Efficient Optimization of Iterative Queries

Leonidas Fegaras
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
Portland, OR 97291-1000.
fegaras@cse.ogi.edu

Abstract

This paper presents a new query algebra based on *fold* iterations that facilitates database implementation. An algebraic normalization algorithm is introduced that reduces any program expressed in this algebra to a canonical form that generates no intermediate data structures and has no more nested iterations than the initial program. Given any inductive data type, our system can automatically synthesize the definition of the fold operator that traverses instances of this type, and, more importantly, it can produce the necessary transformations for optimizing expressions involving this fold operator.

Database implementation in our framework is controlled by user-defined mappings from abstract types to physical structures. The optimizer uses this information to translate abstract programs and queries into concrete algorithms that conform to the type transformation. Database query optimization can be viewed as a search over the reduced space of all canonical forms which are equivalent to the query after type transformation and normalization. The optimization space can be expanded to capture semantic information expressed as integrity constraints attached to types. This information may include specifications of materialized views and of alternative access paths.

The contribution of this paper is twofold. First, a new efficient algebraic optimization algorithm is introduced, based on loop fusion and partial evaluation, that normalizes a large class of queries over a wide spectrum of bulk data structures. Second, an effective query optimizer is described that searches the limited space of equivalent canonical forms for optimal programs, using additional semantic information to generate more alternatives.

1 Introduction

Database systems based on the relational model offer a high degree of data independence. In these systems the semantics of abstract data structures and their operations can be significantly independent of their implementations. This separation of specification from implementation offers many opportunities for optimization which would otherwise be lost if abstract operations were expressed by a detailed algorithm. For example, queries in the relational model are expressed declaratively, without any concern about efficiency, relying on the query optimizer to select the best evaluation plan among a variety of available access plans and algorithms [13].

Modern database applications require more advanced data structures and more expressive operations than those provided by the relational model. If complex storage structures are mapped into relational tables then semantic information is often lost. For example, information about object interconnection needs to be reconstructed in object queries by using joins, if object hierarchies are mapped into flat tables. This lost information offers more opportunities for optimization when stated explicitly in the schema description, as it would if the model supported these complex data structures directly.

In addition, ordered bulk data structures, such as lists, trees, and arrays, cannot be captured directly in relational tables as they are by definition unordered. This implicit ordering information must be captured explicitly or it may introduce inconsistencies in the resulting programs. On the other hand, if sets are represented as ordered sequences then the optimizer may miss opportunities for optimization. For example, using a commutative union (in a set based implementation) may allow a more efficient translation than using an noncommutative append (in a list based implementation).

These observations can be generalized: when a system is underspecified inconsistencies may be introduced in its translation, while when a system is overspecified optimization opportunities are lost. A specification framework must have enough modeling power to directly capture all semantic information of the system being specified. That way, the program optimizer can use semantic information directly to make intelligent decisions. Consequently, restrictions on data values, such as integrity constraints attached to the database state, should be provided explicitly as part of the schema, so that they may be used to generate alternatives during optimization [15].

There have been many proposals for query algebras that are more expressive than the relational algebra. Much research into these designs has been guided by the belief that the more the expressiveness (functionality) of a language the more difficult the program optimization task becomes. This paper will demonstrate that this belief is not necessarily true by presenting a query algebra that is both expressive enough to capture most polynomial time functions and still facilitates optimization. We believe that the infeasibility of optimization is controlled not by reducing the expressiveness of the language, but by reducing the number of possible program schemes that compute a function. The search for an optimal solution is more efficient if there is a smaller number of program schemes to consider. A provision must be taken, though, that most candidates for optimal solution are always within this search space. Therefore, there is a tension that needs to be resolved: the more programs schemes considered, the better optimized the resulting programs, but the more optimization time spent. In our approach program schemes that cannot be optimal under any circumstances will not be considered. By using an algebraic method that improves programs independently of the database state, we reduce the search space size by avoiding states that correspond to suboptimal programs.

The reduced number of program schemes is achieved by requiring that all structure traversals be expressed in terms of a very small number of stereotyped generic recursion schemes. We have only one traversal mechanism in our algebra: the *fold* operation [14, 8, 7]. Given any inductive data type (e.g. tuple, set, list, tree, boolean, and integer) our system is capable of automatically synthesizing the definition of the fold operator that traverses instances of this type. If some simple, easily recognizable, syntactic restrictions to this algebra

are imposed, then some very nice properties result. For example, any operation expressed in this algebra can always be reduced, by a very simple and efficient algorithm, to a canonical form. We call this algorithm the *normalization algorithm*. This canonical form is, in a way, optimal, since it does not generate any intermediate data structures (which might be generated when function calls are nested, passing intermediate results from one to another). It also has no more nested iterations than the program before the normalization. Note that elimination of intermediate data structures does not necessarily imply optimality. In fact, when we have a duplicate computation it is more efficient to pull out this computation, assign its intermediate result to a variable, and use the value of this variable instead of performing the computation twice. Fortunately, this type of program improvement can be done at any stage either before or after program normalization. We therefore choose to ignore it at this phase.

The normalization algorithm is both a loop fusion method [16, 4] that merges two cascaded iterations[1] into one, and an online partial evaluator [5] that specializes programs with respect to their static input. Based upon a generic *promotion theorem*, the algorithm is aided by the explicit inductive structure of folds rather than searching for implicit structure in an analysis phase, as is done in most program transformation systems [6].

The normalization algorithm devotes the same effort to all traversals, independent of the data structures they traverse. In our model there are no primitive types, such as integers or booleans; all types are user-defined data structures. Therefore, all primitive operations, such as integer addition and boolean conjunction, are computed using fold traversals. In this way all data types and the operations upon them are treated uniformly. Some researchers in the database community believe that a query optimizer should not spend time optimizing non-bulk operations, such as integer and boolean operations, since bulk operations involve heavy I/O use which is considerably slower than the CPU calculations. We believe that this argument is not necessarily true for two reasons. First, non-bulk operations may be mixed with bulk operations, as in a nested SQL statement where a selection predicate contains another SQL statement. Second, rewriting a non-bulk operation may reveal a new transformation for a bulk operation which could otherwise be unavailable. For example, transforming a join predicate into disjunctive normal form may trigger new join transformations.

The normalization algorithm can be used for algebraic optimization in both the abstract and physical layers. In fact, it generalizes many algebraic optimization algorithms found in the database literature [10, 1]. In addition, loop fusion can effectively capture most stream-based pipelining techniques used in relational query evaluation for propagating values a tuple at a time instead of a table at a time between relational operations (this is achieved by fusing the loops of two nested relational operations into one loop). This ability to fuse loops serves as an additional argument for defining relational operators, such as the relational join, not as opaque abstractions that satisfy some properties, but as abstractions with an explicit predefined control structure. That way the control of an operator can be fused with the control of another operator resulting in a more efficient program. Our fold operator can be seen as an

[1] That is, a loop that iterates over the result of another loop, such as two piped filters, not a nested loop in an imperative language.

algebraic formulation of a pipe. The difference is that, in our framework, pipes are constructed during optimization time, not during evaluation time. Thus we avoid the usual overhead of pipe synchronization.

Given a canonical form in our algebra computing a function f, it is easy to discover all other equivalent canonical forms, that is, all canonical forms that compute the same function f. In fact, the canonical forms that are equivalent to f are those that are derived mainly by commutativity laws (since associative forms in our algebra have the same canonical form). Our optimizer considers only the canonical forms equivalent to the one derived by applying the normalization algorithm to the original program that represents a query. Non-canonical programs are not considered at all. This is desirable since they can all be reduced to a more efficient canonical form from this class. This restricts the search for the best evaluation plan to a small, finite, class of programs that contains very few suboptimal programs. Note that by specifying a join as a nested fold, for example, we have not restricted its implementation to nested-loop join, because the optimizer can always discover algorithms, such as the sort-merge join, that have the same functionality as this nested fold but possibly different implementations. Therefore, we gain all the optimization opportunities of a highly declarative algebra in a framework of a more operational language that completely automates algebraic optimization. Furthermore, semantic information, such as integrity constraints, can also be reduced to canonical form and then be used to expand the search space of alternatives, offering different access paths for evaluating the same program. We have reduced the search space by eliminating suboptimal goals, but we have also expanded it by adding more alternatives that correspond to additional semantic information.

One important component of our algebraic query optimizer is *type transformation* [9]. In this framework we specify how an abstract data type is mapped into a concrete type (a data structure in the physical layer) by providing an *abstraction function*. This function, expressed in canonical form, maps any structure of the concrete type into a value of the abstract type. It is in general a many-to-one function. For example, a set can be implemented as an ordered sequence, such as a list, where the abstraction function inserts each element of the list into an initially empty set. Integrity constraints in the physical layer specify materialized views and access paths, such as a secondary indices. Integrity constraints in the logical layer represent conceptual views that may or may not be materialized, depending on the abstraction function and the storage structure they are mapped to. The abstraction functions and the integrity constraints in both the logical and physical layers form the necessary theory that specifies the database implementation. The query optimizer will use this transformation theory to derive efficient algorithms that compute queries as well as to verify the resulting translation.

The physical layer consists of a finite number of prespecified concrete primitives. In our translation framework these concrete primitives are assigned a *behavior* expressed in canonical form. The actual implementation of a concrete operation, such as the implementation for B-tree-find, may be many pages of C code, but its actual behavior can be described by few lines of canonical program. We will assume that the behavior of a concrete primitive is consistent with its implementation without any attempt to prove it. Each concrete primitive is also assigned a cost function that estimates its cost from the cost parameters of its input values, such as set cardinalities. The goal of the optimizer is to find

a composition of concrete primitives such that the composition of their associated behaviors is equal to the canonical program that represents the initial query after type transformation. Furthermore, this composition must have the best cost among all such compositions with the same functionality. We call this algorithm the *fold optimization algorithm.* It is in a sense the inverse of the normalization algorithm, which reduces compositions to canonical form. The optimization algorithm is a search-based algorithm that searches the limited space of canonical forms. It takes into account semantic information, expressed as integrity constraints, to widen the search space, producing alternative access paths and algorithms.

