Using Type Transformation in Database System Implementation †

Leonidas Fegaras David Stemple

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

fegaras@cs.umass.edu, stemple@cs.umass.edu

Abstract

We propose the use of semi-automatic methods for the translation of abstract database programs into efficient lower level user-defined primitives. We present a type transformation model for use in defining and facilitating the query translation and optimization process. We discuss how this model can be used also as a framework for schema evolution and data restructuring.

1 Introduction

Current database management systems, such as systems based on the relational model, lack the expressive power to support complex queries in a high level language and typically provide limited variety in their data structures. The limitation on the expressive power of the high level languages in these systems is commonly circumvented by embedding query languages in more expressive conventional programming languages. The disadvantage of this approach is that it forces the database designer to express his/her conceptual model in terms of two different semantic models. Additionally, for programming language objects to become database objects (or vice versa) they often need to be restructured into the data model of the database system being used. Such restructuring incurs costs both in conceptual terms for programmers and in resources used when performed. Examples of this problem occur in applications such as rule-based systems and computer-aided design. In these applications complex structures are often used in programming the algorithms that access rules and designs. These programming language structures need to be "flattened" in order to be stored in relational structures, for example. The cost of restructuring can make it difficult to meet the efficiency requirements of these applications.

One important advantage of commercial database systems is that they offer data independence, whereby abstract objects and the operations upon them can be significantly independent of their implementations. In a relational database system, for example, a database designer may choose the implementation of a database table from a number of possible implementations, and this decision will not affect how queries are expressed in the database language but only how they are compiled and optimized. Furthermore, some of these systems provide a restructuring mechanism to change the implementation of parts of the database or to modify the database schema itself without losing any stored data.

On the other hand, most programming languages offer satisfactory computational power and a large variety of data objects to choose from but support limited persistence, usually in the form of files. Some of these languages provide an abstract data type mechanism to support a layered model of programming, where abstract objects and the operations upon them are defined in terms of other more implementation oriented operations. This layering provides a type of data independence because, as in the database systems, it hides the implementation details from the user of the abstract data type. The difference is that an abstract data

†This paper is based on work supported by the National Science Foundation under grants IRI-8606424 and IRI 8822121 and by the Office of Naval Research University Research Initiative contract, number N00014-86-K-0764

type typically has its operations decomposed into their implementations in a predefined, rigid way, while a database system offers a number of alternative ways of execution. For example, there may be more than one access path to retrieve the same piece of information from a database. This redundancy of access paths makes it possible for a database system to perform query optimization, in which concrete implementations of abstract expressions are generated from alternatives, rather than expanded hierarchically through layers of abstraction. A rigid refinement style of implementation does not meet the needs of large data-intensive applications. It does not offer sufficient flexibility to solve the problems of evolution of such systems and it does not support optimization techniques exemplified by query optimization in database systems.

There has been an effort to bridge database management technology with programming language philosophy to form persistent and database programming languages [1]. Objects defined in such languages can have sophisticated structures, like those found in modern programming languages, and can be persistent across a wide variety of types. Persistent and non-persistent objects are treated more uniformly than in ordinary languages, hiding the difference in persistence from the programmer of the operations that work on these objects. Most of these systems suffer from the same problem that non-persistent programming languages have: all abstract objects and operations are implemented via rigid translations. For data intensive applications, where most of the data needs to be persistent, this problem becomes more important because the database accesses are relatively slower than the memory ones. This may result in a system that is unnecessarily slow, because the programmer does not have enough flexibility in choosing how abstract objects are stored and the translator does not examine alternative ways to perform the queries using different access paths and algorithms.

In this paper we examine the problem of achieving data independence in persistent and database programming languages. We propose a development method and a formal model for transforming data-intensive programs in a high level, strongly typed language into efficient implementations in a manner that will support optimization and system evolution. In the next section we present our development methodology in which several roles for specifiers and implementors are distinguished. Following this we briefly discuss work that is closely related to ours. Then we give our type transformation model, first by example and then formally. We then discuss how the type transformation model accommodates query optimization and system evolution.

2 Our Methodology

In this paper we explore the possibility of achieving a level of data independence similar to that found in database systems but in the context of a high-level database programming language, such as ADABTPL [7]. We believe that the best approach to this problem is to give the database designer the ability to specify how the abstract objects defined in a program are to be mapped into the storage structures provided by the database, as well as to specify the implementation of some of the abstract operations, but leave the compiler to translate and optimize the rest. A database storage structure in this approach is not just an abstract data type that implements an abstract object. Here the designer must give explicitly the *dependency* between the abstract and the storage objects. A well-suited means of expressing this dependency is the use of a representation function to map the concrete object into the abstract object. The compiler will use these dependencies to translate the operations and to assure the correctness of the translation. We have developed a formal model that describes this process, called the **type transformation model**, based on Darlington's work on transformational programming [6, 5, 14]. The dependencies between abstract objects, which can be expressed as integrity constraints, as well as the dependencies between the storage objects suggest that there may be more than one access path to retrieve the same data, since there is redundant information. The query translation process is a search over these alternatives guided by heuristics and cost functions.

