On the Translation of Relational Queries into Iterative Programs

JOHANN CHRISTOPH FREYTAG
European Computer-Industry Research Centre
and
NATHAN GOODMAN
Kendall Square Research

This paper investigates the problem of translating set-oriented query specifications into iterative programs. The translation uses techniques of functional programming and program transformation.

We present two algorithms that generate iterative programs from algebra-based query specifications. The first algorithm translates query specifications into recursive programs. Those are simplified by sets of transformation rules before the algorithm generates the final iterative form. The second algorithm uses a two-level translation that generates iterative programs faster than the first algorithm. On the first level a small set of transformation rules performs structural simplification before the functional combination on the second level yields the final iterative form.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; H.2.4 [**Database Management**]: Systems—*query processing*; I.2.2 [**Artifical Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Languages, Performance

Additional Keywords and Phrases: Algebraic specification, database, iterative programs

1. INTRODUCTION

With today's computer technology, data are frequently stored and accessed by a *database management system* (DBMS). Such a system provides a uniform interface for the definition of internal structures to store, modify, and access data.

In a *relational database system* [7, 25], queries are expressed in a data independent manner: The user formulates a request without knowing anything about the internal representation of the data accessed and describes in the query only those conditions that the final response to the request must satisfy. Therefore the database system is responsible for developing an evaluation strategy that computes the desired result efficiently. The component that decides on the best evaluation strategy is called the *query optimizer.*

This research was supported by the office of Naval Research under ONR-N00014-83-K-0770. The research was done while J. C. Freytag was a student at Harvard University, Cambridge, MA 02138.
Authors' addresses: J. C. Freytag, European Computer-Industry Research Centre, D-8000 Munich, West Germany; N. Goodman, Kendall Square Research, Cambridge, MA 02139.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0362-5915/89/0300-0001 $01.50

Using the information about the internal data representation, the query optimizer generates a *query evaluation plan* (QEP) for a submitted user query. In this paper we assume that a query evaluation plan describes the query computation in terms of set-oriented operations [20]. Before it can be executed, a query evaluation plan must be translated into a program that operates on tuples as its basic objects. We call the problem of generating this program from a QEP the *translation problem for relational queries.* The next section describes the problem in more detail and outlines solutions that are currently used. Their disadvantages lead to the different approach taken in this paper.

Our approach to the translation problem is based on techniques and results of *functional programming* [2] and *program transformation* [6]. Functional programming formalism helps us to formulate simple algebraic rules. Those rules manipulate programs in their functional form to derive efficient iterative programs from initial query plans. Some important aspects of functional programming and program transformation are discussed in Section 1.3.

This paper is divided into five sections. Section 2 describes the source and the target level of the translation. Sections 3 and 4 develop two different, but closely related, algorithms for the translation of algebra-based QEPs into iterative programs. The first algorithm gives an elegant solution to the translation problem. However its inefficiency motivated the design of the second one which advances more quickly towards the final iterative form of the program. Finally, Section 5 summarizes the contributions of this paper and briefly outlines open problems that are closely related to the translation problem of relational queries.

1.1 Query Optimization and Evaluation

In a relational database system we distinguish two major components: the *logical database processor* (LDBP) and the *physical database processor* (PDBP). The LDBP translates a user-submitted query into an internal representation and optimizes it for efficient execution. The *query optimizer*, a subcomponent of the LDBP, decides on the best evaluation strategy for a user-submitted query. Information about the internal representation of the data accessed by the query and the evaluation strategies available for each operator in the query determines the selection of a *query evaluation plan* (QEP) that is described by *set-oriented operators.*

After the optimizer has generated its final QEP, the PDBP must evaluate the QEP against the database to compute the requested result. The PDBP has to satisfy two conflicting requirements: On the one hand, its interface should be set-oriented, thus allowing the immediate evaluation of QEPs; on the other hand, it must be amenable to an efficient execution. To ensure the latter many database systems implement a PDBP whose basic operations manipulate single *tuples*, or *records.* To execute QEPs, they provide a set of procedures that *independently* implement the various set-oriented operators permitted in QEPs. These procedures access data using tuple-oriented operations by the PDBP. For example, Buneman and Frankel suggest a set-oriented language whose operations are implemented as parameterized procedures in terms of operations on the underlying PDBP [5]. Their system, which takes advantage of *lazy evaluation* [14], gives an *interpretive* solution to the translation problem of relational queries.

An alternative solution is to *transform*, or *compile*, the set-oriented QEPs into programs that are directly executable on the tuple-oriented interface of the PDBP. For example, Lorie et al. compile the set-oriented access specification language ASL into machine executable programs which embed calls to the Relational Storage System (RSS), the PDBP in SYSTEM R [18]. However, due to the complexity of the compilation step, R* replaces the compilation of ASL statements by their interpretation [10].

This paper describes a solution to the translation problem that avoids the complexity encountered by Lorie. We develop two algorithms that translate QEPs into programs directly executable on the PDBP. Instead of choosing the machine language as the target language, we translate QEPs into languages such as C or Pascal, which are extended by operations of the PDBP. Existing compilers for these languages can then translate the generated programs into machine-executable ones.

Our two-level translation approach has several advantages over the solution of SYSTEM R. First and most important, the generation of iterative programs from modular specifications *minimizes the amount of overhead during execution.* Procedure calls and repeated evaluation of the same functions are eliminated. The compiler may further optimize the iterative programs during the compilation into machine language. Second, the generation of programs in higher-level languages guarantees a certain degree of *machine independence*: The creation of these programs does not require any knowledge of a particular machine environment. Switching compilers achieves portability onto different machines. Third, the separation into two independent steps *simplifies* the translation problem. High-level programming constructs easily express QEPs without considering other aspects during the generation of programs in the target language. Additionally the suggested translation avoids duplicating tasks that compilers already perform. We assume that they carry out standard program optimization and optimization for a particular machine environment.

1.2 Functional Programming and Program Transformation

In this section we review some important aspects of *functional programming* and *program transformation* that influenced our solutions of the query translation problem. Generally speaking, program transformation promises to provide a comprehensive solution to the problem of producing programs that try to meet several incompatible goals simultaneously: On the one hand, programs should be correct and clearly structured, thus allowing easy modification. On the other hand, one expects them to be executed efficiently. Using languages like C or Pascal for the implementation of programs, one is immediately forced to consider aspects of efficiency that are often unrelated to their correctness, natural structure, and clarity.

For this reason, the transformational approach tries to separate these two concerns by dividing the programming task into two steps: The first step concentrates on producing programs that are written as clearly and understandably as possible without considering efficiency issues. The second step successively transforms programs into more efficient ones—possibly for a particular machine environment—using methods that preserve the meaning of the original

program. The kind of improvements that are necessary during the second step must go beyond the ones achievable by the optimization phase of conventional compilers.

In many ways, these intentions guided the design of query languages for relational database systems such as QUEL [22] or SQL [1]. Both languages permit the user to express his or her request in a clear and understandable form that describes properties of the requested result without considering an efficient execution. As we discussed in the previous section, the database system has to find an efficient execution strategy. The query optimizer produces a QEP that determines all steps of an evaluation. However, the QEP still needs further refinement to guarantee an efficient execution on conventional computer systems.

Many early transformation methods considered the manipulation of recursively defined programs or programs defined by recursive equations [11]. In particular, Burstall and Darlington developed a set of transformation schemes for removal of recursion together with a systematic approach for the manipulation of recursive programs [6, 11].

John Backus' Turing award lecture introduced a new style of programming using the FP language [2]. As Williams explains, "The FP style of functional programming differs from the lambda (i.e., recursive) style in that the emphasis in the former is on the application of functionals (i.e., combining forms) to functions, whereas the emphasis in the latter is on the application of functions to objects to produce new objects" [26]. Much of the current research in functional programming and program transformation has centered around this new approach [3, 4, 9].

The different transformation systems in this paper reflect the two directions in program transformation. Section 3 develops a translation algorithm that exclusively relies on the definition of recursive programs. Long transformations of recursive programs encouraged us to develop a new transformation system based on ideas of Backus' FP language. The transformation, described in Section 4, leads to the final program form much faster.

2. THE SOURCE LEVEL AND THE TARGET LEVEL OF THE TRANSFORMATION

This section defines in detail the operations used for the definition of QEPs and for the programs executable on the PDBP. They form the source and the target level of the transformation. The former is based on the standard file organization, such as used by [1] or [20]. We use *tables* as an abstract data structure to separate the target level from details of the data model used for a particular PDBP. Tables storing user-provided data are called *data tables*.

The set of operators consists of the following two subsets: Operators in the first subset, called *source operators*, provide access to tables in the database returning sets of tuples. Operators in the second group, called *set operators*,[1] transform sets of tuples into sets of tuples. Sets of tuples differ from tables in that we leave unspecified if they reside in main memory or on secondary memory. This aspect will become irrelevant for the performed transformations. We define

[1] We use the term "set" to describe a collection of tuples without eliminating duplicates.

both groups of operators as follows:

(SCAN *table*) is a source operator that accesses a table and returns the *set of all its tuples.* Our definition only determines the source and leaves the destination unspecified.[2]

(FILTER *pred*? *set_of_tuples*) is a set operator that returns the subset of those tuples satisfying the predicate *pred*?. The precise format of predicate *pred*? is unimportant for the later transformation.