2 Definitions

In this section we define the fold operator for a family of inductively defined algebraic data type. We will analyze sets and set folds in Section 4. There are no primitive types in our model, because integers, booleans and strings are defined in the same way as any bulk data type. This is desirable since we want all operations over any data type to be treated uniformly, in a context of a simple universal optimization algorithm. Our fold operator is similar to, but more expressive than, the *pump* operator [1] and the *structural recursion* operator [3]. The difference is that our system can derive the fold definition for most data types automatically by examining the type details.

The type definitions considered in this section are the simple sums-of-product types defined by using recursive equations of the form:

$$\begin{array}{rcl} T(\alpha_1,\ldots,\alpha_p) & = & C_1(s_{1,1}:t_{1,1},\ldots,s_{1,m_1}:t_{1,m_1}) \\ & & \mid \cdots \\ & & \mid C_n(s_{n,1}:t_{n,1},\ldots,s_{n,m_n}:t_{n,m_n}) \end{array}$$

where $\alpha_1,\ldots,\alpha_p$ denote type variables, the C_i are unique names of value constructor functions (we use the convention that the first letter of a constructor name is always in uppercase), $s_{i,j}$ are names of selector functions, $t_{i,j}$ are either type variables (in the set $\alpha_1,\ldots,\alpha_p$) or the type $T(\alpha_1,\ldots,\alpha_p)$ itself. For example, the following are simple sums-of-products type definitions:

$$\begin{array}{rcl} \text{boolean} & = & \text{False} \mid \text{True} \\ \text{prod}(\alpha,\beta) & = & \text{Pair}(\text{fst}:\alpha,\text{snd}:\beta) \\ \text{list}(\alpha) & = & \text{Nil} \mid \text{Cons}(\text{hd}:\alpha,\text{tail}:\text{list}(\alpha)) \\ \text{nat} & = & \text{Zero} \mid \text{Succ}(\text{pred}:\text{nat}) \\ \text{tree}(\alpha) & = & \text{Tip}(\text{info}:\alpha) \mid \text{Node}(\text{left}:\text{tree}(\alpha),\text{right}:\text{tree}(\alpha)) \\ \text{person} & = & \text{Make\_person}(\text{name}:\text{string},\text{ssn}:\text{nat},\text{address}:\text{string}) \\ & & \text{where string} = \text{list}(\text{nat}) \end{array}$$

Before expressing the general definition of the fold operator for any sums-of-products type, we give the definition of the *list* fold as an example. If a function $g:\text{list}(\alpha)\rightarrow\beta$ is defined by the following recursive equations:

$$\begin{array}{rcl} g(\text{Nil}) & = & f_n() \\ g(\text{Cons}(a,l)) & = & f_c(a,g(l)) \end{array}$$

for some functions f_n and f_c, then g is a list fold: $g = \text{fold}^{list}(f_n, f_c)$. Consider for example the length of a list computed by the following equations:

$$\begin{array}{lcl} \text{length(Nil)} & = & \text{Zero} \\ \text{length(Cons}(a, l)) & = & \text{Succ(length}(l)) \end{array}$$

In this case, $f_n() = \text{Zero} \Rightarrow f_n = \lambda().\text{Zero}$ and $f_c(a, r) = \text{Succ}(r) \Rightarrow f_c = \lambda(a, r).\text{Succ}(r)$[2]. Therefore,

$$\text{length}(x) \;=\; \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r))\; x$$

Note that f_c in $\text{fold}^{list}(f_n, f_c)$ has the form $\lambda(a, r).e$, which is a lambda abstraction that has two parameters: a, bound to the current head of the list and r, bound to the recursively transformed tail of the list. That is, if the output type of the *list* fold is β then the type of r is also β.

In general, the fold operation over a sums-of-products type will use a pattern of recursion related to the pattern of recursion in the type definition. This recursion pattern is captured by the functor E. Functor E, like functors in category theory, satisfies $E(\text{id}) = \text{id}$ and $E(g) \circ E(h) = E(g \circ h)$, where g and h are functions, $\circ$ is function composition, and id is the identity function (i.e. $\text{id}(x) = x$). These properties can be easily verified for our definition below.

Definition 1 (The Functor E) *Associated with each constructor C_i of type $(t_{i,1}, \ldots, t_{i,m_i}) \rightarrow T(\alpha_1, \ldots, \alpha_p)$ is a monadic functor:*

$$E_i^T(f) \;=\; \lambda(x_{i,1}, \ldots, x_{i,m_i}).(K(x_{i,1}), \ldots, K(x_{i,m_i}))$$

where the bound variables $x_{i,j}$ have type $t_{i,j}$ and $K(x_{i,j})$ is either $f(x_{i,j})$, if $t_{i,j} = T(\alpha_1, \ldots, \alpha_p)$, or $x_{i,j}$, otherwise.

For example, for type *list*: $E_{Nil}^{list}(f) = \lambda().()$ and $E_{Cons}^{list}(f) = \lambda(x, y).(x, f(y))$.

With this notation it is now possible to describe the *fold* operator for any simple sums-of-products type.

Definition 2 (Fold) *The fold function over $T(\alpha_1 \ldots \alpha_p)$ is defined by the following set of recursive equations, one for each constructor C_i of T:*

$$\text{fold}^T(\overline{f}) \circ C_i \;=\; f_i \circ E_i^T(\text{fold}^T(\overline{f}))$$

where $\overline{f} = (f_1, \ldots, f_n)$.

For example, the *list* fold is defined recursively by the following two equations (in applicative form):

$$\begin{array}{lcl} \text{fold}^{list}(f_n, f_c)\,\text{Nil} & = & f_n() \\ \text{fold}^{list}(f_n, f_c)\,\text{Cons}(a, l) & = & f_c(a, \text{fold}^{list}(f_n, f_c)\, l) \end{array}$$

The *nat* fold is defined as:

$$\begin{array}{lcl} \text{fold}^{nat}(f_z, f_s)\,\text{Zero} & = & f_z() \\ \text{fold}^{nat}(f_z, f_s)\,\text{Succ}(i) & = & f_s(\text{fold}^{nat}(f_z, f_s)\, i) \end{array}$$

[2] $\lambda(x_1, \ldots, x_n).e$ is a lambda abstraction, expressed as $\lambda x_1 \ldots . \lambda x_n . e$ in lambda calculus, and $(e_1, \ldots, e_n)$ constructs a tuple of n values. Application $f(e_1, \ldots, e_n)$ can be seen as the application of the unary function f to the tuple $(e_1, \ldots, e_n)$.

The *tree* fold is defined as:

$$\begin{array}{lcl} \text{fold}^{tree}(f_t, f_n)\,\text{Tip}(a) & = & f_t(a) \\ \text{fold}^{tree}(f_t, f_n)\,\text{Node}(l, r) & = & f_n(\text{fold}^{tree}(f_t, f_n)\,l, \text{fold}^{tree}(f_t, f_n)\,r) \end{array}$$

The *boolean* fold is defined as:

$$\begin{array}{lcl} \text{fold}^{boolean}(f_f, f_t)\,\text{False} & = & f_f() \\ \text{fold}^{boolean}(f_f, f_t)\,\text{True} & = & f_t() \end{array}$$

Note that $\text{fold}^{boolean}(f_f, f_t)\,x \equiv \textbf{if}\ x\ \textbf{then}\ f_t()\ \textbf{else}\ f_f()$.

We call each f_i in $\overline{f}$ an *accumulating function*. Note that if each $f_i = C_i$ then $\text{fold}^T(\overline{f}) = \text{id}$.

The following are some examples of computations that can be defined using fold functions:

$$\begin{array}{lcl} \text{append}(x, y) & = & \text{fold}^{list}(\lambda().y, \lambda(a, r).\text{Cons}(a, r))\ x \\ \text{map}(x, f) & = & \text{fold}^{list}(\lambda().\text{Nil}, \lambda(a, r).\text{Cons}(f(a), r))\ x \\ \text{sum}(x) & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a, r).a + r)\ x \\ \text{reverse}(x) & = & \text{fold}^{list}(\lambda().\text{Nil}, \lambda(a, r).\text{append}(r, \text{Cons}(a, \text{Nil})))\ x \\ x + y & = & \text{fold}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\ x \\ x \times y & = & \text{fold}^{nat}(\lambda().\text{Zero}, \lambda(r).y + r)\ x \\ \text{even}(x) & = & \text{fold}^{nat}(\lambda().\text{True}, \lambda(r).\text{not}(r))\ x \\ x \wedge y & = & \text{fold}^{boolean}(\lambda().\text{False}, \lambda().y)\ x \\ \text{if } x \text{ then } y \text{ else } z & = & \text{fold}^{boolean}(\lambda().z, \lambda().y)\ x \\ \text{flat}(x) & = & \text{fold}^{tree}(\lambda(i).\text{Cons}(i, \text{Nil}), \lambda(l, r).\text{append}(l, r))\ x \\ p.\text{address} & = & \text{fold}^{person}(\lambda(n, s, a).a)\ p \end{array}$$

The general law which applies to all fold functions is called the *promotion theorem*. We will use this theorem as a major component of our automated transformation algorithm, called the *normalization algorithm*. The promotion theorem states that the composition of any function g with some fold is another fold whose accumulating functions are related to the accumulating functions of the original fold by fixed equations. The normalization algorithm will describe how these new accumulating functions can be calculated. The promotion theorem for folds has appeared in the literature using various notations [11, 12]:

Theorem 1 (The Fold Promotion Theorem)

$$\forall i:\ \phi_i \circ E_i^T(g)\ =\ g \circ f_i\ \ \Rightarrow\ \ g \circ \text{fold}^T(\overline{f})\ =\ \text{fold}^T(\overline{\phi})$$

Proof: Let $\eta = g \circ \text{fold}^T(\overline{f})$ and C_i a constructor of T. Then $\eta \circ C_i = g \circ \text{fold}^T(\overline{f}) \circ C_i = g \circ f_i \circ E_i^T(\text{fold}^T(\overline{f})) = \phi_i \circ E_i^T(g) \circ E_i^T(\text{fold}^T(\overline{f})) = \phi_i \circ E_i^T(g \circ \text{fold}^T(\overline{f})) = \phi_i \circ E_i^T(\eta)$. Therefore, $\eta \circ C_i = \phi_i \circ E_i^T(\eta)$, which shows that η has a form of a fold with accumulating functions $\overline{\phi}$ (by Definition 2). Thus $\eta = \text{fold}^T(\overline{\phi})$. □