The language that we are using for this system is ADABTPL [7]. ADABTPL is a database programming language with functional semantics. We have developed technology for transforming a high-level specification in ADABTPL into an application specific database theory [16]. The database theory is used by a theorem prover to validate parts of the design and to assure functionality. This theorem prover, based on the work of Boyer and Moore [4], has already been used to prove transaction safety [16], that is, whether a transaction leaves the database in a consistent state, and to provide rich feedback to the designer in case of inconsistency [17]. We intend to use the database theory developed for validating specifications to verify all stages of the compilation, that is, to assure that the implementation, at least the form that is the target of the transformation stage, satisfies the design specifications. Our technology is not designed to verify the transformations into the lowest level of implementation, which could be in some formally intractable

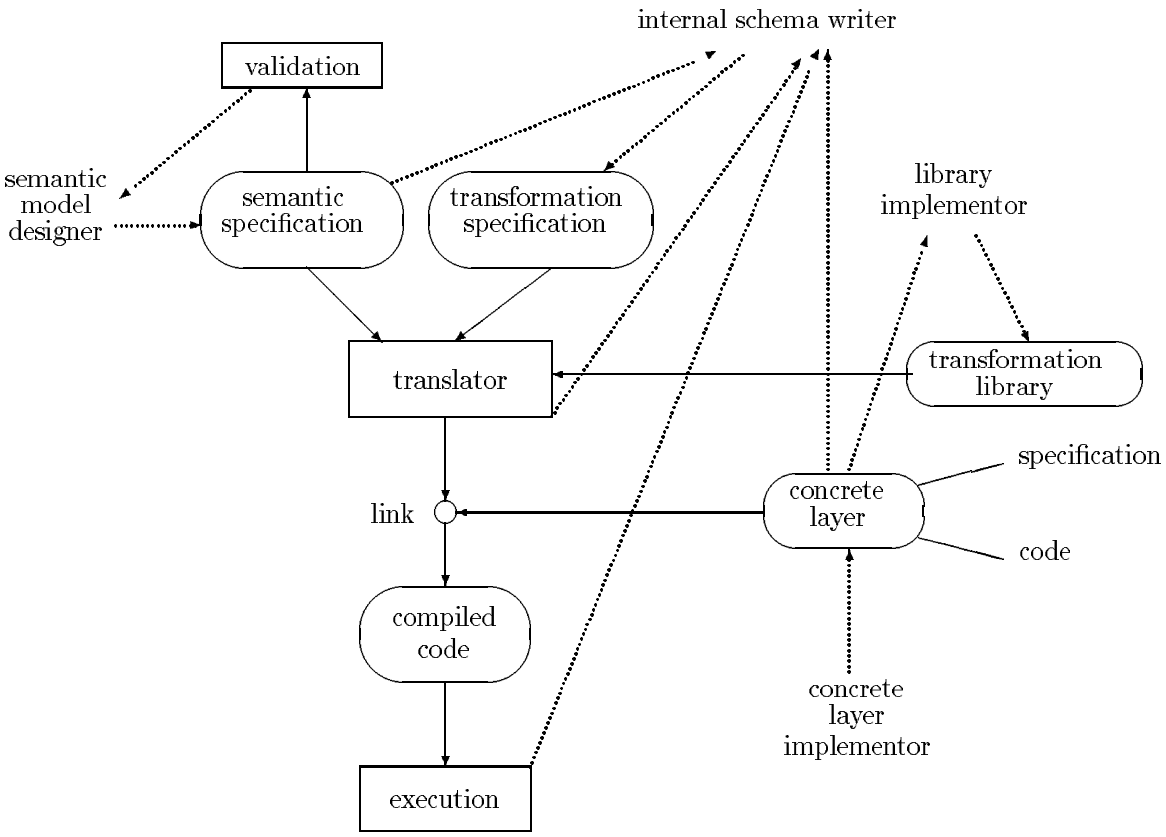


Figure 1: The translation process (dotted lines represent feedback and control)

language such as C.

The necessary separation of specification from implementation requires that the translation be done in stages by people with different expertise. Figure 1 shows the necessary stages for compiling an ADABTPL program. The **semantic model designer** is the person who writes the abstract application program (the *semantic specification*) using the ADABTPL language. This person should not be concerned with assigning storage structures to the abstract objects to achieve efficient execution. The main concern here is to write a functionally correct specification satisfying all the design requirements. The ADABTPL translator type checks the specification and generates an application specific database theory to help the theorem prover find inconsistencies in the specification and suggest corrections to overcome them. The semantic model designer can compile the specification into a non-persistent prototype using a simple translation, not shown in the figure, in order to check the behavior of the specified system. The whole design process can be repeated until the specification is consistent and the specified system behaves as intended.

After the program is proved to be syntactically and semantically correct, a second stage is initiated by the **internal schema writer** who assigns storage structures to all the abstract objects by defining explicitly the type transformations that will achieve the object translations. All these type transformations will constitute the *transformation specification* of the abstract program. To make this task easier, a rich library of type transformations, called the *transformation library*, will be provided. This library contains the transformations that are commonly used, such as the mappings of abstract sets into B^+trees, attaching indexes to a set, etc. The task of writing this library is assigned to the **library implementor**. The internal schema writer is left with the task of carefully selecting mappings from the library that suit this specific application, and defining his/her own mappings when none of the mappings in the library are satisfactory. The code that manipulates the database storage structures constitutes the *concrete layer* and it is written by the **concrete layer implementor.** The internal schema writer as well as the library implementor need to know about this layer in order to write valid mappings. If the concrete layer is changed, usually by the introduction of new storage structures, this will affect both the transformation library and the transformation specification.

The task of the compiler is to accept the semantic specification along with the transformation spec-

ification and produce the interface to the concrete layer. The theorem prover is used again to verify the correctness of the translation by proving whether the translation of each abstract operation manipulates the storage structures according to the defined object mappings. After this, the produced object code must be linked with the concrete layer library to yield executable modules. The internal schema writer will evaluate the performance of some of the functions and of the whole system to decide if there is an alternative way to implement some of the abstract objects to improve the system performance. The process of the transformation specification is repeated until the system performance is acceptable.