(PROJECT *project_list set_of_tuples*) is a set operator that projects a set of tuples onto the specified attributes of the projection list.

(LJOIN *join*? *set_1 set_2*) is a set operator that defines a *nested-loop join.* Each tuple in the first set is matched with every tuple in the second set to test the join predicate *join*?. If the test evaluates to *true* for two tuples their concatenation is added to the output set.

This set of operators forms the basis for the definition of QEPs. A QEP is said to be *well formed* if it is either a source operator, or a set operator whose input set is defined by a well-formed QEP. We assume that all other parameters to the operators are correctly provided according to the above definitions.

Example 1. To demonstrate different transformations we shall use the following two relations containing data of employees at a university.

EMP (Emp#, Name, Salary, Dept. Status)
PAPERS (Emp#, Title, Year)

Each tuple in relation EMP describes an employee by his or her employee number, name, salary, department, and status. Relation PAPERS stores the employees who have written papers, recording the title and the year of the publication.

We use the database to evaluate the following query:

Find the names of all professors who published papers after 1980.

The query optimizer may have decided that it performs the evaluation best for the submitted query by the following QEP:

```
(PROJECT (Name)
  (LJOIN (Emp# = Emp#)
    (FILTER (Status = "Prof") EMP)
    (FILTER (Year > 1980) (SCAN PAPERS))))
```

The current set does not include operators to access indexes, to create new tables, or to store tuples in tables. The programs generated by the transformation algorithm will reflect this restriction. However, the current set of operators can easily be extended to cope with additional operators without changing the transformation algorithms presented in the next sections.

While the data model for QEPs performs operations on sets of tuples and tables, the PDBP operates on *individual tuples.* Abstracting from particular implementations, we introduce the notion of *streams* [14]. Streams are sequences of tuples that were introduced by [14] in the context of lazy evaluation. In any

[2] [21] uses the same operator to move an entire table from a source to a destination location.

stream only the first element is computed. The rest of the stream is evaluated only if needed. In our context the word connotes images of tuples *streaming through* the different operators. We define a generic set of operators that manipulate streams while hiding details of a specific PDBP. This approach treats the access to tables and sets of tuples in a uniform way. We define the following five operators that access, create, and test streams:

(*first stream*): returns the first element in the *stream.*

(*rest stream*): returns a new stream by removing the first element in *stream.*

**empty*: is the empty stream.

(*out ele stream*): creates a new stream whose first element is *ele* followed by all elements in *stream.*

(*empty? stream*): evaluates to true if *stream* is empty and to false if *stream* is an expression of the form (*out ele str*).

While QEP operations completely specify the source language, a target language must be defined to express the programs resulting from the translation. For this purpose we propose a small functional language in a Lisp-like notation that is well suited for formal manipulation. We will describe the target language informally. The language is based on *expressions.* An expression is either a *variable,* a *function expression,* or a *conditional expression.* A function expression has the form $(f_1\ t_1\ \ldots\ t_n)$ where f_1 is a function symbol and t_i, $i = 1, \ldots, n$, are expressions, called actual parameters. A conditional expression has the form $(if\ t_1\ t_2\ t_3)$, where t_1, t_2, and t_3 are expressions. If t_1 evaluates to *true,* then the value of t_2 is the value of the expression, otherwise it is the value of t_3. When t_3 is superfluous we use the form $(if\ t_1\ t_2)$.

In addition to expressions, we introduce the *definitions of functions* by the equation $(f_1\ x_1\ \ldots\ x_n) = t_1$ where f_1 is a function symbol, the x_i are formal parameters, and t_1 is an expression built from function symbols, possibly including f_1 and variables x_i. $(f_1\ x_1\ \ldots\ x_n)$ is called the *function header* of f_1 with t_1 as its *function body* $B(f_1)$. If f_1 is called in a function expression, then its value is determined by substituting $B(f_1)$ for the expression and replacing the formal parameters by the actual ones. In the sequel, we shall use $t[x_1/t_1]$ to denote the substitution of all occurrences of variable x_1 by expression t_1 in expression t.

Functions and expressions are sufficient for most steps in the translation process. However, to express iterative programs we extend the language by an *assignment statement* and a *loop statement.* Generally, these statements do not occur in function definitions; again they are used only in the final output of the transformation. An assignment statement is of the form $(set\ x\ t_1)$ where x is a variable and t_1 is an expression. The loop statement has the form $(do\ (x\ t_1\ t_2\ t_3)\ t_4)$ with x as a variable, t_1, t_2, t_3 as expressions, and t_4 is an assignment statement or an expression. The semantics of the loop statement can be described best by the Pascal-like program:

$x := t_1$; WHILE t_3 DO BEGIN t_4; $x := t_2$; END;

3. QUERY TRANSFORMATION BASED ON RECURSIVE PROGRAMS

This section develops a transformation algorithm based on the *manipulation of recursive programs.* The interested reader can find a brief description of the

algorithm in [13]. However, this section presents the algorithm in a more comprehensive form by providing rigorous definitions and specifications of the transformation that later guarantees the correctness of the algorithm developed.

The transformation algorithm is based on four major transformation steps, all of which have been suggested by Burstall and Darlington [6]. The first step, called *unfolding*, replaces an operator by its corresponding recursive function. To perform this first step we define recursive programs, or functions, for each QEP operator. Each program correctly implements the corresponding operator.

During the second step, a set of transformation rules *simplifies* the initial program without changing its intended meaning. The simplification step leads to a *minimal form* of the program. The third step, folding, further reduces the program generated so far. This step is the inverse of unfolding. Using the recursive function form produced by the third step, the final step replaces recursion by iteration. Based on these four transformation steps, the complete transformation algorithm is presented in Section 3.5.

3.1 DB Functions and DB Expressions

For the unfolding step we define programs, called *DB functions*, that implement each QEP operator, respectively. A combination of DB functions is called *DB expression*. While QEP operators are denoted by *capital letters*, DB functions will be denoted by *lowercase letters*. They are defined by expressions of the target language introduced in Section 2. Each DB function calls only itself, functions of the PDBP, other DB functions, and possibly additional functions, which are denoted by pred?, join?, and prj. Functions pred?, join?, and prj *implement* the predicates used as the first parameter to the FILTER operator and LJOIN, and the projection list for PROJECT operator, respectively. Their exact form is unimportant for the transformation process. We denote the arguments to QEP operators by *capital letters*, i.e., *PRED*?, *JOIN*?, and *PRJ*, and the functions implementing them by *lowercase letters*, i.e., pred?, join?, and prj. For the implementation of LJOIN we also introduce the function conc to concatenate two tuples. We define the following DB functions for QEP operators:

```
(scan R)          = (if (empty? R)  *empty*
                        (out (first R) (scan (rest R))))
(filter stream)   = (if (empty? stream)  *empty*
                        (if (pred? (first stream))
                            (out (first stream) (filter (rest stream)))
                            (filter (rest stream))))
(project stream)  = (if (empty? stream)  *empty*
                        (out (prj (first stream))
                             (project (rest stream))))
(ljoin str1 str2) = (if (empty? str1) *empty*
                        (union (jn (first str1) str2)
                               (ljoin (rest str1) str2)))
(jn ele stream)   = (if (empty? stream)  *empty*
                        (if join? ele (first stream))
                            (out (conc ele (first stream))
                                 (jn ele (rest stream)))
                            (jn element (rest stream))))
(union str1 str2) = (if (empty? str1) str2
                        (out (first str1) (union (rest str1) str2)))
```

The reader may have noticed the difference in the number of arguments for QEP operators FILTER, PROJECT, and LJOIN, and their implementing functions filter, project, and ljoin respectively. The first parameter of the QEP operators does not occur as a parameter in the corresponding function. *Not including these parameters in the definition header will simplify the later transformation significantly.* Instead this parameter is "compiled into" the DB functions. We denote the specialized functions by filter1, project1, and ljoin1, respectively. Furthermore, for uniformity reasons we treat table R in the definition of (scan R) as a stream. Again, this will allow us to simplify the later transformations.

Example 2. Consider the operator (FILTER $PRED_1$? *set*). The corresponding DB function is defined as

```
(filter1 stream) = (if (empty? stream)   *empty*
                       (if (pred1? (first stream))
                           (out (first stream) (filter1 (rest stream)))
                           (filter1 (rest stream))))
```

Function filter1 calls pred1? in its definition. filter1 may be viewed as a "specialized" version of the function filter since the function pred1?, implementing predicate $PRED_1$?, replaces the general function pred?. The same conventions apply to functions project and ljoin.