For example, the fold promotion theorem for *list* is:

$$\frac{\begin{array}{lcl} \phi_n() & = & g(f_n()) \\ \phi_c(a, g(r)) & = & g(f_c(a, r)) \end{array}}{g(\text{fold}^{list}(f_n, f_c)\,x)\ =\ \text{fold}^{list}(\phi_n, \phi_c)\,x}$$

For example, the boolean promotion theorem is:

$$g \circ \text{fold}^{boolean}(f_f, f_t) \;=\; \text{fold}^{boolean}(g \circ f_f, g \circ f_t)$$

The normalization algorithm which is based on the promotion theorems is described in detail in Section 3.2. Here we give an example of using the list promotion theorem for normalizing programs. We will improve filter(filter$(x,p),q)$, where filter(x,p) returns the list of all elements in the list x that satisfy the predicate p:

$$\text{filter}(x,p) \;=\; \text{fold}^{list}(\lambda().\text{Nil}, \lambda(a,r).\textbf{if}\, p(a)\, \textbf{then}\, \text{Cons}(a,r)\, \textbf{else}\, r)\, x$$

We need to find some $\text{fold}^{list}(\phi_n, \phi_c)\, x = \text{filter}(\text{filter}(x,p),q)$. We apply the *list* promotion theorem with $g(x) = \text{filter}(x,q)$ and $\text{fold}^{list}(f_n, f_c)\, x = \text{filter}(x,p)$:

1) $\phi_n() \;=\; g(f_n()) \;=\; \text{filter}(f_n(), q) \;=\; \text{filter}(\text{Nil}, q) \;=\; \text{Nil}$

2) $\phi_c(a, \text{filter}(r,q)) \;=\; g(f_c(a,r)) \;=\; \text{filter}(f_c(a,r), q)$
$= \text{filter}(\textbf{if}\, p(a)\, \textbf{then}\, \text{Cons}(a,r)\, \textbf{else}\, r, q)$
$= \textbf{if}\, p(a)\, \textbf{then}\, \text{filter}(\text{Cons}(a,r), q)\, \textbf{else}\, \text{filter}(r,q)$
by the boolean promotion theorem
$= \textbf{if}\, p(a)\, \textbf{then}\, (\textbf{if}\, q(a)\, \textbf{then}\, \text{Cons}(a, \text{filter}(r,q))\, \textbf{else}\, \text{filter}(r,q))$
$\textbf{else}\, \text{filter}(r,q)$
by the filter definition
$\Rightarrow\; \phi_c(a,u) \;=\; \textbf{if}\, p(a)\, \textbf{then}\, (\textbf{if}\, q(a)\, \textbf{then}\, \text{Cons}(a,u)\, \textbf{else}\, u)\, \textbf{else}\, u$
where filter(r,q) *was generalized to* u

Therefore, the composition filter(filter$(x,p),q)$ is:

$$\text{fold}^{list}(\lambda().\text{Nil}, \lambda(a,u).\textbf{if}\, p(a)\, \textbf{then}\, (\textbf{if}\, q(a)\, \textbf{then}\, \text{Cons}(a,u)\, \textbf{else}\, u)\, \textbf{else}\, u)\, x$$

which is equal to filter$(x, \lambda(a).p(a) \wedge q(a))$.

The following corollary says that there is a unique way of expressing a function as a fold [12]. It is used for testing the functional equality of two folds:

Corollary 1 (Uniqueness Property)

$$\forall i:\; g \circ C_i = \phi_i \circ E_i^T(g) \quad \Leftrightarrow \quad g = \text{fold}^T(\overline{\phi})$$

3 Normalization of Expressions with Folds

The normalization algorithm presented in this section is a reduction algorithm that improves any program which is expressed only in terms of folds. This algorithm reduces any fold applied to another fold into a fold applied to a variable. Since every fold builds a data structure, improved programs will build fewer intermediate data structures.

Program normalization is accomplished by pushing the outer fold into the accumulating functions of the inner fold, as is directed by the promotion theorem. This is a generalization of loop fusion to arbitrary types because the outer fold will be pushed inside the inner one until it is eliminated by the generalizations introduced by the normalization algorithm.

3.1 The Term Language

In the following definitions we assume that programs are well-typed.

Definition 3 (The Term Language) *A* program *in the term language has the form* $\lambda(x_1, \ldots, x_n).\tau$*, where each* x_i *is a variable and* τ *is a term. Each term has one of the following forms:*

- variable: x*, bound in some outer lambda abstraction;*
- construction: $C(\tau_1, \ldots, \tau_n)$*, where* C *is a constructor and each* τ_i *is a term;*
- fold: $\text{fold}^T(f_1, \ldots, f_n)\,\tau$*, where* τ *is a term and each* f_i *has the form* $\lambda(x_1, \ldots, x_m).\tau_i$*, where* τ_i *is a term and each* x_j *is a variable.*

Definition 4 (Accumulative Result Variable) *Each accumulating function* f_i *in a fold in the term language has the form* $\lambda(x_1, \ldots, x_m).\tau$*, where the types of the bound variables* $x_1, \ldots, x_m$ *are associated with the types* $t_1, \ldots, t_m$ *in the domain of the corresponding constructor* C_i*. Each bound variable* x_j *whose associated type* t_j *is the recursive type* T *is an* accumulative result variable*.*

For example, in $\text{append}(x, y) = \text{fold}^{list}(\lambda().y, \lambda(a, r).\text{Cons}(a, r))\,x$, variable r is an accumulative result variable.

Definition 5 (Safe Program) *A program in the term language is safe if it does not contain terms* $\text{fold}^T(\overline{f})\,t$*, where* t *contains a reference to an accumulative result variable which is bound in an outer lambda abstraction.*

This basically says that the partial results of iterations (i.e. the values of the accumulative result variables) are black boxes (they cannot be traversed).

For example, reverse(x), defined in Section 2, is computed by

$$\text{fold}^{list}(\lambda().\text{Nil}, \lambda(a, r).\text{fold}^{list}(\lambda().\text{Cons}(a, \text{Nil}), \lambda(b, s).\text{Cons}(b, s))\,r)\,x$$

is not safe, since the inner fold (that computes $\text{append}(r, \text{Cons}(a, \text{Nil}))$) is over r, an accumulative result variable.

Definition 6 (Canonical Terms) *A canonical term is a term in which*

- *all folds in the term are over variables;*
- *none of these variables are accumulative result variables.*

Note that a canonical term is a safe term that does not produce any intermediate results. We will show that if a program in our term language is safe then it can be transformed into a canonical program by the normalization algorithm.

Even though the restriction that the intermediate results of recursion (i.e. the values of accumulative result variables) are not allowed to be traversed limits the expressiveness of safe programs, there are many useful programs that are still expressible. In fact, in [2], a language that poses a similar restriction was proved to capture all polynomial-time programs[3].

[3] Our language cannot be exactly PTIME since we have a decision algorithm for equalities (see Section 5) and it is well-known that function equality is undecidable for this class.

3.2 The Normalization Algorithm

Before expressing the normalization algorithm in detail, we give one simple example that illustrates how it works. We will improve length(append(x, y)), where:

$$\begin{array}{lcl} \text{length}(x) & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r))\, x \\ \text{append}(x, y) & = & \text{fold}^{list}(f_n, f_c)\, x \quad \text{where} \left\{ \begin{array}{l} f_n = \lambda().y \\ f_c = \lambda(a, r).\text{Cons}(a, r) \end{array} \right. \end{array}$$

We need to find some $\text{fold}^{list}(\phi_n, \phi_c)\, x$ equivalent to length(append(x, y)). We apply the *list* promotion theorem with $g =$ length:

$$\begin{array}{lcl} \phi_n() & = & \text{length}(f_n()) = \text{length}(y) \\ \phi_c(a, \text{length}(r)) & = & \text{length}(f_c(a, r)) \\ & = & \text{length}(\text{Cons}(a, r)) \\ & = & \text{Succ}(\text{length}(r)) \quad \textit{by length definition} \\ & \Rightarrow & \phi_c(a, u) = \text{Succ}(u) \\ & & \textit{where } \text{length}(r) \textit{ was generalized to } u \end{array}$$

Therefore, the composition length(append(x, y)) is:

$$\text{fold}^{list}(\lambda().\text{length}(y), \lambda(a, u).\text{Succ}(u))\, x$$

Note that the intermediate list structure append(x, y) is no longer produced.

The normalization algorithm is a meaning preserving transformation from a term to another term. It uses a parameter ρ which is a partial function from terms to variables. In our notation, $\rho[g/r]$ extends ρ with the mapping from g to r. Function ρ keeps all bindings, such as length$(r)/u$ in the previous example, to be used for eliminating all calls to g from the premise of the promotion theorem, that is, from $\phi_i \circ E_i^T(g) = g \circ f_i$, thus calculating an expression for the accumulating function ϕ_i. This is called the *generalization phase*. This generalization derives the fixpoint of the composition given the fixpoints of the components. We will prove that such a generalization is always possible for all safe terms.

The normalization algorithm consists of the following parts:

- *Generalization:* If the normalization algorithm derives a term mapped in ρ to some variable v, then this term is replaced by v.

- *Application to a Construction:* From the fold definition:

  $$\text{fold}^T(\overline{f})(C_i(\overline{u})) = f_i(E_i^T(\text{fold}^T(\overline{f}))(\overline{u}))$$

  For example, for length $= \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r))$:

  $$\text{length}(\text{Cons}(a, l)) = \text{Succ}(\text{length}(l))$$

- *Fold Promotion:* If the term is a composition $g(\text{fold}^T(\overline{f})\, x)$, where g is a fold, then the fold promotion theorem is applied to derive the term

$\text{fold}^T(\overline{\phi})\,x$, where, for all i, ϕ_i is computed by recursively improving the equation:

$$\phi_i(r_1,\ldots,r_{m_i}) \;=\; g(f_i(x_1,\ldots,x_{m_i}))$$

where all x_i and r_i are new variables. In each case, ρ is extended with the mappings from terms in $(E_i(g)(x_1,\ldots,x_{m_i}))$ to variables $r_1,\ldots,r_{m_i}$.