It is very common for either the system specification or implementation to change, because the needs of most applications evolve through time. Decisions about schema evolution are made by the semantic model designer, while the internal schema writer is responsible for the changes in the implementation. It is desirable to restructure the database efficiently to reflect these changes, instead of destroying and recreating it from scratch. This task is performed by the internal schema writer who writes the required mappings to transform the old abstract objects to the new abstract objects. These mappings are the same as the mappings from abstract objects to concrete. The compiler will use this information to generate a program that restructures the database. Accommodation of system evolution has been one of the main motivations for our approach and constitutes one of the criteria of success for the transformation work.

3 Related Work

The Genesis extensible database management system developed at the University of Texas at Austin [2, 3] is a project with similar goals and motivations. Genesis introduced a technology that enables customized database management systems to be developed rapidly, using user-defined modules as building blocks. A transformation model is used to map abstract models to concrete implementations. This is done with possibly more than one level of conceptual to internal mappings, transferring abstract models to more implementation-oriented ones, until a primitive layer is reached. These mappings are a sequence of database definitions that are progressively more implementation-oriented. The programs that translate abstract schemas into concrete schemas, called type transformers, are written by the Genesis database implementor, whose role is similar to the role of an internal schema writer. The database implementor is also responsible for writing the program transformers for each type transformer, called the operation expanders, that translate every operation on an abstract type to a sequence of operations on the concrete type.

Other work that adopts a similar approach towards a user-controlled translator is the Polya project [9]. Polya, developed at Cornell University, introduces a new programming language construct, called the transform, for expressing coordinate transformations. Each transform includes a coordinate transformation from a set of abstract variables to a set of concrete variables and explicit transformation rules to map each individual expression or statement that works on abstract variables into the corresponding one that works on concrete variables. Each part of an abstract program must match exactly with one of these patterns in order to be compiled.

A lot of work has been done on query optimization for commercial database management systems [13, 15]. Most optimizers have been designed having a specific semantic model in mind. The EXODUS optimizer generator [8] is an exception. EXODUS allows the database implementor to write a customized optimizer or to improve an existing one by providing explicit rules. The input to the optimizer generator is a model description file, where the database implementor lists the set of operators of the data model, the set of 'methods' to be considered when building and comparing access plans, the transformation rules that define legal transformations on the query trees, and the implementation rules that associate methods with operators. Each transformation rule is associated with an expected cost factor, which is derived automatically by the optimizer by learning from its past experience. The optimizing process is a hill climbing method guided by these cost factors.

4 The Type Transformation Model

This section presents the type transformation model which is an extension of the work done by Darlington and others [6, 5, 14]. Their model is enhanced to include parametric types and high order polymorphic functions, and its definition of coding functions is expanded to cover a wider class of mappings. This model is used for transforming abstract operations in terms of concrete operations, in conformance with the storage

structures assigned to the abstract objects which these abstract operations manipulate. Before describing the formal specification of the type transformation model, we present some examples that illuminate the basic idea.

The language used in this and subsequent sections is ADABTPL [7], which is a strongly-typed functional programming language [11]. Bulk types, like lists and trees, are defined as parametric recursive types. For example, the `list` type is specified as:

```
list(alpha) =
   union ( null: singleton nil,
           non_null: struct cons [ head: alpha, tail: list(alpha) ] );
```

This recursive definition says that a list is either a `nil` object (of a type constructed by the `singleton` type constructor) or a tuple (constructed by the `struct` type constructor) that has two components: a component `head` that holds a value of type `alpha`, and a component `tail` that holds the rest of the list. It defines two list constructor functions, the nullary constructor `nil` and the function `cons` of type `[alpha,list(alpha)]->list(alpha)`. It also defines the two selector functions, `head` and `tail`.

In the following we will use reduction functions over recursive types and finite sets to express transformations. There are three reasons for this. First, this does not unduly limit expressiveness, since all primitive recursive functions can be coded using the reduction function of a recursive type [12]. Second, reductions are convenient abstractions for expressing manipulations of bulk data types represented as recursive types. And third, we believe theorems about reductions will prove to be effective tools for reasoning about transformations among recursive data and finite sets, just as they were crucial to reasoning about maintaining integrity constraints [16].

Function `list_reduce` is a reduction over lists:

```
function(alpha,beta) list_reduce
    ( x: list(alpha), acc: [beta,beta]->beta, appl: [alpha]->beta, base: beta ) : beta;
case x
{ nil -> base;
  cons(a,r) -> acc(appl(a),list_reduce(r,acc,appl,base))
};
```

This function accumulates, using function `acc`, the result of a mapping function `appl` over the values in list `x`. Here the `case x` statement pattern-matches the value of `x` with the patterns `nil` and `cons(a,r)`. If `x` matches `cons(a,r)` then variables `a` and `r` are bound to the head and the tail of the list `x`. An example use of `list_reduce` is given by the length function expressed as `list_reduce(x,plus,[x]->1,0)`, where $[x_1,\ldots,x_n] \rightarrow exp$ is a lambda abstraction with variables $x_1,\ldots,x_n$ and body exp (expressed as $\lambda x_1 \ldots \lambda x_n.exp$ in lambda calculus).