The definitions of functions scan, project, filter, jn, and union all have a common form, which is called *linear recursive form.* The definition for function f has a *linear recursive form* if f is the only recursive function symbol in its body. Function ljoin is an exception since it accesses two streams. Its definition reflects the functional decomposition. Function jn joins the first tuple of the first stream with all tuples of the second before the union function concatenates the result of jn with the set of tuples resulting from the recursive call to ljoin. However, since the recursive functions union and jn do not call function join in their body, the definition of ljoin resembles a "quasilinear" form. Therefore, the form of ljoin's definition is called *extended linear form* [8]. Both forms of linear recursion will later allow us to translate recursive programs into iterative ones as discussed in Section 3.4.

3.2 Simplification Rules and Their Properties

To simplify expressions in our target language, this section introduces a set of transformation rules that forms the *simplification step* of the transformation algorithm presented in Section 3.3.4. The rules perform transformations that do not change the computational behavior of the program. They generate a *minimal form* for each expression with a minimal number of conditional expressions and only one output stream. The rule set, which we call $\mathcal{R}_{simp}$,[3] consists of the following rules:

Rule I: *Function Exchange Rule*

$$((f\,(if\ t_1\ t_2\ t_3)) \rightarrow (if\ t_1\ (f\ t_2)\ (f\ t_3)))$$

[3] *simp* for simplification.

Rules IIa and IIb: *Deletion Rules*

$((\mathit{if}\ t_1 (\ldots (\mathit{if}\ t_1\ t_2\ t_3) \ldots)\ t_4) \rightarrow (\mathit{if}\ t_1 (\ldots t_2 \ldots)\ t_4))$
$((\mathit{if}\ t_1\ t_2\ (\ldots (\mathit{if}\ t_1\ t_3\ t_4) \ldots)) \rightarrow (\mathit{if}\ t_1\ t_2\ (\ldots t_4 \ldots)))$

Rule III: *Distribution Rule*

$((\mathit{if}\ (\mathit{if}\ t_1\ t_2\ t_3)\ t_4\ t_5) \rightarrow (\mathit{if}\ t_1\ (\mathit{if}\ t_2\ t_4\ t_5)\ (\mathit{if}\ t_3\ t_4\ t_5)))$

Rules IVa and IVb: *True/False Rules*

$((\mathit{if}\ \mathit{true}\ t_1\ t_2) \rightarrow t_1)$
$((\mathit{if}\ \mathit{false}\ t_1\ t_2) \rightarrow t_2)$

Rules Va, Vb, Vc, and Vd: *Stream Rules*

$((\mathit{first}\ (\mathit{out}\ t_1\ t_2)) \rightarrow t_1)$
$((\mathit{rest}\ (\mathit{out}\ t_1\ t_2)) \rightarrow t_2)$
$((\mathit{empty?}\ *\mathit{empty}*) \rightarrow \mathit{true})$
$((\mathit{empty?}\ (\mathit{out}\ t_1\ t_2)) \rightarrow \mathit{false})$

The *function exchange rule* performs the "unwinding" of a function with *one parameter*: the function is distributed over the conditional expression. Note that we only define the function exchange rule for *one-parameter functions*. We shall later show that this rule is sufficient for the transformation of expressions generated from QEP operators. The *deletion rules* apply to conditional expressions; they replace superfluous conditional expressions by either the consequent or the alternate of the eliminated expression. The *distribution rule* simplifies two nested conditional expressions. The first pair of *stream rules* implement partial evaluation for conditional expressions. The remaining rules reflect some semantic knowledge about the stream operators. They help to reduce the number of intermediate results by performing *partial evaluation* on streams. For correctness, we assume the evaluation of any function occurs without side-effects. Furthermore, the arguments to all functions are strict [14]; that is, any computation to obtain values for function parameters is guaranteed to terminate [12].

The set of transformation rules $\mathcal{R}_{simp}$ is certainly not the most general one. Other researchers suggest larger sets of rules that can transform more general programs [3, 4, 15]. However, our set of rules is sufficient to transform all functions and programs derived from QEPs.

The rule set $\mathcal{R}_{simp}$ enjoys two desirable properties. First, any transformation terminates after a finite number of rule applications. That is, there exists an expression that cannot be transformed any further by a given set of rules. Huet calls this property *noetherian* [17]. Second, the final form of an expression derived by the rule set is independent of the order in which rules are applied. Huet calls a term transformation system *confluent* if it has this desirable property [17]. He also provides tests for both properties that are based on the set of transformation rules. For further details we refer the interested reader to [17] for the general theory of term rewriting systems. For space restrictions, we cannot include the proofs in this paper that show the confluent and noetherian property for the above rules set and other sets presented in this paper. We refer the reader to [12] for more details of these proofs.

3.3 The Transformation of DB Functions

Based on the function definitions for QEP operators, this section derives necessary results for the correctness of the transformation algorithm IDEAL, which is presented in Section 3.3.4. The algorithm consists of three major transformation steps, called *unfolding*, *simplification*, and *folding*, all of which are introduced in Section 3.3.1. They transform an expression into an *ideal form*, which is defined in Section 3.3.2. The ideal form allows us to replace recursion by iteration during the fourth transformation step as we shall discuss in Section 3.4. Section 3.3.3 shows that the transformation of any combination of DB functions yields the ideal form.

3.3.1 *The Transformation Steps.* Once a QEP has been translated into a DB expression, the transformation proceeds in three major steps. The first step, called *unfolding*, replaces each DB function by the body of its definition using the actual parameters. Formally, let f_1 and f_2 be two functions with their definitions $(f_1\ x_1 \ldots x_{n_1}) = B(f_1)$ and $(f_2\ y_1 \ldots y_{n_2}) = B(f_2)$, respectively. Then, the expression $s(f_1, f_2, i) = (f_1\ x_1 \ldots x_{i-1}\ (f_2\ y_1 \ldots y_{n_2})\ x_{i+1} \ldots x_{n_1})$ with $i \leq n_1$ denotes the DB expression with a function call to f_2 as the ith parameter in f_1's function call. We denote the expansion by the function bodies of f_1 and f_2 by $S(f_1, f_2, i) = B(f_1)[x_i/B(f_2)]$ which represents the result of the unfolding step.

The unfolding of expressions allows us to investigate the "interaction" between different functions. When combined and unfolded, the same function calls with the same parameter may appear more than once in the resulting expression. The second step, called the *simplification step*, simplifies these expressions by the set of transformation rules $\mathscr{R}_{simp}$ introduced in Section 3.2. The rules in $\mathscr{R}_{simp}$ rearrange the order of functions without changing the computational behavior of expressions, and eliminate superfluous function calls. Given an expression t, the rules are applied *exhaustively* to t deriving its *minimal form* as discussed in Section 3.2. We denote the minimal form using rule set $\mathscr{R}_{simp}$ by SIMPLIFY(t).

Example 3. Consider the functions f_1 = project1 and f_2 = filter1, then

$s(f_1, f_2, 1)$ = (project1 (filter1 str))

Due to space restrictions we do not show the unfolded expression $S(f_1, f_2, 1)$ resulting from $s(f_1, f_2, 1)$. Applying the transformation rules to $S(f_1, f_2, 1)$, the minimal form is

```
if (empty? str)  *empty*
  (if (pred1? (first str))
      (out (prj1 (first str)) (project1 (filter1 (rest str))))
      (if (empty? (filter1 (rest str))))  *empty*
         (out (prj1 (first (filter1 (rest str))))
              (project1 (rest (filter1 (rest str))))))))
```

The expression resulting from the simplification step already contains fewer operations than the unfolded expression. However, the example shows the need for a *third* transformation step. We would like to derive an expression that refers only to str, (first str), and (rest str). That is, the stream functions empty?, first, and rest should operate only on stream variables and not on functions producing streams.

To eliminate the function expressions (first (filter1 (rest str))) and (rest (filter1 (rest str))), we recognize that the subexpression

```
(if (empty? (filter1 (rest str)))  *empty*
   (out (prj (first (filter1 (rest str))))
        (project1 (rest (filter1 (rest str))))))
```

resembles the body of the function definition for the function project1 using the actual parameter (filter1 (rest str)). The third step, called the *folding* step, replaces the subexpression by the function expression (project1 (filter1 (rest str))) yielding the final expression

```
(if (empty? str)  *empty*
   (if (pred1? (first str))
      (out (prj1 (first str)) (project1 (filter1 (rest str))))
      (project1 (filter1 (rest str)))))
```

The following definition describes the folding step formally.

Definition 3.1. Let $\mathcal{G}$ be a set of distinct DB functions. For any two functions $f_i, f_j \in \mathcal{G}$, $i \neq j$, we assume that f_i's function body is not derivable from f_j's function body using $\mathcal{R}_{simp}$. Assume that there exists a subexpression t'' in t such that $t'' = B(f)[x_1/t_1 \ldots x_n/t_n]$ for some expressions $t_1 \ldots t_n$ and for some function $f \in \mathcal{G}$ where $x_1 \ldots x_n$ are the variables in f. If we replace t'' by the function expression $(f\ t_1 \ldots t_n)$ we derive a new expression t'. We say that $\mathcal{G}$ folds t by f deriving t'.