For example, from the fold promotion theorem for lists we have:

$$g(\text{fold}^{list}(f_n, f_c)\,x) \;=\; \text{fold}^{list}(\phi_n,\phi_c)\,x$$

where

$$\begin{array}{lcl} \phi_n() & = & g(f_n()) \\ \phi_c(r_1,r_2) & = & g(f_c(x_1,x_2)) \quad \text{with } \rho[x_1/r_1,\, g(x_2)/r_2] \end{array}$$

For example, the following improves $\text{plus}(\text{length}(x),\text{length}(y))$, where

$$\text{plus}(x,y) \;=\; \text{fold}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\,x$$

Let $\text{plus}(\text{length}(x),\text{length}(y)) \;=\; \text{fold}^{list}(\phi_n,\phi_c)\,x$. We apply the promotion theorem for *list* with

$$\begin{array}{lcl} g(z) & = & \text{plus}(\text{length}(y), z) \\ & = & \text{fold}^{nat}(\lambda().\text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\,y, \\ & & \quad \lambda(r).\text{Succ}(r))\,z \\ \text{fold}^{list}(f_n,f_c)\,x & = & \text{length}(x) \\ & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\,x \end{array}$$

to compose $g(\text{length}(x)) = \text{fold}^{list}(\phi_n,\phi_c)\,x$:

$$\begin{array}{lcll} \phi_n() & = & g(f_n()) & \\ & = & g(\text{Zero}) & \\ & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\,y & \\ \phi_c(r_1,r_2) & = & g(f_c(x_1,x_2)) & (\textit{where } \rho = [x_1/r_1,\, g(x_2)/r_2]) \\ & = & g(\text{Succ}(x_2)) & \\ & = & \text{Succ}(g(x_2)) & (\textit{application of } g \textit{ to a constructor}) \\ & = & \text{Succ}(r_2) & (g(x_2) \textit{ was generalized to } r_2) \end{array}$$

Therefore, $\text{plus}(\text{length}(x),\text{length}(y))$ is:

$$\text{fold}^{list}(\lambda().\text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\,y, \\ \lambda(r_1,r_2).\text{Succ}(r_2))\,x$$

which is equal to $\text{length}(\text{append}(x,y))$ (presented in the beginning of this subsection).

Theorem 2 (Correctness of the Normalization Algorithm) *The normalization algorithm transforms any safe term into a canonical form.*

Proof: We can see from the definitions of the term language and canonical forms that folds of the form $g(t) = \text{fold}^T\,(\overline{f})\,t$ need to be rewritten into folds over variables. If t is a construction, then we apply the application-to-a-construction rule. In that case the fold g is pushed inside to only those components of the construction that have recursive type. If t is a fold, then g is pushed into the accumulators of the inner fold, as is directed by the fold promotion theorem, and some new mappings are attached to ρ. After recursively applying the normalization algorithm only terms of the form $g(x)$, where x is a variable, remain. If x is an accumulative result variable, then, from the way ρ was extended during promotion, $g(x)$ is always bound in ρ to some new variable r. This is the generalization phase of the algorithm. If x is not an accumulative result variable, then $g(x)$ remains as is. Therefore, all folds in the resulting term are over non-accumulative result variables. It is necessary to prove that after fold promotion there remain no references of the old accumulative result variables other than those generalized by ρ. This is true only when g is a safe fold, that is, when the term t does not contain accumulative result variables. □

Note that the composition of two or more canonical terms is a safe term and, therefore, it can be reduced to canonical form. This makes our canonical term language closed under composition.

The normalization algorithm can be implemented very efficiently. Whenever we apply the promotion theorem we annotate each function g (the left part of the application) with a new number. Then, instead of inserting $g(x_i)/r_i$ in ρ, for some variables x_i and r_i, we can insert the triple of this number with x_i and r_i. Then the generalization phase is performed by checking if the current term $f(x)$ was annotated by a number already in ρ. That way, the complexity of the normalization algorithm is $\mathcal{O}(n \log n)$, where n is the size of the resulting canonical term (since a list of numbers can be searched in $\mathcal{O}(\log n)$). The size of the resulting term, though, can be exponential to the size of the initial term, such as in the case a fold is applied to a deeply nested construction[4]. This is a problem that all partial evaluators face.

The normalization algorithm can capture most stream-based pipelining techniques found in relational query optimization. These techniques model database operations as threads that propagate tuples or partitions of tuples to each other, instead of materializing intermediate relations. This can be achieved in our framework by fusing the loops of the nested relational operators into one loop using the normalization algorithm. Our method, though, is algebraic, fully automated, and it can be performed at any level and for any data structure. Note that the cases that we are forced to materialize results during pipelining, are the cases that require some special operators, such as the sort operator before a sort-merge join. These operations reflect the non-improvable terms in our algebra. Our work states this condition explicitly as a syntactic restrictions to program schemes.

[4]This does not contradict with the fact that the resulting programs are always more efficient than the initial, since folds are lazily evaluated, that is, only one accumulator function is evaluated each time.

4 Finite Sets

Finite sets cannot be captured as regular sums-of-products types since they must satisfy some special properties, such as, being independent of the order in which elements are inserted. All the type constructors described so far form free algebras, in which there is a unique way of constructing an instance of a type. Sets are very important abstractions in database specification as they offer high degree of data independence. They are also important in database implementation because operations on sets have more alternative translations than the operations on free algebra types, therefore they offer more opportunities for optimization, yielding more efficient programs. In spite of that, sets must not be considered as data structures suitable for all cases, as systems based on the relational model do. If there is some additional information on a collection, such as an order of some kind, then an ordered data structure may be more suitable than sets. Furthermore, if there is some dependency between two sets, then a hierarchical structure, such as sets of sets, may be more natural for expressing programs over these structures. The latter case does not imply that these structures can not be actually implemented as flat tables. As we will see in Section 7, abstract modeling can be effectively separated from implementation details. Therefore, the abstract specification of a program should capture as much as possible the semantic information of the problem domain and ignore implementation issues. The program optimizer must be able to use this semantic information to make intelligent decisions.

Finite sets are expressed as instances of a special built-in type $\mathrm{set}(\alpha)$. The set type has two constructors: Emptyset that returns an empty set and $\mathrm{Insert}(e, s)$ that constructs a new set by inserting the element e into the set s. (The type of Insert is $\alpha \times \mathrm{set}(\alpha) \to \mathrm{set}(\alpha)$.) Note that $\mathrm{Insert}(a, \mathrm{Insert}(b, s)) = \mathrm{Insert}(b, \mathrm{Insert}(a, s))$ and that if e is already in s then $\mathrm{Insert}(e, s) = s$. The selector function $\mathrm{split}(s)$ is a non-deterministic function that splits the non-empty input set s into an element and a set. It returns a pair that contains an arbitrary element of s and the set equal to s with this chosen element removed. That is, if $(a, r) = \mathrm{split}(s)$ then $a \notin r$ and $\{a\} \cup r = s$.

A function suitable for traversing sets is set fold (similar to list fold):

Definition 7 (Set Fold) *The set fold* $\mathrm{fold}^{set}(f_e, f_s)$ *is defined by the following recursive equation:*

$$\mathrm{fold}^{set}(f_e, f_s)\, s \quad = \quad \left\{ \begin{array}{l} \textbf{if } s = \mathrm{Emptyset} \textbf{ then } f_e() \\ \textbf{else let } (a, r) = \mathrm{split}(s) \textbf{ in } f_s(a, \mathrm{fold}^{set}(f_e, f_s)\, r) \end{array} \right.$$

For example,

$$\begin{array}{lcl} \mathrm{emptyp}(x) & = & \mathrm{fold}^{set}(\lambda().\mathrm{True}, \lambda(a, r).\mathrm{False})\ x \\ \mathrm{union}(x, y) & = & \mathrm{fold}^{set}(\lambda().y, \lambda(a, r).\mathrm{Insert}(a, r))\ x \\ \mathrm{member}(e, x) & = & \mathrm{fold}^{set}(\lambda().\mathrm{False}, \lambda(a, r).(a = e)\ \textbf{or}\ r)\ x \\ \mathrm{restrict}(s, f) & = & \mathrm{fold}^{set}(\lambda().\mathrm{Emptyset}, \\ & & \quad \lambda(a, r).\textbf{if } f(a) \textbf{ then } \mathrm{Insert}(a, r) \textbf{ else } r)\ s \end{array}$$

Set equality of two sets x and y is computed by the following:

$$\mathrm{set\_equal}(x, y) \quad = \quad \left\{ \begin{array}{l} \mathrm{fold}^{set}(\lambda().\mathrm{fold}^{set}(\lambda().\mathrm{True}, \\ \qquad \lambda(b, s).\mathrm{member}(b, x)\ \textbf{and}\ s)\, y, \\ \quad \lambda(a, r).\mathrm{member}(a, y)\ \textbf{and}\ r)\, x \end{array} \right.$$

Another example is the generic join join$(x, y, match, concat)$ (it is generic enough to have selection and projection embedded in its input functions):

$$\begin{array}{l}\text{fold}^{set}(\lambda().\text{Emptyset} \\ \quad \lambda(a,r).\text{fold}^{set}(\lambda().r, \lambda(b,s).\textbf{if } match(a,b) \\ \qquad\qquad\qquad\qquad \textbf{then } \text{Insert}(concat(a,b),s) \\ \qquad\qquad\qquad\qquad \textbf{else } s)\, y)\, x\end{array}$$

join$(x, y, match, concat)$ takes every combination of elements a and b from the sets x and y, checks whether these elements satisfy the *match* predicate, and if so it returns a new element by applying the *concat* function to form a new element of the result set. An example call to this join retrieves all employees working in the CSE department (this is a semijoin):

join(employees, departments,
λ(emp,dept).(emp.dno=dept.dno **and** dept.name="CSE"),
λ(emp,dept).emp)

A sufficient condition for a set fold being order-independent is f_s being both commutative and idempotent [3]:

Definition 8 (Commutative-idempotent function)
A function f is commutative-idempotent if:

$$\begin{array}{lrcll}\forall m\,\forall n\,\forall s: & f(m,f(n,s)) & = & f(n,f(m,s)) & (\textbf{commutativity}) \\ \forall n\,\forall s: & f(n,f(n,s)) & = & f(n,s) & (\textbf{idempotence})\end{array}$$

For example, Insert is commutative-idempotent, while Cons is not.

The commutativity and the idempotence property of a function f can be verified by the equality decision algorithm described in Section 5.