The set reduction function is `set_reduce`, very similar to `list_reduce`:

```
function(alpha,beta) set_reduce
    ( s: set(alpha), acc: [beta,beta]->beta, appl: [alpha]->beta, base: beta ) : beta;
if s=emptyset then base
   else acc(appl(choose(s)),set_reduce(rest(s),acc,appl,base));
```

The `choose` function returns an arbitrary element of its input set. The `rest` function returns a set equal to its input with the `choose` element deleted. For example, the set union `union(s1,s2)` can be expressed as `set_reduce(s1,insert,[x]->x,s2)`. That is, we insert every element of `s1` into a copy of `s2`. Note that set `s2` is not destructively modified by this operation because ADABTPL is a functional language and therefore it uses copy semantics for passing parameters.

We present now an example of implementing set objects. Suppose that we want to implement objects of type `set(alpha)` as ordered lists. An ordered list could have the following definition:

```
sequence(alpha) =
  struct makeseq ( info: list(alpha),
                   before: [alpha,alpha]->boolean )
         where(x) ordered(x.info,x.before);
```

The `info` component of this type holds all values in the form of an ordered list. Function `before` is a partial order that serves as the sorting function. The integrity constraint of a type expresses a predicate that any object of this type must satisfy. Here the integrity constraint `where(x) ordered(x.info,x.before)` expresses the requirement that every object `x` of type `sequence(alpha)` has `x.info` ordered by `x.before`, where `ordered` is defined by:

```
function(alpha) ordered
   ( x: list(alpha), f: [alpha,alpha]->boolean ) : boolean;
case x
{ cons(a,cons(b,r))
        -> if f(a,b) then ordered(cons(b,r),f)
                     else false;
  other -> true
};
```

A reduction over sequences is `seq_reduce` defined as:

```
function(alpha,beta) seq_reduce
    ( x: sequence(alpha), acc: [beta,beta]->beta,
      appl: [alpha]->beta, base: beta ) : beta;
makeseq(list_reduce(x.info,acc,appl,base),x.before);
```

Type `sequence(alpha)` is a convenient implementation for sets because there is an isomorphism between set objects and sequence objects. More specifically, if `x` is a set and `y` is the sequence implementation of `x`, then:

```
y  =  set_reduce(x,seq_insert,[z]->z,makeseq(nil,order_fun))        (1)
```

where `seq_insert(e,s)` inserts the element `e` into the sequence `s` such that `s.info` remains ordered:

```
function(alpha) seq_insert ( e: alpha, s: sequence(alpha) ) : sequence(alpha);
makeseq(ordered_insert(e,s.info,s.before),s.before);
```

where `ordered_insert(e,x,before)` inserts `e` into the ordered list `x`:

```
function(alpha) ordered_insert
    ( e: alpha, x: list(alpha), before: [alpha,alpha]->boolean ) : list(alpha);
case x
{ nil -> cons(e,nil);
  cons(a,r) -> if before(e,a)
               then if before(a,e) then x else cons(e,x)
               else cons(a,ordered_insert(e,r,before))
}
```

We can rewrite equation 1 as `y=cod(x,order_fun)`, where `cod` is:

```
function(alpha) cod
    ( x: set(alpha), order_fun: [alpha,alpha]->boolean ) : sequence(alpha);
set_reduce(x,seq_insert,[z]->z,makeseq(nil,order_fun));
```

We call function `cod` the **coding function** of this mapping. Function `order_fun` is an extra parameter to the coding function that determines how the resulting sequence will be ordered.

The inverse mapping for implementing sets as sequences is:

```
x = rep(y) = seq_reduce(y,insert,[z]->z,emptyset)
```

We call function `rep` the **representation function** that interprets sequences as sets:

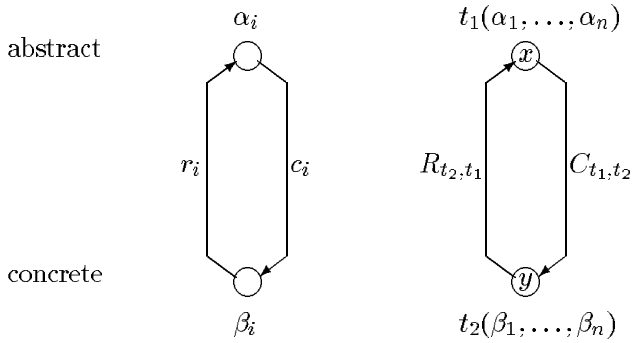
```
function(alpha) rep ( y: sequence(alpha) ) : set(alpha);
seq_reduce(y,insert,[z]->z,emptyset);
```

Therefore, to map sets into sequences one must choose an ordering function `order_fun` that will determine the implementation of a set. That way the mapping becomes isomorphic.

4.1 The Formal Framework

The types we use in this work are all formalized by algebraic axioms. Types constructed using `union` (labelled disjoint sum), `struct` and `singleton` along with the devices of parameterization and well-founded recursion are formalized using axioms sets that are basically equivalent to Hoare's axioms for recursive data structures [10]. We will refer to these types as recursive types. Finite set theory cannot be captured in the framework of recursive types. In our work we use a finite set theory based explicitly on a set of axioms as similar to recursive type axioms as possible [16].

Let $t_1(\alpha_1,\ldots,\alpha_n)$ and $t_2(\beta_1,\ldots,\beta_n)$ be two parametric types, where α_i and β_i are type parameters. A representation function R_{t_2,t_1} that maps the type $t_2(\beta_1,\ldots,\beta_n)$ into the type $t_1(\alpha_1,\ldots,\alpha_n)$ is a high-order function of type $(t_2(\beta_1,\ldots,\beta_n) \times (\beta_1 \rightarrow \alpha_1) \times \cdots \times (\beta_n \rightarrow \alpha_n)) \rightarrow t_1(\alpha_1,\ldots,\alpha_n)$. In the context of this mapping, t_1 is referred to as the **abstract type** and t_2 as the **concrete type**. If y is an instance of the concrete type t_2, then there is an instance x of the abstract type t_1 such that $x = R_{t_2,t_1}(y, r_1,\ldots,r_n)$. Each r_i is a representation function for mapping β_i into α_i.