If such an f exists the restriction on functions f_i in $\mathcal{G}$ ensures that the result of the folding step is unique. We use the notation FOLD(t, $\mathcal{G}$) to denote the expression t' that results from exhaustively folding t by functions f_i in $\mathcal{G}$.

3.3.2 *Forms of Expressions.* To precisely describe the *form* of the expressions derived by the three transformation steps, we introduce *1-recursive functions* and *ideal forms*. Intuitively, a function f of arity n is called 1-recursive if the recursion occurs only on *one* parameter. The parameter is called the *recursive parameter*; all other parameters that do not change during the evaluation are called *constant* parameters. Those may be needed by functions called in the body of f. For example, the definition of ljoin uses two parameters: str1 is the recursive parameter; str2, which is the constant parameter for ljoin, becomes the recursive parameter of function jn.

Definition 3.2. Let $(f\ x_1 \ldots x_n) = B(f)$ be a linear recursive function f. If there exists a function f1 with $n + 1$ parameters such that f does not appear in f1's function body and f1 does not appear in f's function body, and $B(f) = B(\text{f1})[x_{n+1}/(f\ x_1 \ldots x_{i-1}\ (\text{f2}\ x_i)\ x_{i+1} \ldots x_n)]$ for some f2 then f is called 1-recursive in parameter i, $i \leq n$.[4]

Example 4. The definition of function ljoin is 1-recursive in its first parameter. Function f2 is equal to rest, and function f1 is defined by

```
(f1 t1 t2 t3) = (if (empty? t1)  *empty*
                    (union (jn (first t1) t2) t3))
```

[4] Cohen [8] calls these functions "non-ordered, totally ordered."

PROPOSITION 3.1. *All DB functions are 1-recursive.*

For the composition of functions we introduce the notion of *ideal form*. The ideal form precisely describes the form of expression we would like to derive from DB expressions by all three transformation steps since this form later guarantees the replacement of recursion by iteration. We shall argue in Section 3.3.3 that the three steps achieve this transformation goal. The ideal form also enables us to define a *new 1-recursive function* that computes the same result as the combination of two 1-recursive functions as the following example shows.

Example 5. Consider the DB expression (project1 (filter1 str)) of Example 3 that yielded the expression

```
(if (empty? str)  *empty*
   (if (pred1? (first str))
      (out (prj1 (first str)) (project1 (filter1 (rest str))))
      (project1 (filter1 (rest str)))))
```

We introduce the new function

```
(project_filter1 str) = (project1 (filter1 str))
```

and derive its definition by replacing all occurrences of (project1 (filter1 str)) by (project_filter1 str) in the above expression, thus yielding the 1-recursive function

```
(project_filter str) =
    (if (empty? str)  *empty*
       (if (pred1? (first str))
          (out (prj1 (first str)) (project_filter1 (rest str)))
          (project_filter1 (rest str))))
```

The following definition formally introduces the ideal form for the combination of functions.

Definition 3.3. Let $(f_1\ x_1 \ldots x_{n_1}) = B(f_1)$, $(f_2\ y_1 \ldots y_{n_2}) = B(f_2)$ be 1-recursive in parameters i, j, respectively, such that f_1 does not appear in f_2's function body and f_2 does not appear in f_1's function body, and $t' = s(f_1, f_2, i)$. If there exists a function f' with $n_1 + n_2$ parameters such that

$$t'' = B(\mathrm{f}')[x_{n_1+n_2}/(f_1\ x_1 \ldots x_{i-1}\ (f_2\ y_1 \ldots (f''\ y_j) \ldots y_{n_2})\ x_{i+1} \ldots x_{n_1})]$$

for some f″ that computes the same result as t', then we say $s(f_1, f_2, i)$ has an ideal form t''.

If there exists an ideal form for $s(f_1, f_2, i)$, then we may define a new 1-recursive function $(g\ x_1 \ldots x_{i-1}\ x_{i+1} \ldots x_{n_1}\ y_1 \ldots y_{n_2})$ with $n_1 + n_2 - 1$ parameters such that

$$B(g) = B(\mathrm{f}')[x_{n_1+n_2}/(g\ x_1 \ldots x_{i-1}\ x_{i+1} \ldots x_{n_1}\ y_1 \ldots (\mathrm{f}''\ y_j) \ldots y_{n_2})].$$

We say that $s(f_1, f_2, i)$ *yields* a 1-recursive function.

Example 6. Consider the first expression of Example 5 that fulfills the requirement of Definition 3.3 with f″ = rest and

```
(f' t1 t2) = (if (empty? str)  *empty*
                 (if (pred1? (first t1))
                    (out (prj (first t1)) t2) t2))
```

Thus, the expression represents the ideal form for s(project1, filter1, 1). The function project_filter1 as defined in Example 5 is the newly derivable function g.

Using the transformation as an *operation* on functions we shall show in the next two sections that the set of 1-recursive functions is *closed* under the operation transformation as defined by unfolding, simplification, and folding.

We distinguish three different kinds of DB functions according to the structure of their bodies. In the sequel, variables e1 ... en denote *elements* (that is, one tuple), while str1 ... strn denote streams of tuples. To describe DB functions scan or project we introduce the general function form

```
(f1 e1 ... em str1) =
    (if (empty? str1)  *empty*
        (out (g1 e1 ... em (first str1)) (f1 e1 ... em (rest str1))))
```

Similarly, for DB functions such as filter, we generalize their form to

```
(f2 e1 ... em str2) =
    (if (empty? str2) *empty*
        (if (pred? (first str2))
            (out (g2 e1 ... em (first str2))
            (f2 e1 ... em (rest str2)))
            (f2 e1 ... em (rest str2))))
```

Since functions of both forms perform the recursion on *one* stream, we call these functions *1-stream functions*. For functions which have two stream parameters, such as nljoin, we introduce the general form of *2-stream functions*

```
(f3 e1 ... em str3 str4) =
    (if (empty? str3) *empty*
        (union (g3 e1 ... em (first str3) str4)
               (f3 e1 ... em (rest str3) str4)))
```

which we can generalize to a form for *n-stream functions*.[5]

```
(f e1 ... em str1 ... strn) =
    (if (empty? str1) *empty*
    (if (pred1? (first str))
        (if (pred2? (first str))
        ...
            (union (g e1 ... em (first str1) str2 ... strn)
                   (f e1 ... em (rest str1) str2 ... strn))
            (f e1 ... em (rest str))
        ...
            (f e1 ... em (rest str)))
        (f e1 ... em (rest str))))
```

3.3.3 *The Transformation of 1-stream and 2-stream Functions.* Based on the definitions of these general function forms we have to prove that any combinations of functions yields an ideal form, that is, a form that represents a linear recursive function. Due to space limitations we do not present the proofs; instead, we describe the aspects that are common to all proofs. The interested reader can find the complete proofs in [12].

[5] Without loss of generality we assume that recursion occurs on the first stream.

As a first step, we need to show that any combination of all 1-stream functions yields an ideal form using unfolding, simplification, and folding. Using the definitions of the previous section, each inductive proof[6] first unfolds the expression before simplifying the expression by a sequence of applications of rules from set $\mathscr{R}_{simp}$, thus yielding an ideal form that, as one expects, resembles the form of a new 1-stream function. Appendix A shows one of the proofs in more detail.

Unfortunately, the unfolding, simplification, and folding steps are not sufficient to create the ideal forms for 2-stream functions. Since the definition of ljoin uses the union operator, DB expressions which include ljoin require more manipulation after these three transformation steps have been performed. Consider the DB expression

```
(filter1 (join1 str1 str2))
```

Substitution, simplification, and folding yield the expression

```
(if (empty? str1)  *empty*
   (filter1 (union (jn1 (first str1) str2)
                   (join1 (rest str1) str2))))
```

which does not represent an ideal form. Function filter1 has to be moved over the union function to achieve the ideal form

```
(if (empty? str1)  *empty*
   (union (filter1 (jn1 (first str1) str2))
          (filter1 (join1 (rest str1) str2))))
```

We introduce a second set of transformation rules $\mathscr{R}_U$ that defines a fourth transformation step, called the *union step*. To correctly define the rules in $\mathscr{R}_U$, we need to introduce *restricted rules*. A restricted rule is denoted by

$$(a \xrightarrow{C} b)$$

where C is a condition containing variables used in a. The rule can only be applied if the condition evaluates to *true* when assigning values to variables in a. Consider the restricted rule

$$((f_2 \text{ (union } t_1\ t_2)\ t_3) \xrightarrow{C} \text{(union } (f_2\ t_1\ t_3)\ (f_2\ t_2\ t_3)))$$

where $C = (f_2 \neq \text{union})$. If we apply the rule to the expression

```
(join1 (union str1 str2) str3)
```

f_2 = join1 evaluating C to *true*. Therefore the rule can be applied yielding the expression

```
(union (join1 str1 str3) (join1 str2 str3))
```