Theorem 3 *If a function f_s is commutative-idempotent then*

$$\text{fold}^{set}(f_e, f_s)\,(\text{Insert}(a,s)) \;=\; f_s(a, \text{fold}^{set}(f_e, f_s)\,s)$$

Proof: If $s = \text{Emptyset}$ then $\text{fold}^{set}(f_e, f_s)\,(\text{Insert}(a, \text{Emptyset})) = f_s(a, f_e)$ which is true (because $\text{split}(\text{Insert}(a, \text{Emptyset})) = (a, \text{Emptyset})$). If $a \in s$ then $\text{Insert}(a,s) = s$ and $f_s(a, \text{fold}^{set}(f_e, f_s)\,s) = \text{fold}^{set}(f_e, f_s)\,s$ (idempotence property). Otherwise, let $(b,r) = \text{split}(\text{Insert}(a,s))$. We assume that the theorem is true for $m = s - \{a,b\}$: $\text{fold}^{set}(f_e, f_s)\,(\text{Insert}(e,m)) = f_s(e, \text{fold}^{set}(f_e, f_s)\,m)$. Then $\text{fold}^{set}(f_e, f_s)\,(\text{Insert}(a,s)) = f_s(b, \text{fold}^{set}(f_e, f_s)\,r)$ $= f_s(b, \text{fold}^{set}(f_e, f_s)\,\text{Insert}(a,m)) = f_s(b, f_s(a, \text{fold}^{set}(f_e, f_s)\,m))$ (hypothesis) $= f_s(a, f_s(b, \text{fold}^{set}(f_e, f_s)\,m))$ (commutativity property) $= f_s(a, \text{fold}^{set}(f_e, f_s)\,s)$ (hypothesis). □

Theorem 4 (The Promotion Theorem for Set Folds)

$$\frac{\begin{array}{lcl}\phi_e() & = & g(f_e()) \\ \phi_s(a, g(r)) & = & g(f_s(a,r))\end{array}}{g(\text{fold}^{set}(f_e, f_s)\,x) \;=\; \text{fold}^{set}(\phi_e, \phi_s)\,x}$$

and if f_s is commutative-idempotent then so is ϕ_s.

Proof: For $x =$ Emptyset the theorem is true, since $\phi_e() = g(f_e())$. Otherwise, let $(a, r) = \text{split}(x)$. We assume that the theorem is true for $x = r$. Then we have $g(\text{fold}^{set}(f_e, f_s)\, x) = g(f_s(a, \text{fold}^{set}(f_e, f_s)\, r)) = \phi_s(a, g(\text{fold}^{set}(f_e, f_s)\, r))$ $= \phi_s(a, \text{fold}^{set}(\phi_e, \phi_s)\, r) = \text{fold}^{set}(\phi_e, \phi_s)\, x$. In addition, $\phi_s(a, \phi_s(b, g(r))) = \phi_s(a, g(f_s(b, r))) = g(f_s(a, f_s(b, r)))$. Let f_s be a commutative-idempotent function. Then $\phi_s(a, \phi_s(a, g(r))) = g(f_s(a, f_s(a, r))) = g(f_s(a, r)) = \phi_s(a, g(r))$ and $\phi_s(a, \phi_s(b, g(r))) = g(f_s(a, f_s(b, r))) = g(f_s(b, f_s(a, r))) = \phi_s(b, \phi_s(a, g(r)))$. Therefore, ϕ_s is commutative-idempotent too. □

The normalization algorithm does not need any extensions to handle set folds, since the application-to-a-construction rule can be used as is for sets and the fold promotion theorem for sets is exactly like any fold promotion theorem in the free algebra. The condition f_s being an commutative-idempotent can be seen as an additional condition for a set fold being safe. The normalization algorithm can now improve any safe program that involves sets, lists, trees, tuples, integers, booleans etc.

The normalization algorithm captures many types of algebraic transformations already in use in relational algebra. For example, the following query:

```
restrict(join(employees, departments,
              λ(emp,dept).(emp.dno=dept.dno),
              λ(emp,dept).(emp,dept)),
         λ(emp,dept).dept.name="CSE")
```

is reduced by the normalization algorithm into a canonical form which is equivalent to:

```
join(employees, departments,
     λ(emp,dept).(emp.dno=dept.dno and dept.name="CSE"),
     λ(emp,dept).(emp,dept))
```

that is, the selection was pushed inside the join. Note that, if you have a good index to evaluate the join and if the result of the join is very small, it might be preferable not to push the selection inside the join. But there is some information missing in the above query. Our normalization algorithm always improves programs if these programs are evaluated as folds *directly*. That is, when select and join in our example are evaluated naively as loops. We will see in Section 7 that normalization should happen after type transformation. If there is an index available in the physical representation of the join inputs then this information will appear as part of the transformed query. That way path selection can be integrated neatly with algebraic optimization.

As another example, consider the following nested SQL query:

```
select * from d departments
where d.name="CSE"
      and exists( select * from e employees
                  where e.dno=d.dno and e.age>65 )
```

This query can be computed by the following safe program:

```
restrict(departments,
         λ(d).d.name="CSE"
              and (not emptyp(restrict(employees,
                                       λ(e).e.dno=d.dno and e.age>65))))
```

which is normalized into the following canonical form:

$$
\begin{array}{l}
\text{fold}^{set}(\lambda().\text{Emptyset},\\
\quad \lambda(d,r).\textbf{if } d.\text{name} = \text{“CSE”}\\
\quad\quad \textbf{then } \text{fold}^{set}(\lambda().r,\\
\quad\quad\quad\quad \lambda(e,s).\textbf{if } e.\text{dno} = d.\text{dno}\\
\quad\quad\quad\quad\quad\quad \textbf{then if } e.\text{age} > 65 \textbf{ then } \text{Insert}(e,s) \textbf{ else } s\\
\quad\quad\quad\quad\quad\quad \textbf{else } s)\\
\quad\quad\quad\quad employees\\
\quad\quad \textbf{else } r)\\
\quad departments
\end{array}
$$

5 Testing Functional Equalities

The following algorithm tests whether any two canonical terms compute the same function. It is based on the uniqueness property that says that there is a unique way for expressing a function as a fold. Given any two canonical programs $\lambda\overline{x}.t_1(\overline{x})$ and $\lambda\overline{y}.t_2(\overline{y})$ in our term language (i.e. t_1 and t_2 are canonical terms), $\mathcal{E}(t_1(\overline{x}), t_2(\overline{x}))$ returns true if and only if $t_1(\overline{x}) = t_2(\overline{x})$ for any input $\overline{x}$. $\mathcal{E}(x,y)$ is computed by the following rules:

1	$\mathcal{E}(x,x)$	$\longrightarrow$	**true**	
2	$\mathcal{E}(x,y)$	$\longrightarrow$	**false**	if $x \neq y$
3	$\mathcal{E}(C_k(\overline{u}), C_k(\overline{w}))$	$\longrightarrow$	$\bigwedge_i \mathcal{E}(u_i, w_i)$	
4	$\mathcal{E}(C_k(\overline{u}), C_m(\overline{w}))$	$\longrightarrow$	**false**	if $k \neq m$
5	$\mathcal{E}(\text{fold}^T(\overline{f})\,x, \text{fold}^T(\overline{g})\,x)$	$\longrightarrow$	$\bigwedge_i \mathcal{E}(f_i(\overline{y}), g_i(\overline{y}))$	
6	$\mathcal{E}(g(x), \text{fold}^T(\overline{f})\,x)$	$\longrightarrow$	$\bigwedge_i \mathcal{E}(g(C_i(\overline{y})), f_i(E_i^T(g)\,\overline{y}))$	
7	$\mathcal{E}(\text{fold}^T(\overline{f})\,x, g(x))$	$\longrightarrow$	$\bigwedge_i \mathcal{E}(f_i(E_i^T(g)\,\overline{y}), g(C_i(\overline{y})))$	

where $\overline{y}$ in Rules 5, 6, and 7 are new variable names. All rules are evaluated in sequence and only the first applicable rule is used. The last two rules come from the uniqueness property. They require that both terms $g(C_i(\overline{y}))$ and $f_i(E_i^T(g)\,\overline{y})$ are improved before they are passed to the equality checker $\mathcal{E}$. This implies that this algorithm can decide functional equalities only for safe terms. Rule 5 is used as the bottom case for Rules 6 and 7.

For example, $\mathcal{E}(\text{plus}(y,x), \text{plus}(x,y)) = \mathcal{E}(\text{fold}^{nat}(\lambda().x, \lambda(r).\text{Succ}(r))y, \text{fold}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))x)$. We apply the uniqueness property for $g(x) = \text{fold}^{nat}(\lambda().x, \lambda(r).\text{Succ}(r))\,y$:

$$
\begin{array}{lll}
x = \text{Zero}: & \mathcal{E}(g(\text{Zero}), y) & = \mathcal{E}(\text{fold}^{nat}(\lambda().\text{Zero}, \lambda(r).\text{Succ}(r))y, y)\\
 & & = \textbf{true}\\
x = \text{Succ}(i): & \mathcal{E}(g(\text{Succ}(i)), \text{Succ}(g(i))) & \\
 & = \mathcal{E}(\text{fold}^{nat}(\lambda().\text{Succ}(i), \lambda(r).\text{Succ}(r))y, \text{Succ}(g(i))) &
\end{array}
$$

we apply the uniqueness property again with $f(y) = \text{Succ}(g(i))$:

$$
\begin{array}{lll}
y = \text{Zero}: & \mathcal{E}(f(\text{Zero}), \text{Succ}(i)) & = \mathcal{E}(\text{Succ}(i), \text{Succ}(i))\\
 & & = \mathcal{E}(i,i) = \textbf{true}\\
y = \text{Succ}(j): & \mathcal{E}(f(\text{Succ}(j)), \text{Succ}(f(j))) & = \mathcal{E}(\text{Succ}(f(j)), \text{Succ}(f(j)))\\
 & & = \mathcal{E}(f(j), f(j)) = \textbf{true}
\end{array}
$$

Therefore, $\mathcal{E}(\text{plus}(y,x),\text{plus}(x,y))$ is true.

Set equality is different than structural equality for sums-of-products types. Testing whether two expressions e_1 and e_2 of type *set* are equal is equivalent to testing whether set_equal(e_1,e_2) is equal to the term True.