For example, the representation function that interprets sequences as sets is:

```
function(alpha,beta) rep ( y: sequence(beta), r1: [beta]->alpha ) : set(alpha);
seq_reduce(y,insert,r1,emptyset);
```

Isomorphic mappings are of special interest. In isomorphisms there is a one-to-one correspondence between the objects of the abstract and concrete types. In this case, there is the inverse function of the representation function, called the coding function. More specifically, for the mapping with representation function R_{t_2,t_1} the coding function is $C_{t_1,t_2}(x, c_1,\ldots,c_n)$, if $\lambda y.R_{t_2,t_1}(y, r_1,\ldots,r_n)$ is the inverse of $\lambda x.C_{t_1,t_2}(x, c_1,\ldots,c_n)$, where each c_i is the inverse of r_i.

We can generalize this definition of coding function to include non-isomorphic mappings by adding the extra parameters $e_1,\ldots,e_m$ to C_{t_1,t_2}. More specifically, $C_{t_1,t_2}(x, c_1,\ldots,c_n, e_1,\ldots,e_m)$ is a coding function associated with the representation function $R_{t_2,t_1}(y, r_1,\ldots,r_n)$, if each c_i is a coding function associated with the representation function r_i and

$$\forall e_1 \ldots \forall e_m \forall x : \; R_{t_2,t_1}(C_{t_1,t_2}(x, c_1,\ldots,c_n, e_1,\ldots,e_m), r_1,\ldots,r_n) \;=\; x \qquad (2)$$

For example, the coding function for mapping sets into sequences is:

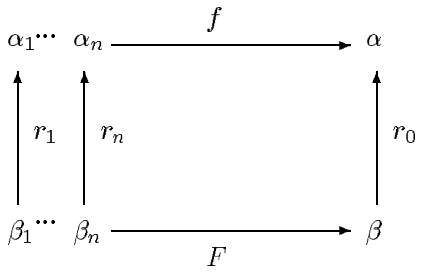
```
function(alpha,beta) cod
    ( x: set(alpha), c1: [alpha]->beta, before: [beta,beta]->boolean ) : sequence(beta);
set_reduce(x,seq_insert,c1,makeseq(nil,before));
```

where `c1` is the coding function from `alpha` to `beta` and `before` is an extra parameter (parameter e_1 of C_{t_1,t_2}).

Each homomorphic mapping defined by a representation function and a coding function that has some extra parameters can be considered as a parameterized isomorphism, that is, it becomes an isomorphism whenever these parameters are instantiated. This is very convenient, as it captures many possible implementations of an abstract type and uses a similar theory to that of isomorphisms that makes the analysis easier.

We consider now the case of mapping functions into functions. Let $t_1(\alpha_1,\ldots,\alpha_n,\alpha) = \alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha$ and $t_2(\beta_1,\ldots,\beta_n,\beta) = \beta_1 \times \cdots \times \beta_n \rightarrow \beta$. That is, function F of type $\beta_1 \times \cdots \times \beta_n \rightarrow \beta$ is mapped into a

function f of type $\alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha$, using the representation functions $r_i : \beta_i \rightarrow \alpha_i$ and $r_0 : \beta \rightarrow \alpha$



If in addition we require that all mappings are homomorphic then the above diagram commutes. That is, for all $x_1, \ldots, x_n$ we have

$$r_0(F(x_1, \ldots, x_n)) \quad = \quad f(r_1(x_1), \ldots, r_n(x_n)) \tag{3}$$

We call any function F that satisfies the above equation an **implementation** of f. If c_0 is a coding function for the mapping of α into β, then one solution for F is:

$$F(x_1, \ldots, x_n) \quad = \quad c_0(f(r_1(x_1), \ldots, r_n(x_n)), e_1, \ldots, e_m) \tag{4}$$

where $e_1, \ldots, e_m$ are extra parameters that need to be provided to the coding function c_0. For the simple case of an isomorphism, equation 4 becomes:

$$F(x_1, \ldots, x_n) \quad = \quad c_0(f(r_1(x_1), \ldots, r_n(x_n))) \tag{5}$$

In equation 3 we assumed that all r_i were unary functions. For a polymorphic function, where input/output parameters have parametric types, the representation and coding functions may have forms similar to R_{t_2,t_1} and C_{t_1,t_2}. In that case, equation 3 becomes:

$$r_0(F(x_1, \ldots, x_n), r_{0,1}, \ldots, r_{0,k_0}) \quad = \quad f(r_1(x_1, r_{1,1}, \ldots, r_{1,k_1}), \ldots, r_n(x_n, r_{n,1}, \ldots, r_{n,k_n})) \tag{6}$$

and one solution for F is:

$$F(x_1, \ldots, x_n) \quad = \quad c_0(f(r_1(x_1, r_{1,1}, \ldots, r_{1,k_1}), \ldots, r_n(x_n, r_{n,1}, \ldots, r_{n,k_n})), c_{0,1}, \ldots, c_{0,k_0}, e_1, \ldots, e_m) \tag{7}$$

Homomorphisms with no coding function are difficult to work with. Function F cannot be deduced directly from equation 3, as r_0 is not a one-to-one function. For that reason, we require that all mappings be defined in terms of both a representation and a coding function (in its general form with extra parameters). A theorem prover can be used to prove whether this pair of functions satisfies equation 2.

