ (\mathrm{id} \times (\lambda s.(\mathrm{zero}[T], s))) \circ f \\ & = (\mathrm{merge}[T] \times \mathrm{id}) \circ (\lambda(a, s).((a, \mathrm{zero}[T]), s)) \circ f \\ & = (\lambda(a, s).(\mathrm{merge}[T](a, \mathrm{zero}[T]), s)) \circ f \\ & = (\lambda(a, s).(a, s)) \circ f = f \end{array}$$

3) We need to prove that $\mathrm{merge}[\Phi(T)](\mathrm{merge}[\Phi(T)](f, g), h) = \mathrm{merge}[\Phi(T)](f, \mathrm{merge}[\Phi(T)](g, h))$. The product of functions $\langle f, g \rangle$ is defined as $\langle f, g \rangle(a) = (f\, a, g\, a)$. It is easy to prove the following properties:

$$\begin{array}{rcl} \langle f \circ \pi_1, g \circ \pi_2 \rangle & = & f \times g \\ \mathrm{distr} & = & \langle \mathrm{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \\ \pi_1 \circ \mathrm{merge}[\Phi(T)](f, g) & = & \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ g)) \circ f \\ \pi_2 \circ \mathrm{merge}[\Phi(T)](f, g) & = & \pi_2 \circ g \circ \pi_2 \circ f \end{array}$$

In addition, we know that $\mathrm{merge}[T]$ is associative, that is, $\mathrm{merge}[T] \circ (\mathrm{merge}[T] \times f) = \mathrm{merge}[T] \circ \langle \pi_1 \circ \pi_1, \mathrm{merge}[T] \circ (\pi_2 \times f) \rangle$.

We have two cases:

$$
\begin{aligned}
3.a)\quad & \pi_2 \circ \mathrm{merge}[\Phi(T)](\mathrm{merge}[\Phi(T)](f,g),h) \\
&= \pi_2 \circ h \circ \pi_2 \circ \mathrm{merge}[\Phi(T)](f,g) \\
&= \pi_2 \circ h \circ \pi_2 \circ g \circ \pi_2 \circ f \\
&= \pi_2 \circ \mathrm{merge}[\Phi(T)](g,h) \circ \pi_2 \circ f \\
&= \pi_2 \circ \mathrm{merge}[\Phi(T)](f,\mathrm{merge}[\Phi(T)](g,h)) \\
3.b)\quad & \pi_1 \circ \mathrm{merge}[\Phi(T)](\mathrm{merge}[\Phi(T)](f,g),h) \\
&= \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ h)) \circ \mathrm{merge}[\Phi(T)](f,g) \\
&= \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ h)) \circ (\mathrm{merge}[T] \times \mathrm{id}) \circ \langle \mathrm{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ (\mathrm{merge}[T] \times (\pi_1 \circ h)) \circ \langle \mathrm{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1 \circ \pi_1, \mathrm{merge}[T] \circ (\pi_2 \times (\pi_1 \circ h)) \rangle \circ \langle \mathrm{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1, \mathrm{merge}[T] \circ (\pi_2 \times (\pi_1 \circ h)) \circ \langle \mathrm{id} \times \pi_1, \pi_2 \circ \pi_2 \rangle \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1, \mathrm{merge}[T] \circ \langle \pi_1 \circ \pi_2, \pi_1 \circ h \circ \pi_2 \circ \pi_2 \rangle \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1, \mathrm{merge}[T] \circ \langle \pi_1, \pi_1 \circ h \circ \pi_2 \rangle \circ \pi_2 \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1, \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ h)) \circ \pi_2 \rangle \circ (\mathrm{id} \times g) \circ f \\
&= \mathrm{merge}[T] \circ \langle \pi_1, \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ h)) \circ g \circ \pi_2 \rangle \circ f \\
&= \mathrm{merge}[T] \circ (\mathrm{id} \times (\mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ h)) \circ g)) \circ f \\
&= \mathrm{merge}[T] \circ (\mathrm{id} \times (\pi_1 \circ \mathrm{merge}[\Phi(T)](g,h))) \circ f \\
&= \pi_1 \circ \mathrm{merge}[\Phi(T)](f,\mathrm{merge}[\Phi(T)](g,h))
\end{aligned}
$$

Therefore, $\mathrm{merge}[\Phi(T)](\mathrm{merge}[\Phi(T)](f,g),h) = \mathrm{merge}[\Phi(T)](f,\mathrm{merge}[\Phi(T)](g,h))$. $\square$