Each canonical form specifies an equivalence class of safe programs that are reduced to this form by the normalization algorithm. In fact, there may be infinite number of such programs in a class. For example, $x+y$, $x+(y+0)$, $x+(y+0+0)$ etc. compute the same function and they are all reduced to the same canonical form $x+y$. The equivalence class of all canonical programs that compute the same function f is called the *canonical extension* of f. From the uniqueness property and by using case analysis we can prove that for any function f that can be expressed as a canonical program, the canonical extension is finite. The optimization of function f is a search over the finite space of the canonical extension of f. This process will be explained in detail in Section 8.

6 Program Synthesis

The program synthesis algorithm described in this section is a pattern matching algorithm. It is based on the equality tester for canonical programs, as it is described in Section 5. We use here the term program synthesis in its wider context of constructing programs, not in its usual narrow form of producing programs given the input/output specification of the programs.

A *canonical pattern* is a canonical program in which some of its variables are annotated as pattern variables. A pattern variable with name x is represented as $\#x$, in order to be distinguished from regular term variables. For example,

$$\text{Cons}(\#x,\text{fold}(\lambda().\text{Nil},\lambda(a,r).\#y)\,z)$$

is a canonical pattern with pattern variables $\#x$ and $\#y$.

A substitution list ρ is either **fail** or a partial binding from pattern variables to canonical programs. We denote $\rho[x/g]$ the extension of ρ with the binding from the pattern variable $\#x$ to the canonical program g, $\rho[x]$ is the value of $\#x$ in ρ, and $\rho(f)$ is a canonical pattern that results after replacing all pattern variables in f that occur in ρ by their bindings. A canonical pattern f is equal to a canonical program g under the substitution ρ if $\rho(f)$ does not contain any pattern variables and $\mathcal{E}(\rho(f),g) = \textbf{true}$.

A canonical pattern f matches a canonical program g under a substitution ρ, denoted as $[f \equiv g]\,\rho$, if there is a substitution ρ' such that $\mathcal{E}(\rho'(\rho(f)),g) = \textbf{true}$. The algorithm in Figure 1 implements $[f \equiv g]\,\rho$. Rules 2 through 8 are very similar to the rules for $\mathcal{E}(f,g)$, as they are described in Section 5. The only difference is that instead of collecting all test results using the *and* operator, we accumulate all substitution lists, starting with ρ, as it is dictated by the $\mathcal{A}_i$ accumulation function:

$$\mathcal{A}_i(\ f_i\ )\,\rho\ =\ f_n(f_{n-1}(\ldots f_2(f_1(\rho))))$$

For example, $\mathcal{A}_i(\ [u_i \equiv w_i]\ )\,\rho$ in Rule 4 is:

$$[u_n \equiv w_n]([u_{n-1} \equiv w_{n-1}]\ldots([u_1 \equiv w_2]([u_1 \equiv w_1]\,\rho)))$$

Rules 9 through 12 involve pattern variables. Rule 9 is straightforward. Rule 10 is similar to Rule 6. It serves as a bottom case for Rules 11 and 12. If Rule 10

$$
\begin{array}{lcll}
[e \equiv u]\,\mathbf{fail} & \longrightarrow & \mathbf{fail} & \mathbf{1} \\
[x \equiv x]\,\rho & \longrightarrow & \rho & \mathbf{2} \\
[x \equiv y]\,\rho & \longrightarrow & \mathbf{fail} \quad \text{if } x \neq y & \mathbf{3} \\
[C_k(\overline{u}) \equiv C_k(\overline{w})]\,\rho & \longrightarrow & \mathcal{A}_i(\,[u_i \equiv w_i]\,)\,\rho & \mathbf{4} \\
[C_k(\overline{u}) \equiv C_m(\overline{w})]\,\rho & \longrightarrow & \mathbf{fail} \quad \text{if } k \neq m & \mathbf{5} \\
[\mathrm{fold}^T(\overline{f})\,x \equiv \mathrm{fold}^T(\overline{g})\,x]\,\rho & \longrightarrow & \mathcal{A}_i(\,[f_i(\overline{z_i}) \equiv g_i(\overline{z_i})]\,)\,\rho & \mathbf{6} \\
[g(x) \equiv \mathrm{fold}^T(\overline{f})\,x] & \longrightarrow & \mathcal{A}_i(\,[g(C_i(\overline{z_i})) \equiv f_i(E_i^T(g)\,\overline{z_i})]\,)\,\rho & \mathbf{7} \\
[\mathrm{fold}^T(\overline{f})\,x \equiv g(x)] & \longrightarrow & \mathcal{A}_i(\,[f_i(E_i^T(g)\,\overline{z_i}) \equiv g(C_i(\overline{z_i}))]\,)\,\rho & \mathbf{8} \\
[\#f \equiv g]\,\rho & \longrightarrow & \rho[f/g] & \mathbf{9} \\
[\mathrm{fold}^T(\overline{g})\,\#f \equiv \mathrm{fold}^T(\overline{\phi})\,x]\,\rho & \longrightarrow & [\mathrm{fold}^T(\overline{g})\,x \equiv \mathrm{fold}^T(\overline{\phi})\,x]\,(\rho[f/x]) & \mathbf{10} \\
[\mathrm{fold}^S(\overline{g})\,\#f \equiv \mathrm{fold}^T(\overline{\phi})\,x]\,\rho & \longrightarrow & \rho'[f/\mathrm{fold}^T(\lambda\overline{z_1}.\rho'[f_1], \ldots, \lambda\overline{z_n}.\rho'[f_n])\,x] & \mathbf{11} \\
 & & \text{where } \rho' = \mathcal{A}_i(\,[\mathrm{fold}^S(\overline{g})\,\#f_i \equiv & \\
 & & \qquad \phi_i(E_i^S(\mathrm{fold}^S(\overline{g}))\,\overline{z_i})]\,)\,\rho & \\
[\mathrm{fold}^S(\overline{g})\,\#f \equiv \phi]\,\rho & \longrightarrow & \rho'[f/C_k(\rho'[f_1], \ldots, \rho'[f_n])] & \mathbf{12} \\
 & & \text{where } C_k \text{ is any constructor of } S \text{ and} & \\
 & & \rho' = [g_k(E_k^S(\mathrm{fold}^S(\overline{g}))\,(\#f_1, \ldots, \#f_n)) \equiv \phi]\,\rho &
\end{array}
$$

Figure 1: The Program Synthesis Algorithm

is invoked then neither of Rules 11 or 12 can be invoked. Rule 11 comes from the promotion theorem with $g = \mathrm{fold}(\overline{g})$. Rule 12 tests every constructor C_k of type S to see if it gives an acceptable solution for $\#f$. Rules 11 and 12 may overlap. This may result to more than one solution to a pattern matching. All these rules can be used as rewrite rules in a rule-base system. The search engine of this rule-base system should test each alternative until it finds one that does not fail. This search can be guided by cost functions and heuristics for synthesising the best solutions. Rule 12 may lead into an infinite recursion as in the following example:

$$[\mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{Cons}(a,r))\,\#f \equiv x]$$

This has the solution $f = \mathrm{Cons}(\#f_1, \mathrm{Cons}(\#f_2, \ldots))$ (infinite times). Fortunately, the search engine can detect such infinite loops by keeping each triple $(\mathrm{fold}^S(\overline{g}),\, \phi,\, C_k)$ in Rule 12 in a stack and abort Rule 12 if it is invoked twice for the same triple.

For example, suppose that we want to synthesize f in

$$[\mathrm{length}(\#f) \equiv \mathrm{plus}(\mathrm{length}(x), \mathrm{length}(y))]$$

where $\mathrm{length}(x) = \mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\,x$. After normalizing the right part of the previous equation we get:

$$[\mathrm{length}(\#f) \equiv \mathrm{fold}^{nat}(\lambda().\mathrm{length}(y), \lambda(r).\mathrm{Succ}(r))\,x]$$

We will try first $f = \mathrm{fold}^{list}(\lambda().\#f_1, \lambda(a,r).\#f_2)\,x$ (from Rule 11). Then we get the following two equations:

$$[\mathrm{length}(\#f_1) \equiv \mathrm{length}(y)] \quad \text{and} \quad [\mathrm{length}(\#f_2) \equiv \mathrm{Succ}(\mathrm{length}(r))]$$

The first equation gives $f_1 = y$ (from Rule 11). To solve the second equation we try first $f_2 = \text{Nil}$ (from Rule 12) which fails. Then we try $f_2 = \text{Cons}(f_3, f_4)$ that gives:

$$[\text{Succ}(\text{length}(\#f_4)) \equiv \text{Succ}(\text{length}(r))]$$

After using Rules 4 and 10 we get $f_4 = r$. Therefore,

$$f = \text{fold}^{list}(\lambda().y, \lambda(a,r).\text{Cons}(\#f_3, r))\, x$$

Variable f_3 was not bound to any term since it was eliminated during normalization. This means that it is universally quantified and can be bound to any canonical term. If $f_3 = a$ then f is list append.