$\mathscr{R}_U$ contains the following rules:

$$\begin{aligned}\mathscr{R}_U = \{&((\text{union } (\text{union } t_1\ t_2)\ t_3) \rightarrow (\text{union } t_1\ (\text{union } t_2\ t_3))),\\ &((f_1\ (\text{union } t_1\ t_2)) \rightarrow (\text{union } (f_1\ t_1)\ (f_1\ t_2))),\\ &((f_2\ (\text{union } t_1\ t_2)\ t_3) \xrightarrow{C} (\text{union } (f_2\ t_1\ t_3)\ (f_2\ t_2\ t_3))),\\ &((f_2\ t_1\ (\text{union } t_2\ t_3)) \xrightarrow{C} (\text{union } (f_2\ t_1\ t_2)\ (f_2\ t_1\ t_3)))\}\end{aligned}$$

[6] The induction is on the number of functions in the DB expression.

where f_i denotes a function of arity one and two. The last two rules are needed for 2-stream functions such as ljoin. For their application we have to exclude function union and impose the restriction $C = (f_2 \neq \text{union})$. Notice that the restriction is important for two reasons: first, if $f_2 = \text{union}$ the transformation result is semantically incorrect since we would duplicate values of the two streams.[7] Second, without the restrictions $\mathcal{R}_U$ does not have the noetherian property.

We are now ready to prove that any combination of 1-stream functions and 2-stream functions yield an ideal form using unfolding, simplification, folding, and the union step. Again, we restrict ourselves to outlining the proof steps. The interested reader can find all proofs in [12]. First, we show that any combination of one 2-stream function with any 1-stream function yields an ideal form. Again, the proof is by induction: we unfold the functions, construct a sequence of rules that simplify the expression before folding the expression and applying the union step. The result resembles the form of a 2-stream function. Similarly we show that the combination of n 2-stream functions result in an ideal form that resembles an $n - 1$-stream function. It is interesting to notice that any n-stream function (f str1 ... strn) that is derived from any DB expression, contains a subexpression of the form:

(union (f1 (first str1) str2 ... strn) (f (rest str1) str2 ... strn))

That is, an n-stream function is evaluated by calling an $n - 1$-stream function and calling itself recursively. We shall later use this observation to prove the correctness of the transformation algorithm.

It follows from the above results that *any combination of 1-stream and 2-stream functions yields an ideal form that represents the function body of some n-stream function.*

3.3.4 *The Simplification Algorithm.* Based on the results of the previous section we define algorithm IDEAL, which returns the ideal form for any DB expression. If $t = (f_1\ (f_2\ (f_3\ str)))$, then the ideal form for $(f_2\ (f_3\ str))$ is computed first, before the combination with f_1 is encountered.

Algorithm IDEAL

Input: DB expression t **Output**: ideal form

Assume all table names are unique.

begin

if $t = (f_1\ t_1)$ and t is a table name: /* case 1 */.

return $(B(f_1)[str1/t_1])$;

if $t = (f_1\ t_1)$ and t is a nested DB expression: /* case 2 */

l_1 := IDEAL (t_1);

l_2 := SIMPLIFY $(B(f_1)[str1/l_1])$;

l_3 := FOLD $(l_2, \{f_1, t_1\})$;

l_4 := UNION (l_3);

return (l_4);

[7] We do not assume that union is a set operator, which eliminates duplicate tuples.

```
    if t = (f1 t1 t2):  /* case 3 */
      l1 := IDEAL (t1);
      l2 := IDEAL (t2);
      l3 := SIMPLIFY (B(f1) [str1/l1, str2/l2]);
      l4 := FOLD (l3, {f1, t1, t2});
      l5 := UNION (l4);
      return (l5);
end;
```

In case (1) of the algorithm, we have reached the bottom of the recursion and do not need any simplification or folding. The algorithm simply returns the body of the function with the actual parameter substituted. Cases (2) and (3) take care of the combination with a 1-stream and a 2-stream function, respectively. In both cases variables t_1 and t_2 represent DB expressions, which are transformed by calling IDEAL recursively. For folding, t_1 and t_2 are considered as functions whose bodies are the ideal forms l_1 and l_2, respectively, as discussed in Section 3.3.3.

3.4 Recursion Removal

The final transformation step that we propose for the improvement of programs replaces recursion by iteration. While the recursive form of programs supports clear programming style and simplifies program manipulation, it may not be the most efficient form for program execution [6, 8]. Cohen gives a comprehensive overview on various recursive forms and their transformation into iteration [8].

This section discusses the replacement of extended linear recursion by iteration using the transformation schemes suggested by [6]. Let f be a linear recursive function of the form

```
(f t1) = (if (pred? t1) f1 (f2 (f3 t1) (f (f4 t1))))
```

and f2 is commutative, then f can be transformed into the iterative program

```
((set result f1)
 (do (t t1 (f4 t) (pred? t))
     (set result (f2 (f3 t) result))))
```

The iterative form introduces the variable result, which *accumulates* the intermediate results on each call to f2. The variable is initialized to f1, to the "bottom" value of the recursion.

LEMMA 3.1. *The ideal form of any combination of DB functions yields an iterative program using the above recursion scheme.*

PROOF. The proof immediately follows from the definitions of a 1-recursive function and the ideal form together with the results of the previous sections. Since the order of tuples is irrelevant, functions out and union satisfy the above condition for f2. □

For the transformation algorithm in the next section we introduce the mapping RI, which replaces recursion by iteration as follows: Let f be an n-stream function

that is 1-recursive on its first-stream parameter with

```
(f e1 ... em str1 .. strn) =
    (if (empty? str1)   *empty*
        (g3 e1 ... em (first str1) str2 ... strn
            (f e1 ... em (rest str1) str2 ... strn)))
```

for some g3. Then we define RI(f, B(f)) to be

```
(do (t stri (rest t) (empty? t))
    (g3 e1 ... em (first t) str2 ... strn result))
```

3.5 The Transformation Algorithm

Using the different transformation steps developed in the previous sections, we are now ready to present the complete transformation algorithm $\mathcal{TR}_{rec}$:

Let $\hat{e}$ denote the empty expression. Let $\mathcal{R}(t)$ be any subexpression $t' = (f_1\ t_1\ \ldots\ t_n)$ in t such that f_1 is a recursive function and t' is not embedded in another function expression calling a recursive function. If t' does not exist in t, let $\mathcal{R}(t) = \hat{e}$. For any QEP *qep*, function $\mathcal{DB}(qep)$ returns the corresponding DB expression. Function $\mathcal{A}(t, r)$ applies rule r exhaustively to expression t.

Algorithm $\mathcal{TR}_{rec}$

Input: Query evaluation plan *qep* **Output:** Iterative program

begin

$t = \mathcal{DB}(qep)$; $t_1 = t$;
while $t_1 \neq \hat{e}$ *do*
  t_2 := IDEAL(t_1); /* *compute the ideal form* */
  t_3 := RI(t_1, t_2); /* *generate iterative expression* */
  $t := t[t_1/t_3]$; /* *replace recursion by iteration* */
  $t_1 := \mathcal{R}(t)$ /* *more recursion ?* */
end_do;
/* *generate assignment statements* */
t := ((*set result* * *empty* *) *prg*) [*prg*/t];
$t := \mathcal{A}(t, ((out\ a\ b) \rightarrow (set\ result\ (out\ a\ b))))$
end;

THEOREM 3.1. *Given a QEP qep with n tables, algorithm* $\mathcal{TR}_{rec}$ *produces an iterative expression with n loops.*

PROOF. We perform induction on the number of tables. The translation of a QEP with n tables yields a DB expression with n streams. If $n = 1$, then steps IDEAL and RI produce an expression with one loop containing no more recursive functions. If $n = i$, then step IDEAL produces an ideal form yielding an i-stream function. The ideal form contains the union of an $i - 1$-stream function with itself as the only recursive expression. Step RI replaces the reference to the i-stream function by the variable *result*, thus leaving an $i - 1$-stream expression; by the hypothesis this yields an iterative expression with $i - 1$ loops. □

An analysis of $\mathcal{TR}_{rec}$'s computation reveals that much time is spent with unfolding and folding of functions. Consider the nesting of joins of the form

```
(join1 (join2 ... (joinn str strn) ... str2) str1)
```

then the alternation between both transformation steps results in little progress for the overall transformation. The major reason is that the innermost stream str forms the outermost loop in the final iterative program. Repeated alternations of unfolding and folding "pushes" str through the levels of joins [12].

Another inefficiency arises when computing the ideal form for a subexpression within an expression. Many times this computation is superfluous since the ideal form is always folded back to the original DB expression to achieve the ideal form for the whole expression.

Finally, a significant number of the transformation steps in the simplification phase are concerned with the elimination of stream functions. Whenever two DB functions are combined, the rules eliminate the intermediate stream that is passed from one DB function to the next. This observation suggests that we should find a better representation of DB functions that reduce the number of transformation steps during simplification.