It is obvious from equation 5 that isomorphic mappings produce only one function F that implements f, even though there are different programs, some more costly than others, that are equivalent to F. In contrast, non-isomorphic mappings express function F in terms of some extra arguments to be provided before the translation, as it is stated in equation 4. When these arguments are provided, they pin down the implementation of f. These extra parameters must not be seen as a problem but as a source of alternative ways of implementing f and, therefore, as increasing opportunities for optimization. Choosing the right values for the extra parameters can significantly improve the performance of the implementation.

The translation process of an abstract program starts by assigning concrete types to all abstract types. This is done by specifying both the representation and the coding function for each mapping. At this stage, the extra parameters of the coding functions are left unspecified. We may have more than one concrete type assigned to the same abstract type. This is permitted because when we are programming using a high-level language we sometimes work on the same type abstractions even when we intend to use them in different contexts. Most mappings are parametric and, therefore, can be packed into reusable modules. The compiler uses all these mappings to express the implementation F of an abstract operation f by using equation 4. In Section 5 we will explain how values for the extra parameters are chosen by the compiler during the course of the translation, such that the produced translation is efficient. The compiler delays a commitment to a decision for these extra values as long as possible, to reduce the possibility of the decisions being withdrawn, forcing backtracking.

Consider for example the union of sets defined as:

```
function(alpha) union ( x: set(alpha), y: set(alpha) ) : set(alpha);
set_reduce(x,insert,[z]->z,y);
```

Suppose that we implement `set(alpha)` as `coded_set(beta)` using a coding function associated with the representation function `r`. Then `UNION`, some concrete implementation of `union`, has the type signature:

```
function(beta) UNION ( X: coded_set(beta), Y: coded_set(beta) ) : coded_set(beta);
```

and satisfies the following instantiation of equation 6:

```
r(UNION(X,Y),ra) = union(r(X,ra),r(Y,ra))
```

where `r` has been substituted for r_0, r_1 and r_2 since they are all the same and `ra` is the representation function for mapping `beta` into `alpha`. For example, if we implement `set(alpha)` as `sequence(alpha)`, then `ra` is `[z]->z` since `beta` is equal to `alpha`. In this case `UNION` can be derived from equation 7, after replacing c_0 with `cod` and r_1 and r_2 with `rep`, where `cod` and `rep` were introduced earlier as the coding and representation function for implementing sets as sequences:

```
UNION(X,Y) = cod(union(rep(X,[z]->z),rep(Y,[z]->z)),[z]->z,before)
```

We may set the extra parameter `before` either to `X.before`, or to `Y.before`, or to any other partial order between objects of type `alpha`. If `before=Y.before` then `UNION(X,Y)` can be simplified into

```
seq_reduce(X,seq_insert,[z]->z,Y)
```

after unfolding the function calls, making some program transformations, and folding back to `seq_reduce`. This simplification reflects the fact that since the `seq_reduce` starts with `Y` as the base from which the union is built and `Y` is ordered by `Y.before`, it is simpler to use the `Y` order as the final order. If `X.before` is chosen the base `Y` has to be reordered using `X.before` prior to the reduction.

As can be seen from above, choosing an ordering function is important because this choice may affect the quality of the translation. Consider, for example the expression `union(x,y)=z`, where `x`, `y`, and `z` are sets and `union` is implemented as `UNION`, given before. Then, if we set the value `before` in `UNION` equal to `z.before`, the equality test can be done in time proportional to the cardinality of `z` (because both sequences `UNION(x,y)` and `z` are sorted by the same function), instead of the square of this number (for checking whether the two lists in the sequences are equal).

Another example, which is a high order function, is the generic join function (it is generic enough to have selection and projection embedded in its input functions):

```
function(alpha,beta,gamma)
    join ( x: set(alpha), y: set(beta),
           match: [alpha,beta]->boolean,
           concat: [alpha,beta]->gamma ) : set(gamma);
set_reduce(x,union,
           [ex]->set_reduce(y,union,
                            [ey]->if match(ex,ey)
                                  then insert(concat(ex,ey),emptyset)
                                  else emptyset,
                            emptyset),
           emptyset);
```

`join(x,y,match,concat)` takes every combination of elements 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 COINS department:

```
join(employees,departments,
     [emp,dept]->(emp.dno=dept.dno && dept.name="COINS"),
     [emp,dept]->emp)
```

The implementation `JOIN` of `join` must satisfy the following equations:

```
ro(JOIN(X,Y,MATCH,CONCAT),rc) = join(rx(X,ra),ry(Y,rb),match,concat)
MATCH(X,Y) = match(ra(X),rb(Y))
rc(CONCAT(X,Y)) = concat(ra(X),rb(Y))
```

where `MATCH` and `CONCAT` are the implementations of `match` and `concat`, and `ra`, `rb`, and `rc` are the representation functions for `alpha`, `beta`, and `gamma`. For the implementation of `set(alpha)` as `sequence(alpha)`, `JOIN(X,Y,MATCH,CONCAT)` is:

```
cod(join(rep(X,[z]->z),rep(Y,[z]->z),MATCH,CONCAT),[z]->z,before)
```

5 Query Translation and Optimization

The type transformation process described in Section 4.1 maps abstract operations into their concrete counterparts. Equation 4 can be used to compute F by translating the input concrete objects into abstract objects, performing the abstract operation f upon them, and translating back the resulting abstract object into a concrete one. The objective of the translation is to express the concrete implementation in terms of the lower-level primitives exclusively, avoiding the object translation overhead. These primitives are part of the concrete layer and include objects such as B^+trees and the operations upon them.