7 Mapping Logical to Physical Operations

One of the goals of this work is to achieve a high degree of data independence, that is, a separation between the abstract model and the implementation details. That way the model designer does not have to worry about program efficiency and space utilization, but only to be concerned with expressing correct specification. In addition, this separation may offer more opportunities for optimization. One effective mechanism for separating the model from implementation is *type transformation* [9]. In this framework each abstract data type in the abstract program is mapped into a storage structure by an *abstraction function.*

Definition 9 (Abstraction Function) *Let T be an abstract type and S the type of a concrete storage structure. An abstraction function is a unary function $\mathcal{R}_S^T$ of type $S \rightarrow T$ expressed in canonical form.*

For example, if $T = set$ and $S = list$ then one possible $\mathcal{R}_{list}^{set}$ is:

$$\mathcal{R}_{list}^{set} = \text{fold}^{list}(\lambda().\text{Emptyset}, \lambda(a,r).\text{Insert}(a,r))$$

The abstraction function is in general a many-to-one function, that is, an abstract type may be mapped into many different storage structures. For example, a set can be mapped into a list of any order. Having the optimizer select the order is an additional source for optimization. Functions $\mathcal{R}_S^T$ can be provided in a form of an extensible system library. A mapping from an abstract type T to a concrete type S may have more than one abstraction functions $\mathcal{R}_S^T$. The database implementor is responsible for selecting the right storage structures to map the abstract types as well as the abstraction functions that specify this mapping. This implementation phase is performed after the specification phase. The optimizer will use this information to derive a translation that satisfies both the abstract specification and the mapping transformations. This is achieved by the following theory (derived from the type transformation diagram that commutes):

Definition 10 (Function Implementation) *Let f be an abstract function of type $T_1 \times \cdots \times T_n \rightarrow T_0$ and for $0 \leq i \leq n$, r_i is an abstraction function $\mathcal{R}_{S_i}^{T_i}$. Then a function F of type $S_1 \times \cdots \times S_n \rightarrow S_0$ is an implementation of f iff:*

$$r_0 \circ F = f \circ (r_1 \times \cdots \times r_n)$$

For example, let sets be mapped into lists by the function $\mathcal{R}^{set}_{list}$ described above. Then UNION (of type $list \times list \rightarrow list$) is the implementation of union (of type $set \times set \rightarrow set$) iff:

$$\mathcal{R}^{set}_{list} \circ \text{UNION} \;=\; \text{union} \circ (\mathcal{R}^{set}_{list} \times \mathcal{R}^{set}_{list})$$

that is, for all x and y:

$$\mathcal{R}^{set}_{list}(\text{UNION}(x, y)) \;=\; \text{union}(\mathcal{R}^{set}_{list}(x), \mathcal{R}^{set}_{list}(y))$$

For example, one solution for UNION that satisfies this equation is list append. Note that append maintains all duplicated elements while union does not. This is consistent with our type transformation since the result of append will be mapped by $\mathcal{R}^{set}_{list}$ into a set. If we want the output of UNION to reflect the output of union then we better use an isomorphic mapping for the output type of union.

The homomorphic equation $r_0 \circ F \;=\; f \circ (r_1 \times \cdots \times r_n)$ can be solved in two ways. We can select a detailed implementation of the output type T_0 of f by specifying the inverse abstraction function r_0^{-1} from T_0 to S_0. Then

$$F \;=\; r_0^{-1} \circ f \circ (r_1 \times \cdots \times r_n)$$

This implies that the mapping from T_0 to S_0 is specified as an isomorphism. This solution could miss some valid optimizations, especially in the case of sets. A more general solution is to use the program synthesis algorithm as it is described in Section 6. In that case, F is derived by the following pattern matching:

$$[r_0(\#F) \equiv f(r_1(x_1), \ldots, r_n(x_n))]\,\rho$$

Note that in both cases F is expressed in terms of concrete primitives only when it is normalized, such as UNION is expressed in list primitives exclusively. This can be proved by induction[5]: if F is a variable then the statement is true; if it is a construction $C(t_1, \ldots, t_n)$ then none of the parameters of C returns an abstract object since C is a constructor of a concrete type which by definition does not refer to any abstract type; if it is a fold $\text{fold}^T(\overline{f})\,x$ that returns a concrete type then all f_i return concrete types.

The type transformation model can also be used for translating virtual view updates. Each such view can be captured as a many-to-one function v from the abstract database type T to the view type S. Suppose that we perform an update u over this view, that is, u is a function from S to S. The view update problem is to find a database update U that transforms the database state in such a way that the view of the new database state is identical to the original view after the application of u. That is $v \circ U \;=\; u \circ v$. As before, the solution for U can be derived from $[v(\#U) \equiv u(v(db))]$.

Another application of type transformation is database restructuring after schema and/or implementation evolution. Schema evolution from a database type T to a database type T' can be captured as a function c of type $T' \rightarrow T$. If the abstraction function that implements the abstract database type T as the storage structure S was r, then this function should change to r' (of type $S' \rightarrow T'$) to reflect the schema changes. The function R that restructures

[5] We assume that there are no types in common between the abstract and concrete types.

the database state satisfies the equation $c \circ r' \circ R = r$, which can also be solved by using the synthesis algorithm. Note that the resulting solution for R may have uninstantiated variables. This indicates that there are values in the new database state that need to be filled, such as, in the case of a tuple extended with new components. Schema evolution can be performed in small parts by specifying how some components of the database type change. The compiler should be able to accumulate these pieces of information and use them to synthesize the functions c and r' from r. A convenient tool for performing this task is compile-time reflection [7].

8 Decomposition into Concrete Primitives

Suppose that a query is normalized into the following canonical form:

$$\text{fold}^{list}(\lambda().\text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\, y, \lambda(a,r).\text{Succ}(r))\, x$$

and there are the following abstractions in the physical layer:

$$\begin{array}{lcl} \text{append}(x,y) & = & \text{fold}^{list}(\lambda().y, \lambda(a,r).\text{Cons}(a,r))\, x \\ \text{length}(x) & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\, x \\ \text{plus}(x,y) & = & \text{fold}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\, x \end{array}$$

then this canonical form can be expressed either as length(append(x, y)) or as plus(length(x), length(y)). If the cost of plus is 1, the cost of length is the size of its input, and the cost of append is the size of its first input, then clearly the latter choice is better than the first. The following analysis automates this process of mapping canonical forms into compositions of concrete primitives.

Definition 11 (Concrete Layer) *The concrete layer consists of a set $\mathcal{CL}$ of quadruples* (name, behavior, implementation, cost) *where name is the name of a concrete abstraction, behavior is the canonical program that specifies the behavior of this abstraction, implementation is the actual implementation of this abstraction in a possibly non-applicative reference-based language, and cost is the cost function that manipulates a user-defined cost data structure.*

The cost data structure is typically a tuple that contains the necessary components for computing cost values. Cost values satisfy a user-supplied partial order. Each component of a quadruple p from $\mathcal{CL}$, such as name, can be accessed as p[name].

The following algorithm, called the *fold optimization algorithm*, transforms a canonical form f into a composition of concrete primitives. It is invoked as $\mathcal{G}\, f\, \rho$. It returns the set of all possible compositions of concrete names from $\mathcal{CL}$ whose resulting canonical form is equal to f. This algorithm uses set comprehensions to derive all permutations.

$$\begin{array}{lll} \mathbf{1} & \mathcal{G}\, f\, \textbf{fail} & = \emptyset \\ \mathbf{2} & \mathcal{G}\, v\, \rho & = \{ v \} \qquad \textit{(if } v \textit{ is a variable)} \\ \mathbf{3} & \mathcal{G}\, C(f_1, \ldots, f_n)\, \rho & = \{ C(\phi_1, \ldots, \phi_n) \,/\, \forall i : \phi_i \leftarrow \mathcal{G}\, f_i\, \rho \} \\ \mathbf{4} & \mathcal{G}\, f\, \rho & = \{ a[\text{name}](\phi_1, \ldots, \phi_n) \,/\, a \leftarrow \mathcal{CL}, \\ & & \qquad \rho' = [a[\text{behavior}](\#f_1, \ldots, \#f_n) \equiv f]\, \rho, \\ & & \qquad \forall i : \phi_i \leftarrow \mathcal{G}\, (\rho'[f_i])\, \rho' \} \end{array}$$

Rules 1 and 2 are straightforward. Rule 3 says that if f is a construction $C(f_1, \ldots, f_n)$ then we return $C(\phi_1, \ldots, \phi_n)$, where each ϕ_i is derived by calling this algorithm recursively for each component f_i. The last rule uses the synthesis algorithm to transform f into a call to the concrete operation a. The parameters ϕ_i of this call can be derived by using the program synthesis algorithm and can be transformed by using the fold optimization algorithm recursively. Rule 4 is evaluated when f is either a construction or a fold. The efficiency of this algorithm can be improved in many ways. First, this algorithm can be implemented as a search-based algorithm guided by a[cost]. The cost of a variable is derived from statistical information while the cost of a single construction is 'unit', which is a user-supplied constant. (A complete framework for specifying database statistics and for assigning costs to all variables in a canonical program is proposed in [7].) Estimating costs for canonical programs is easier than for non-canonical programs, since the former do not materialize intermediate results and, therefore, they do not require selectivity estimations. Second, we do not need to check all concrete abstractions a in $\mathcal{CL}$, since we can use type information to discriminate them.

9 Specifying Semantic Information as Type Restrictions

Each type is associated with a set of values, the instances of the type, that share common properties, such as common operations. We can restrict the set of instances of a type T further by using the **where** type constructor:

$$T' \quad = \quad T \ \textbf{where}(x)\ p(x)$$

This defines a new type T' whose instances are all the instances x of type T that satisfy the predicate $p(x)$, where p is a canonical term of type $T \rightarrow$ boolean.

The following are examples of restrictions:

$$\begin{array}{lcl} \text{range1} & = & \text{nat } \textbf{where}(x)\ x \geq 10 \ \textbf{and}\ x \leq 20 \\ \text{nset}(\alpha) & = & \text{set}(\alpha)\ \textbf{where}(x)\ x \neq \text{Emptyset} \\ \text{slist}(\alpha) & = & \text{Msl}(\text{info} : \text{list}(\alpha), \text{size} : \text{nat})\ \textbf{where}(x)\ x.\text{size} = \text{length}(x.\text{info}) \end{array}$$

The last example keeps the length of a list as redundant information attached to the list. Redundant information is very important to optimization as it offers alternative methods of execution to choose from that may have different costs. For example, it is cheaper to access x.size from the slist x to derive the length of x than it is to compute length(x.info).