4. QUERY TRANSFORMATION BASED ON MAP EXPRESSIONS

The analysis of algorithm $\mathcal{TR}_{rec}$ motivated us to develop a different formalism that would overcome the described problems without losing the advantages offered by techniques of functional programming and program transformation. The following observation explains one of the reasons for the difficulties encountered: the recursive definition of DB functions intertwines aspects of *control structure information* and *functional* "behavior." Each function definition specifies in a tightly coupled manner *what* has to be computed and *how* to perform the computation. This observation motivated a notation which—at least initially—would clearly separate these two aspects for their manipulation. This section first develops such notation together with two transformation systems to manipulate information about the control structure and functional behavior of programs *independently*. We first translate DB expressions into *map expressions* that emphasize the control structure aspects. Once map expressions have been transformed into a normal form we translate them into λ-*expressions* to perform *functional combination*. Their manipulation produces programs of the final iterative form.

One may conclude from the discussion that the new approach makes the transformation of the previous section superfluous. However, the following sections will show that correctness, meaning, and understanding of map expressions are based on the concepts and ideas of the first algorithm.

Map expressions are motivated by the Lisp operator *map*; the operator represents a loop—or linear-recursive—control structure without specifying its exact implementation. The meaning of the expression (*map f list*) with *f* being a function of arity one is defined as "applying function *f* to each element in the list."[8] Similarly, Backus introduced the α-operator in the FP language with the same semantics [2]. Usually, *map* takes a list of elements as its second argument and produces a new list of elements using *f*. We extend its use to produce *lists of functions* from lists of elements. This aspect is further discussed in the first Section 4.1.

[8] This explanation actually simplifies the description of operator *map*. For example, the second argument could be a list of pairs for a function *f* of arity two.

The *map* operator's high-level nature encourages the manipulation of control structure information by a simple, but powerful set of transformation rules. Several researchers have developed different sets of rules to improve the efficiency of FP programs [3, 4, 15]. For our purpose we introduce a small set of transformation rules to manipulate map expressions. Its application leads to a final canonical form representing a program with the minimal number of loops necessary to perform the specified evaluation. Map expressions in their canonical form are then translated into iterative expressions of the target language. A second set of transformation rules, with rules similar to the ones in $\mathcal{R}_{simp}$ derives the final iterative expressions.

4.1 Map Expressions and Lambda Expressions

For the manipulation of control structure information we introduce *map expressions*. They are modeled after the Lisp operator *map* that represents a loop—or linear-recursive—control structure without specifying its exact implementation. For the transformation we change operator *map* in two ways. First, for notational uniformity we use streams instead of lists and, for convenience, we use one element and a stream consisting of only one element interchangeably. The second change to *map* is concerned with the allowable range of functions provided as its first argument. For the transformation the first argument can be a *stream of functions* all of which have the same arity. For example, let + and * denote the addition and multiplication functions, respectively, then the expression (*map* < +, * > < 1, 3 >) produces a stream of four functions of arity one, i.e., < +1, +3, *1, *3 > where the first two functions add one and three to any given argument, and the last two functions multiply any given argument by one and three.

In general, if *map* applies a stream of functions *fstr* of arity n to a stream of constants *cstr*, the result is a stream whose elements are functions of arity $n - 1$. To distinguish between the original *map* definition and our extensions, we shall use *map** in the sequel. The result of (*map** *fstr cstr*) may be determined by evaluating one element of the function stream *fstr* against all elements of *cstr* or vice versa. Since the order of elements in streams is irrelevant in our context, both evaluations yield the same result.

For the manipulation of control structure information we translate QEPs into map expressions. We therefore redefine the DB functions of Section 2: Prj_i denotes the function prji that implements the projection of tuples according to parameter PRJ_i for action PROJECT. Similarly, $Pred_i?$ denotes function predi?, $Cjoin_i?$ represents the combination of functions joini? and conc. Additionally, we introduce the identity function *Id*. Using these functions we redefine all DB functions as follows:[9]

$$
\begin{array}{lcl}
(scan\ table) & = & (map*\ Id\ table) \\
(filteri\ str) & = & (map*\ Pred_i?\ str) \\
(projecti\ str) & = & (map*\ Prj_i\ str) \\
(ljoini\ str1\ str2) & = & (map*\ (map*\ (K\ Cjoin_i?)\ str2)\ str1)
\end{array}
$$

[9] For the definition, we use a table and a stream interchangeably.

Note that $Cjoin_i?$ is the only function of arity 2. For the first transformation we assumed that the first argument of $Cjoin_i?$ is bound to elements in *str*1; therefore we have to ensure that (*map** (K $Cjoin_i?$) *str*2) returns a stream with functions whose *second arguments are bound to elements in str*2. Informally, the operator K appearing in the definition of *ljoin* reverses the arguments of $Cjoin_i?$ to guarantee the correct binding of variables. Later in this section we define operator K more formally.

We introduce another operator, called L, that denotes the *combination* of two functions. We need this operator later during the transformation of map expressions. For example, consider the two functions *sqr* and *exp*, which represent the square function and the exponential function, respectively. Then (L *exp sqr*) denotes the function that first performs squaring before applying the exponential function *exp*. At the end of this section we define operator L more carefully.

As we mentioned in the introduction to this section, map expressions simplify the manipulation of control structure information. This first transformation step is followed by functional combination that we cannot describe as elegantly as in the previous section. For a formal description we introduce the *lambda notation* (λ-notation) of denotational semantics as presented in [23] and [24]. We restrict its use to the description of functional combination; for more details and further reference see [19, 23, 24]. The λ-notation becomes necessary to correctly describe functions newly created during the translation from map expressions into λ-expressions. As the number of parameters may vary, all parameters are represented explicitly.

For example, if "+" denotes the addition function then $\lambda t_1.\lambda t_2.\ (+\ t_1\ t_2)$ is called a λ-*expression*, or λ-*abstraction*, denoting the addition function with its formal parameters. This expression, when applied to argument 4 yields another function denoted by the λ-expression $\lambda t_2.\ (+\ 4\ t_2)$, which may be called the "add-four" function. Possible arguments to λ-expression include other λ-expressions to generate new λ-expressions denoting new functions. In general, if $\lambda t_1.u$ is a λ-expression that is *applied* to another λ-expression v the new expression is denoted $u[t_1/v]$. To avoid naming conflicts we assume the set of variables for both expressions to be disjoint. For our purposes a λ-expression is either any expression of the defined target language or an expression $\lambda t_i.u$ where u is a λ-expression. In the sequel $\mathscr{AP}$ denotes the exhaustive application of λ-expressions.

For functional composition in the second transformation step we define functions *Id*, Prj_i, $Pred_i?$, and $Cjoin_i?$ by λ-expressions:

$$
\begin{aligned}
Id &= \lambda t_1.t_1 \\
Prj_i &= \lambda t_1.(prj_i\ t_1) \\
Pred_i? &= \lambda t_1.(if\ (pred_i?\ t_1)\ t_1) \\
Cjoin_i? &= \lambda t_1.t_2.(if\ (join_i?\ t_1\ t_2)\ (conc\ t_1\ t_2))
\end{aligned}
$$

To formally describe functional composition in map expressions we introduce the operator L, which is defined by the λ-expression:

$$(L\ f_1\ f_2) = \lambda t'_1.\ \ldots\ \lambda t'_{n_1}.\lambda t_1.\ \ldots\ \lambda t_{n_1}.\ (f_1\ (f_2\ t'_1\ \ldots\ t'_{n_1})\ t_2\ \ldots\ t_{n_1})$$

with f_1 and f_2 being functions of arity m and n, respectively. The result of operator K was informally described by reversing the argument list for any function f.

Using a λ-expression we define the operator by

$$(K\ f_1) = \lambda t_1. \ldots t_n.\ (f_1\ t_n \ldots t_1)$$

for any function f_1 of arity n. Similarly, we introduced *map** as an operator representing a loop-like control structure. We define the operator in terms of a λ-expression for the second transformation step by

$$(map*\ f_1\ str) = \lambda t_2 \ldots t_n.\ (do\ (t_1\ str\ (rest\ t_1))\ (empty?\ t_1)\ (f_1\ (first\ t_1)\ t_2 \ldots t_n))$$

with f_1 being a function of arity n.

4.2 Structural Manipulation and Functional Composition

The previous section introduced map expressions and λ-expressions to represent intermediate forms in the translation of DB expressions. This section describes two sets of transformation rules that manipulate expressions in both forms. The first set of rules, which we call $\mathscr{R}_{map}$, manipulates the control structures using the following rules:

Rule I: $((map*\ f_1\ (map*\ f_2\ str)) \rightarrow (map*\ (L\ f_1\ f_2)\ str))$

Rule II: $((L\ f_1\ (L\ f_2\ f_3)) \rightarrow (L\ (L\ f_1\ f_2)\ f_3))$

Rule III: $((L\ f_1\ (map*\ f_2\ str)) \rightarrow (map*\ (L\ f_1\ f_2)\ str))$

Using the operator L for functional combination, the first rule implements "vertical loop" fusion; that is, two subsequent loops are combined into one, thus eliminating one intermediate stream. The second rule guarantees a canonical form for functional combination by grouping to the left. The last rule moves functions from the left-over control structures, thus allowing further function combination. Informally, the rule gives preference to functional combination over the application of functions to streams. The set of rules enjoys the confluent and noetherian properties [12].