Equation 3 gives the theorem for proving that a concrete program is a valid implementation of an abstract function. A theorem prover can prove this theorem for particular abstract and concrete operations if it is provided with a theory specifying the abstract layer, a theory specifying the concrete layer, and a theory produced by the transformation specification. The theory that describes a database specification in ADABTPL is derived by the schema definition and the operation specification [16]. The theory of the concrete layer could be expressed in the same way: by defining the types of the storage objects along with the primitive operation definitions. An alternative way is to use axioms for explicitly stating the semantics of operations.

The concrete layer consists of a number of concrete types. One example of a concrete type is the `sequence`. Each concrete type models the behavior of a storage object. For example, `sequence` could be a model of a B^+tree where the `before` function of the sequence serves as the comparison function for B^+tree keys. Each concrete type is associated with a number of concrete operations. Each concrete operation has a modeled behavior, an implementation, and a cost value. The modeled behavior of a concrete operation is its specification in terms of the concrete type primitives (constructors and selectors) or of calls to other concrete operations. The implementation is the actual primitive operation upon the storage objects which can be written in some formally intractable language such as C. The cost value is a real number, computed by applying a cost function to the concrete objects during compile time. These cost functions typically retrieve statistical information, such as the sequence sizes.

Effective operation optimization requires comparing alternative ways of execution. There are five main sources of these alternatives:

5.1 Non-isomorphic Mappings

Non-isomorphic mappings offer a number of different implementations. Here storage structures assigned to abstract objects are not fully specified by the internal schema writer, leaving options in the form of extra parameters in the coding functions to be chosen by the compiler. Assigning different values for the extra parameters in equation 4 chooses different implementations for an abstract function with possibly different execution costs. Non-isomorphisms are preferable to isomorphisms because they offer a variety of different function implementations to choose from.

5.2 Equivalences Between Concrete Operations

While the same computation in the concrete layer can be performed by different concrete operations, the cost of execution may not be the same. That is, some computations over the concrete types can be done redundantly by more than one concrete operation. For example, the membership function for sequences `seq_member` can be modeled as:

```
function(alpha) seq_member ( e: alpha, p: sequence(alpha) ) : boolean;
seq_reduce(p,or,[x]->(p.before(x,e) && p.before(e,x)),false)
```

This is the model of the concrete operation `seq_member`, expressing it in terms of other concrete operations, such as `seq_reduce`. This provides us with the information of how `seq_member` is related with the other concrete operations, not how `seq_member` is implemented. Operations `seq_member` and `seq_reduce` may have different implementations: `seq_member` could be implemented as an access to the B^+tree which is very fast, while `seq_reduce` could be done by a sequential scan which, in the worst case, needs to check all elements of the sequence.

5.3 Equivalences Between Abstract Operations

Two abstract operations can be proved to be equivalent, such as a select of a join and a join with an embedded select, even though an implementation of one may be more efficient than the other.

5.4 Dependencies Between Concrete Objects

We may have materialized views of concrete objects, containing redundant information. This offers alternative ways of accessing the same data. Index accesses are examples of this: some queries can be implemented by accessing only indexes.

One example is mapping `set(alpha)` into `seq_with_size(alpha)`:

```
seq_with_size(alpha) =
   struct make_sws ( seq: sequence(alpha), size: integer )
          where(x) x.size=seq_reduce(x.seq,plus,[z]->1,0);
```

The representation function is:

```
function(alpha,beta) sws_rep ( y: seq_with_size(beta) ) : set(alpha);
seq_reduce(y.seq,insert,[z]->z,emptyset)
```

The set cardinality `length(x)` is defined as `set_reduce(x,plus,[z]->1,0)`. The input type set is mapped into the type `seq_with_size`, while the output type integer is mapped into itself via the identity function. Therefore, the implementation of `length` is:

```
LENGTH(X) = set_reduce(sws_rep(X),plus,[z]->1,0)
          = set_reduce(seq_reduce(X.seq,insert,[z]->z,emptyset),plus,[z]->1,0)
          = ...      (after some program transformations)
          = seq_reduce(X.seq,plus,[z]->1,0)
          = X.size
```

5.5 Dependencies Between Abstract Objects

Integrity constraints on abstract types provide equivalences between abstract operations on these types. For example,

```
employees = set(employee) where(x) key(x,dno);
```

says that `employees` is a set of employees, where attribute `dno` of `employee` is the primary key of the set. That is, no two employees in the set have the same `dno`. This information could be used by the optimizer to perform faster retrievals using an index with key `dno`.

5.6 The Translation Process

In summary, the operation translation process is a search over alternatives generated by the five sources described earlier. This search is guided by cost functions that estimate the cost of expressions. The search algorithm could be implemented as a hill-climbing process, where only the best alternatives are considered and expanded each time for the later stages. The core of this process is a program transformation system that uses rewrite rules to derive equivalent forms for an expression. A very difficult problem here is to derive rewrite rules from the dependencies in the abstract and concrete layer. These are expressed as integrity constraints attached to a type definition, indicating how parts of this type are related. If these predicates are conjunctions of equalities, then we can derive the rewrite rules directly. Otherwise these constraints

need to be transformed into such forms, whenever possible. This is an incomplete process that may produce unsound rules. For that reason, a theorem prover must be used to check whether the produced rules introduce contradictions with the existing rules. The theorem prover can also be used to check whether the produced concrete operations satisfy equation 3.