In addition to type parameters, type definitions can be parameterized by values that are used in the where clauses of these type definitions. This is called a *parameterized restriction*:

$$T'[x_1 : t_1, \ldots, x_n : t_n] \quad = \quad T \ \textbf{where}(x)\ p(x, x_1, \ldots, x_n)$$

This defines a new type T' whose instances are all instances x of the type T that satisfy the predicate $p(x, x_1, \ldots, x_n)$. That is, this restriction is parameterized

and it is instantiated whenever the parameters x_i are instantiated to values during compile time. For example:

$$\begin{aligned}
&\text{bounded}(\alpha)[\mathit{low}:\alpha, \mathit{high}:\alpha, f:(\alpha,\alpha)\to\text{boolean}] \\
&\qquad\qquad = \alpha\ \textbf{where}(x)\ f(\mathit{low}, x)\ \textbf{and}\ f(x, \mathit{high}) \\
&\text{range}[\mathit{low}:\text{nat}, \mathit{high}:\text{nat}] \ = \ \text{bounded(nat)}[\mathit{low}, \mathit{high}, \leq] \\
&\text{range1} \ = \ \text{range}[10, 20] \\
&\text{ordered_list}(\alpha)[f:(\alpha,\alpha)\to\text{boolean}] \ = \ \text{list}(\alpha)\ \textbf{where}(x)\ \text{ordered}(x, f) \\
&\text{keyed_set}(\alpha,\beta)[f:(\alpha)\to\beta] \\
&\qquad\qquad = \text{set}(\alpha)\ \textbf{where}(x)\ \text{card}(x) = \text{card}(\text{image}(f)\,x) \\
&\text{persons} \ = \ \text{keyed_set}(\text{person}, \text{nat})[\lambda p.(p.\text{ssn})]
\end{aligned}$$

where ordered(x, f) is true if list x is ordered by the function f, card is set cardinality, and image$(f) = \text{fold}^{set}(\lambda().\text{Emptyset}, \lambda(a, r).\text{Insert}(f(a), r))$.

Instances of a restricted type should obey all restrictions propagated to this type. The following algorithm derives the predicate $\mathcal{I}\{T\}$ which all instances of a type T need to satisfy, by accumulating all restrictions from all types referred by T:

Definition 12 (Accumulated Restriction) *Each type T is associated with an accumulated restriction $\mathcal{I}\{T\}$ defined by the following inductive equations:*

$$\begin{aligned}
\mathcal{I}\{T(t_1,\ldots,t_n)\}\,x \quad &= \quad \text{fold}^T(\ldots, \lambda\overline{x}.\textstyle\bigwedge\overline{x}, \ldots) \\
&\qquad\qquad (\text{map}^T(\mathcal{I}\{t_1\},\ldots,\mathcal{I}\{t_n\})\,x) \\
\mathcal{I}\{T\ \textbf{where}(x)\ f(x)\}\,x \quad &= \quad \mathcal{I}\{T\}x\ \textbf{and}\ f(x) \\
\mathcal{I}\{T\}\,x \quad &= \quad \textbf{true} \qquad \textit{(otherwise)}
\end{aligned}$$

that is, each value x of type T must satisfy $\mathcal{I}\{T\}x$.

Here map^T maps any parametric type $T(\alpha_1,\ldots,\alpha_n)$ into $T(\beta_1,\ldots,\beta_n)$, thus it requires n functions, one for each type variable. In the first equation, fold^T accumulates all results from map^T using the $\wedge$ (**and**) operator.

For example, for the type T defined as:

$$T \ = \ \text{list}(\text{range}[0, 5])\ \textbf{where}(x)\ \text{length}(x) \leq 10$$

$\mathcal{I}\{T\}\,x$ is:

$$\text{fold}^{list}(\lambda().\textbf{true}, \lambda(a, r).0 \leq a\ \textbf{and}\ a \leq 5\ \textbf{and}\ r)\,x\ \textbf{and}\ \text{length}(x) \leq 10$$

Type checking in the presence of type restrictions requires theorem proving capabilities. For example, $f(g(x))$, where $f : T \to \alpha$ and $g : \beta \to T'$, is type correct if type T is compatible (unifiable) with T' when all type restrictions are removed, and if $\mathcal{I}\{T'\}\,y \Rightarrow \mathcal{I}\{T\}\,y$ is true. This property is very useful for discriminating incompatible compositions of abstractions during decomposition into concrete primitives (Section 8). For example, the type signatures for sort and merge are:

fun sort (x : list(α), f : $\alpha \times \alpha \to$ boolean) $\to$ ordered_list(α)[f]

fun merge (x : ordered_list(α)[f], y : ordered_list(α)[f], f : $\alpha \times \alpha \to$ boolean)
$\to$ ordered_list(α)[f]

For example, merge(sort(x, f), sort(y, f), f) is type correct while merge(x, y, f) is not, where x and y are unordered lists.

Type restrictions must be used during type transformation to derive consistent implementations. Furthermore, they may yield additional alternatives for function implementations, such as in the case of redundancy constraints. That way, path selection can be performed during algebraic optimization, since alternative paths are specified by redundancy constraints attached to storage structures.

For example, suppose that a set(person) is mapped into

$$\begin{array}{rcl} T & = & \text{pair}(\text{ordered\_list}(\text{person})[\lambda(x,y).x.\text{ssn} \leq y.\text{ssn}], \\ & & \quad \text{ordered\_list}(\text{person})[\lambda(x,y).x.\text{name} \leq y.\text{name}]) \\ & & \textbf{where}(x)\, \mathcal{R}^{set}_{list}(x.\text{fst}) = \mathcal{R}^{set}_{list}(x.\text{snd}) \end{array}$$

using the abstraction function $r = \lambda x.\mathcal{R}^{set}_{list}(x.\text{fst})$. That is, a set of persons is mapped into two lists (that represent indices): one ordered by the ssn of persons and the other by the name of persons. Accessing a person can be achieved by accessing either of these two access paths. The optimizer should be able to select the access with the lowest cost.

Each abstract type T_i is mapped into a concrete type S_i by an abstraction function $r_i = \mathcal{R}^{T_i}_{S_i}$. All type restrictions involving either T_i or S_i must be used when deriving a function implementation F of f:

$$\bigwedge_i (\mathcal{I}\{S_i\}x_i \,\textbf{and}\, \mathcal{I}\{T_i\}r_i(x_i)) \;\Rightarrow\; r_0(F(x_1,\ldots,x_n)) = f(r_1(x_1),\ldots,r_n(x_n))$$

where $(x \Rightarrow y) = \text{fold}^{boolean}(\lambda().\text{True}, \lambda().y)\, x$.

Type restrictions can be used to control program synthesis (Section 6). In particular, redundancy constraints, such as the one in the slist example, can expand the search space of the program synthesizer. Suppose that we want to synthesize f such that $[g(\#f) \equiv \phi(x_1,\ldots,x_n)]\,\rho \neq \textbf{fail}$. Each variable x_i of type T_i must obey the $\mathcal{I}\{T_i\}$ predicate:

$$\bigwedge_i \mathcal{I}\{T_i\}x_i \;\Rightarrow\; ([g(\#f) \equiv \phi(x_1,\ldots,x_n)]\,\rho \neq \textbf{fail})$$

The solution for f can be derived from the following:

$$[g(\#f) \equiv \phi(\textbf{if}\, \mathcal{I}\{T_1\}x_1\, \textbf{then}\, x_1\, \textbf{else}\, \Box, \ldots, \textbf{if}\, \mathcal{I}\{T_n\}x_n\, \textbf{then}\, x_n\, \textbf{else}\, \Box)]\,\rho$$

This indicates that if x_i does not satisfy $\mathcal{I}\{T_i\}x_i$ then the ith parameter of ϕ is set to $\Box$ (a distinct canonical term). The program synthesiser must be extended to include the rule $[f \equiv \Box]\,\rho = \textbf{fail}$, that is, pattern matching fails if one of the integrity constraints is not satisfied. When $\Box$ is composed with a canonical term f during program improvement, the result is $\Box$, i.e. $\Box \circ f = f \circ \Box = \Box$. Therefore, the right part of the equation for $g(\#f)$ can be normalized to a canonical form that can be used by the program synthesizer to derive a solution for f.

10 Conclusion

We have presented a new query algebra based on a small number of primitives that facilitates query optimization. This algebra is expressive enough to capture a wide range of database operations over an extensible number of bulk data structures. Its simplicity and uniformity make it a perfect target for a rich functional database programming language.

Algebraic optimization can only be effective if it is combined with information about physical implementation, such as available access paths and algorithms. Our framework integrates implementation specifications with algebraic optimization in such a way that program normalization always improves efficiency independently of the database state. Cost estimates and database statistics can be used by the optimizer in a later phase to search the space of equivalent normalized physical programs, which are derived from the initial query after type transformation and normalization. Our optimizer is an extensible, general-purpose, optimizer that does not make a-priori assumptions about the underlying physical structures and algorithms. It would be interesting to compare the performance of such a powerful system with a special-purpose optimizer, such as the one for Ingres SQL. Special-purpose optimizers contain many shortcuts, especially when it comes to storage mapping, since they assume a fixed model for storage structures and algorithms. Therefore, such optimizers may be more efficient but less extensible.

In the future, we intend to design a user-friendly query language which can be translated directly into the fold algebra. Comprehensions are not adequate for this purpose since they are less expressive than folds. We would like also to define suitable fold operations for other data structures that cannot be captured as sums-of-products types. In particular, arrays are very interesting and challenging since they support random accesses.

Acknowledgements: The author is grateful to Sophie Cluet, Scott Daniels, Dave Maier, Eliot Moss, Sushant Patnaik, Tim Sheard, David Stemple, Jianwen Su, and Bennet Vance for helpful comments on the paper. This work is currently supported by the Advanced Research Project Agency, ARPA order number 018, monitored by the US Army Research Laboratory under contract DAAB07-91-C-Q518.

References

[1] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *International Conference on Database Theory, Paris, France*, pp 72–88. Springer-Verlag, December 1990. LNCS 470.

[2] S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of the Polynomial Time. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, B.C.*, pp 283–293, May 1992.

[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, August 1991.

[4] W. Chin. Safe Fusion of Functional Expressions. *Proceedings of the ACM Symposium on Lisp and Functional Programming, San Francisco, California*, pp 11–20, June 1992.

[5] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pp 493–501, January 1993.

[6] J. Darlington and R. Burstall. A System which Automatically Improves Programs. *Acta Informatica*, 6(1):41–60, 1976.

[7] L. Fegaras. *A Transformational Approach to Database System Implementation*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, February 1993. Also appeared as CMPSCI Technical Report 92-68.

[8] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York*, pp 148–162. Springer-Verlag, June 1992. LNCS 607.

[9] 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, August 1991.

[10] J. C. Freytag and N. Goodman. On the Translation of Relational Queries into Iterative Programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.

[11] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pp 335–347. Springer-Verlag, June 1989. LNCS 375.

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pp 124–144, August 1991. LNCS 523.

[13] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pp 23–34, May 1979.

[14] 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.

[15] S. Shenoy and Z. Ozsoyoglu. Design and Implementation of a Semantic Query Optimizer. *ACM Transactions on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[16] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, pp 344–358, March 1988. LNCS 300.