The minimal form $T_{map}(t)$ of any map expression t derived by $\mathscr{R}_{map}$, reduces the number of *map** operators to the number of table variables used in the expression, thus later minimizing the number of loops necessary for evaluation.

LEMMA 4.1. *For any map expression m the number of map* operators in the minimal form is equal to the number of table variables.*

PROOF. The only rule that removes *map** operators is Rule I. We note that if $t = (map*\ f_1\ (map*\ f_2\ str))$ then the number of *map** operators in f_1 and f_2 is not changed when applying Rule I to t. Since rule set *Rm* enjoys the noetherian and confluent properties, we choose a derivation that initially applies Rule I only on expressions:

$$(map*\ f_1\ (map*\ \ldots\ (map*\ f_n\ str)\ \ldots))$$

for some n. By induction on n the expression reduces to

$$(map*\ (L\ (L\ \ldots\ (L\ f_1\ f_2)\ \ldots\ f_{n-1})\ f_n)\ str)$$

by applying Rule I $n - 1$ times. If f_i, $i = 1 \ldots n$ also contains expressions of the above form, the derivation continues the application of Rule I. □

The second set of transformation rules, called $\mathscr{R}_\lambda$, is concerned with the transformation of map expressions into the defined target language using lambda expressions. The rules are quite similar to Rules I, II and III in rule set $\mathscr{R}_{simp}$ of 3.2. They generate expressions that evaluate conditions before evaluating functions as parameters for other functions. The set $\mathscr{R}_\lambda$ contains the following rules:

Rules Ia, Ib, Ic: *Function Exchange Rules*

$((f_1\ (if\ t_1\ t_2)) \rightarrow (if\ t_1\ (f_1\ t_2)))$
$((f_2\ (if\ t_1\ t_2)\ t_3) \rightarrow (if\ t_1\ (f_2\ t_2\ t_3)))$
$((f_2\ t_1\ (if\ t_2\ t_3)) \rightarrow (if\ t_2\ (f_2\ f_1\ t_3)))$

Rule II: *Deletion Rule*

$((if\ t_1\ (\ldots\ (if\ t_1\ t_2)\ \ldots)) \rightarrow (if\ t_1\ (\ldots\ t_2\ \ldots)))$

Rule III: *Distribution Rule*

$((if\ (if\ t_1\ t_2)\ t_3) \rightarrow (if\ t_1\ (if\ t_3\ t_3)))$

Rule IV: *Loop Rule*

$((do\ (t_1\ t_2\ t_3\ t_4)\ (if\ t_5\ t_6)) \xrightarrow{C} (if\ t_5\ (do\ (t_1\ t_2\ t_3\ t_4)\ t_6)))$

with condition C that the actual loop variable represented by t_1 does not occur as a variable in expression represented by t_5.

Except for the last rule, all other rules in $\mathscr{R}_\lambda$ appear similarly in $\mathscr{R}_{simp}$. Unfortunately, functional composition requires Rule IV. The transformation into λ-expressions may create expressions with conditions whose evaluation does not depend on the loop at all. Those need to be removed from the loop as loop invariants and placed before entering the loop [12].

Using LAMBDA for translating map expressions to λ-expressions and using the set of transformation rules in $\mathscr{R}_\lambda$, we develop an algorithm which successively transforms map expressions into iterative programs of the target language. Let $t' = \mathscr{T}\mathscr{R}_\lambda(t)$ denote the result of applying $\mathscr{R}_\lambda$ exhaustively to expression t. As already defined in Section 4.1, $\mathscr{AP}$ denotes the exhaustive application of λ-expressions.

Algorithm $\mathscr{ML}$

Input: normalized map expression t **Output:** iterative expression
case
If $t \in \{Id, Prj_i, Pred_i?,$ and $Cjoin_i?\}$:
return (LAMBDA(t));
If $t = (K\ f)$:
return ($\mathscr{AP}$(LAMBDA(K), LAMBDA(f)));
If $t = (L\ f_1\ f_2)$:
$t_1 := \mathscr{ML}(f_1)$; $t_2 := \mathscr{ML}(f_2)$;
$t_3 := \mathscr{AP}$(LAMBDA(L), t_1, t_2); /* generate λ-expression */
$t_4 := \mathscr{T}\mathscr{R}_\lambda(t_3)$; /* simplify λ-expression */
return (t_4);

```
    If t = (map* f1 str):
    t1 := ML(f1);
    t2 := MP(LAMBDA(map*), t1, str);   /* generate λ-expression */
    t3 := TRλ(t2);   /* simplify λ-expression */
    return (t3);
  end_case;
```

Algorithm $\mathcal{ML}$ recursively combines functions and control structure information according to operators K, L, and *map**, and simplifies their combination using $\mathcal{TR}_\lambda$. The input to $\mathcal{ML}$ must be in canonical form, otherwise the algorithm does not perform its translation correctly.

4.3 The Transformation Algorithm

While the last section developed the transformation of map expressions into λ-expressions, this section presents the complete transformation algorithm to translate DB expressions into iterative programs. The algorithm calls function *Make_result*, which moves the variable *result* and function *out* into the final expression. *Make_result* is best described by the following two rules, which are applied repetitively to conditional expressions and loop statements:

$$((set\ result\ (out\ (if\ t_1\ t_2)\ result)) \rightarrow (if\ t_1\ (set\ result\ (out\ t_2\ result))))$$
$$((set\ result\ (out\ (do\ t_1\ t_2\ t_3)\ result)) \rightarrow (do\ t_1\ t_2\ (set\ result\ (out\ t_3\ result))))$$

Both rules push the assignment statement together with function *out* over conditional expressions and loop statements down to function expressions that compute the resulting tuples.

Algorithm $\mathcal{TR}_{map}$

Input: QEP qep **Output:** iterative expression

begin

```
    t1 := DB (qep);          /* translate qep into DB expression */
    t2 := T_map(MAP(t1));    /* transform map expressions */
    t3 := ML(t2);            /* transform λ-expressions */
    /* create assignment statements */
    t4 := (set result (out prg result)) [prg/t3];
    t5 := Make_result (t4);
    t6 := ((set result *empty*) prg) [prg/t5];
```

end

THEOREM 4.1. *Given a QEP qep accessing n tables, algorithm* $\mathcal{TR}_{map}$ *produces an iterative expression with n loops.*

SKETCH OF THE PROOF. Termination follows from $\mathcal{R}_{map}$ and $\mathcal{R}_\lambda$ being noetherian, and the termination of $\mathcal{ML}$. The termination of $\mathcal{ML}$ is guaranteed by $\mathcal{R}_\lambda$ being noetherian and by the decreasing numbers of functions that are combined with each recursive call. The number of loops in the final output is determined by the number of *map** operators. By Lemma 2 the number of *map**'s in the minimal expression is equal to the number of tables, thus leading to an iterative program of n loops. □

To estimate the running time of algorithm $\mathcal{TR}_{map}$, let the initial QEP consist of n actions and m tables. For any well formed QEP $n \geq m$. The initial map expression contains $n + m - 1$ *map** operators with n functions as their first arguments. The first rule of $\mathcal{R}_{map}$ is applied $n - 1$ times, the second rule at most n times and the last rule at most $n * m$ times. The resulting time complexity is $O(n^2)$. The minimal form consists of at most $n + m$ L operators, m K operators, and exactly m *map** operators. In Algorithm $\mathcal{ML}$ the application $\mathcal{AP}$ of λ-expressions for operators L and K takes constant time. In each combination each rule in $\mathcal{R}_\lambda$ is applied at most n times, so $\mathcal{ML}$ has a time complexity of $O(n^2)$. Since the number of translations between the different expressions is linear in the number of operators, the overall complexity is $O(n^2)$.

5. DISCUSSION AND CONCLUSIONS

5.1 Comparison of Algorithms

This section compares the two transformation algorithms $\mathcal{TR}_{rec}$ and $\mathcal{TR}_{map}$ developed in this paper. We discuss their strengths and weaknesses and show how they supplement each other despite their differences.

So far we have carefully avoided the question of whether or not both transformation algorithms generate the same programs. Theorems 3.1 and 4.1 guarantee that programs generated from the same QEP contain the same number of loops. Since the transformations performed by both algorithms are sound, the generated programs are computationally equivalent. They are also syntactically identical. However, to prove the latter is more difficult since the transformations proceed differently. The examples provided for both transformations strongly suggest that the resulting programs are indeed the same. Instead of making a laborious effort for proving syntactic equivalence we prefer to compare both transformation algorithms more informally and point out differences that make it cumbersome to translate recursive expressions to map expressions or λ-expressions.

One of the major differences is the kind of intermediate results we obtain during the transformation process. The recursion-based transformation algorithm $\mathcal{TR}_{rec}$ always derives expressions whose evaluation leads to constant streams. Map subexpressions may result in function streams, making it necessary to keep track of the list of unbound variables during the transformation into the target language. For this reason we also have to introduce the auxiliary operator K to guarantee the correct order of variable bindings.

This observation is closely related to two other major differences. First, to obtain expressions that always yield constant streams, algorithm $\mathcal{TR}_{rec}$ alternates between the manipulation of recursive expressions and the generation of loops. On the other hand, the transformation of map expressions is completed first before any translation into iterative expressions is performed. Second, the normal form of map expressions encourages a "bottom up" or "innermost first" translation into iterative programs. Algorithm $\mathcal{TR}_{map}$ produces the innermost loop of the iterative program first, the outermost loop last. On the other hand, the recursion-based transformation generates programs "top down": To produce the outermost loop first the computation of the ideal form reverses the functional order by unfolding, simplification, and folding. The alternation of unfolding and folding is performed repetitively without much progress in the overall

transformation. The separation of control structure information and functional composition in map expressions avoids these time-consuming steps. Intermediate streams are eliminated using only one rule. The functional combination is delayed until the final iterative programs are generated. Much effort is needed to correctly perform functional combination; we must introduce λ-expressions. In the recursion-based transformation the use of stream operators implemented functional combination elegantly and without additional overhead.

5.2 Conclusions

We see the major contributions of this paper to be the following:

(1) The successful application of functional programming and program transformation in the area of database systems.
(2) The development of two transformation algorithms to solve the query translation problem. Without user intervention both algorithms systematically generate structured iterative programs from modular query specifications. The first algorithm is based on recursive function definitions; the second takes advantage of map expressions motivated by Backus' FP language.

In [12], we included into both algorithms additional QEP operations to manipulate indexes as well without changing the transformation algorithms at all. Furthermore, we also extend them to generate iterative programs that evaluate two queries in "parallel." If these queries scan the same relation, we can avoid the second relation scan for the evaluation of the second query by simply "merging" both scans into one.

The results in this paper demonstrate that ideas and techniques of functional programming and program transformation help to solve problems in the area of database systems. Although the results are encouraging, more evidence is necessary to show that the approach taken in this paper is powerful enough to solve other problems for database systems as well. We suggest investigating the following problems:

(1) A more complete set of operators: We would like to include actions that create intermediate streams during their evaluation, allow to sort streams, or implement other join methods, such as the merge-join.
(2) The application of the transformational approach to other problems of database systems: For example, Shasha describes general operations on data structures that include locking operations; the data structure itself is left unspecified [21]. When a particular data structure is chosen, we would like to derive efficient operations for its manipulation using the transformational approach.

The work of [21] leads to the more general question of how to support the modular design and implementation of database systems in general. Currently, the different components in most database systems are highly integrated, which makes desirable changes almost impossible. A high-level specification of the various components, like the user interface, the query optimizer, the access method system, the concurrency control component, or the logging and recovery component, could make changes easier. A powerful transformation system, possibly including some user guidance, could help to generate database systems that

meet the demands for high performance. For this purpose more work is necessary to better understand the interaction between different components in a database system.

APPENDIX A: A SAMPLE PROOF

This appendix gives one example proof to show that the combination of two 1-stream functions yields an ideal form.

LEMMA A.1 *If f_1 is of the form*

```
(f1 e1 ... em str1) =
   (if (empty? str1)  *empty*
      (out (g1 e1 ... em (first str1)) (f1 e1 ... em (rest str1))))
```

and f_2 is of the form

```
(f2 e1 ... em str2) =
   (if (empty? str2)  *empty*
      (if (pred? (first str2))
         (out (g2 e1 ... em (first str2))
            (f2 e1 ... em (rest str2)))
         (f2 e1 ... em (rest str2))))
```

then any of their combinations has an ideal form derivable by UNFOLD, SIMPLIFY *and* FOLD.

PROOF. Consider the case of s (f_1, f_2, 1):

Using the simplification SIMPLIFY the rule sequence $\mathscr{D}_R$ = (*I*, *I*, *III*, *III*, *Vc*, *Vd*, *IVa*, *IVb*, *IIb*, *IIb*, *IIb*, *IIa*, *IIa*, *IIb*, *Va*, *Vb*) yields

```
(if (empty? str2)  *empty*
   (if (pred? (first str2))
      (out (g1 (g2 (first str2))) (f1 (f2 (rest str2))))
      (if (empty? (f2 (rest str2)))
         (out (g1 (first (f2 (rest str2))))
         (f1 (rest (f2 (rest str2)))))))))
```

The folding of the expression by f_1 leads to the ideal form:

```
(if (empty? str2)  *empty*
   (if (pred? (first str2))
      (out (g1 (g2 (first str2))) (f1 (f2 (rest str2))))
      (f1 (f2 (rest str2)))))
```

The case of $s(f_2, f_1, 1)$ is similar. □

ACKNOWLEDGMENTS

We would like to thank the referees for many helpful comments that have improved the clarity and readability of this paper. The first author would also like to thank Laura Haas at the IBM Almaden Research Center, San Jose, CA, for her support in preparing this paper for its submission.

REFERENCES

1. ASTRAHAN ET AL. SYSTEM R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.

2. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. In *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
3. BELLEGARDE, F. Rewriting systems on FP expressions that reduce the number of sequences they yield. In *ACM Symposium on LISP and Functional Programming* (Austin, Tex., Aug. 1984). ACM, New York, 1984, 63–73.
4. BIRD, R. S. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst. 6*, 4 (Oct. 1984), 487–504.
5. BUNEMAN, P., AND FRANKEL, R. E. An implementation technique for database query languages. *ACM Trans. Database Syst. 7*, 2 (June 1982), 164–186.
6. BURSTALL, R. M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
7. CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
8. COHEN, N. H. Source-to-source improvement of recursive programs. Ph.D. dissertation, Harvard University, May 1980.
9. COLUSSI, L. Recursion as an effective step in program development. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan. 1984), 55–67.
10. DANIELS, D. Query compilation in a distributed database system. Tech. Rep. RJ 3432, IBM Research Laboratory, San Jose, CA, Mar. 1982.
11. DARLINGTON, J., AND BURSTALL, R. M. A system which automatically improves programs. In *Acta Inf. 6*, 1 (1976), 41–60.
12. FREYTAG, J. C. Translating relational queries into iterative programs. Ph.D. dissertation, Harvard University, Sept. 1985.
13. FREYTAG, J. C., AND GOODMAN, N. Rule-based translation of relational queries into iterative programs. In *ACM-SIGMOD Conference on Management of Data* (Washington, D.C.), ACM, New York, 206–214.
14. FRIEDMAN, D. P., AND WISE, D. S. CONS should not evaluate its arguments. In *Automata, Languages, and Programming*, S. Michaelson and R. Milner, eds., Edinburgh University Press, Edinburgh, 1976, 257–284.
15. GIVLER, J. S., AND KIEBURTZ, R. B. Schema recognition for program transformation. In *ACM Symposium on LISP and Functional Programming* (Austin, Tex., Aug. 1984). ACM, New York, 1984, 74–84.
16. HENDERSON, P., AND MORRIS, J. H. A Lazy Evaluator. In *Third ACM SIGACT-SIGPLAN Symposium of Principles of Programming Languages* (Atlanta, Ga., 1976). ACM, New York, 1976, 96–103.
17. HUET, G. Confluent reductions: Abstract properties and applications of term rewriting systems. *J. ACM 27*, 4 (Oct. 1980), 797–821.
18. LORIE, R. A., AND WADE, B. W. The compilation of a high level data language. Tech. Rep., IBM Research Laboratory, San Jose, CA, 1979.
19. MEYER, A. What is a model of lambda calculus? *Inf. Contr. 52*, (1982), 87–122.
20. ROSENTHAL, A., AND REINER, D. An architecture for query optimization. In *ACM-SIGMOD Conference on Management of Data* (Orlando, Fla., 1982). ACM, New York, 1982, 246–255.
21. SHASHA, D. Concurrent algorithms for search structures. Ph.D. dissertation, Harvard University, May 1984.
22. STONEBRAKER, M., WONG, E., ET AL. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
23. STOY, J. E. *Denotational Semantics*. MIT Press, Cambridge, Ma., 1977.
24. TENNENT, R. The denotational semantics of programming languages. *Commun. ACM 19*, 8 (Aug. 1976), 437–453.
25. ULLMAN, J. D. *Principles of Database Systems*. Computer Science Press, San Francisco, 1982.
26. WILLIAMS, J. H. Notes on the FP Style of Functional Programming. In *Functional Programming and its Applications*. J. Darlington, P. Henderson, and D. A. Turner, eds. Cambridge University Press, 1982, 193–215.

Received November 1986; revised September 1987 and February 1988; accepted January 1989