6 Cost Estimation

In the previous sections we often stated that a particular expression was more efficient than another, but we did not give any criterion for this judgment. We could prove that the cost of an access path is smaller than the cost of another, whenever the cost parameters, such as the set cardinalities, become large. But if we have operations that depend on more than one parameter, then increasing the value of one parameter and decreasing another may result in a different optimal translation. One such example is the join operation whose optimal translation depends on the data stored in the join tables. This is a well known fact in database systems: the cost of query processing depends not only on the database schema itself, but also on the actual data stored in the database. Therefore, it is not enough to know how the input/output types of a function are mapped to their concrete implementations, but it is also necessary to know about the actual data passed as parameters. This problem can be solved by making the compiler perform a kind of data flow analysis where all actual calls to a function are considered to help the compiler deduce the parts of the database passed as parameters.

In this section we propose a cost model well-suited for the transformation framework, but good for other translation frameworks as well. This model can be applied to any system with recursively defined hierarchical structures or other bulk types, such as sets. The cost estimation here is a global process that considers every function call in the program. It has two parts. The first part is finding the **path signatures** for each function definition, that is, what parts of the concrete database object are passed to functions as parameters. This is done by performing a non-standard abstract interpretation of functions, described below in detail. For each path signature of a function there is an implementation program possibly different from the others, as it depends on a different distribution of data. The second part is estimating the cost of each implementation, in terms of the costs of the parts of the database passed as parameters and in terms of the concrete operation this function calls.

More specifically, we use a special graph, called the **access path graph**, as a comprehensive image of the actual data stored in the database and of the operations upon them. This graph contains all the possible valid access paths to a database. We construct this graph by initially laying out the database hierarchical schema along with the selectors. For example, consider the type `set(struct ms(a:list(integer),b:integer))`, that is, a set of tuples having constructor `ms` and selectors `a` and `b`. The `a` component of the tuple has type `list(integer)`, while the `b` component has type `integer`. The access path graph of a database of this type, containing only the selector functions, is shown in figure 2.

Thick arrows in figure 2 represent selectors, while thin arrows represent choices in a union. For example, node n_5 has two choices: nil (node n_8) and cons (node n_7). Non-unary functions, such as the type constructors, are represented as hyper-edges (multiple-input arrows). There is a special node, n_0, that represents all objects that are not part of the database, such as constants and temporary values. We can represent the graph with a set of rules, called the path signatures, indicating the relation between input and output nodes of a function. Some of these rules from the previous example are:

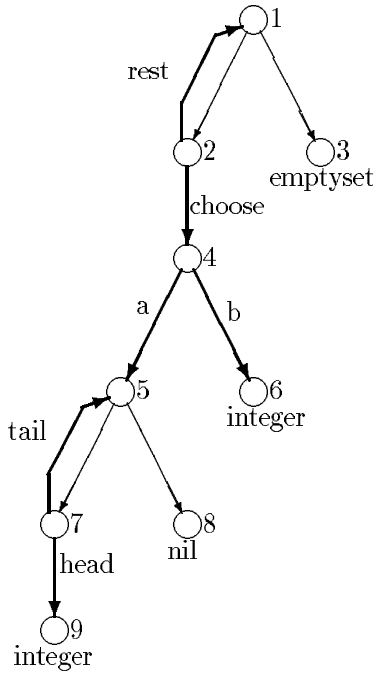
$$\begin{aligned} \text{rest}(n_2) &\rightarrow n_1 \\ \text{ms}(n_5, n_6) &\rightarrow n_4 \\ \text{cons}(x, n_5) &\rightarrow n_7 \\ \text{cons}(x, n_0) &\rightarrow n_0 \end{aligned}$$

where variable x can be any node of type integer, that is, n_9, n_6, or n_0.

Given any expression involving constructors, selectors, and nodes from the access path graph, we can traverse the graph starting from these nodes and reach a graph node. For example, the expression:

$$\text{cons}(10, \text{choose}(\text{rest}(\text{rest}(n_1))).a)$$

reaches node n_7. For each n-ary function f we can compute a set of path signatures, such that, given the graph nodes of the input and the body of f we can find the node that we reach by traversing the graph. This



Node Types

n_1: `set(struct mt (a:list(integer),b:integer))`
n_2: non empty set
n_3: empty set
n_4: `struct mt (a:list(integer),b:integer)`
n_5: `list(integer)`
n_6: integer
n_7: non empty list
n_8: nil list
n_9: integer

Figure 2: Example of an access path graph

is done by using the path signatures as rewrite rules to reduce the body of f to a graph node. A database transaction has a fixed signature: its functional translation accepts the database object as input (node n_1) along with some extra values (nodes n_0) and returns a new database object (node n_1) [16]. Therefore, the algorithm that computes the path signatures starts from the path signatures of transactions, constructors and selectors. Then it uses them as rules to reduce the body of transactions, adding new path signatures as rules along the way. This process is guaranteed to terminate, even when there are recursive calls, since there is a finite number of nodes in the graph and, therefore, there is a finite number of path signatures a function can take. For example, one path signature for `append` is append$(n_5, n_5) \rightarrow n_5$. This model can be extended to include high-order functions by allowing a node in a path signature to be a path signature itself. In this case, lambda abstractions also need to be assigned path signatures that can be computed by the same process.

In our model, the path access graph contains the representation of the concrete database type exclusively, along with the concrete operations and the function implementations. The compiler assigns a size to each graph node by performing some concrete operations, pre-specified by the internal schema writer. For example, the size of a sequence `s` could be derived from `seq_reduce(s,plus,[z]->1,0)`. If we have nested bulk types, like nested sequences, then the size of the innermost type is the average of the sizes of all the innermost objects.

In addition to the sizes assigned to nodes, each edge in the graph is assigned a cost value. The cost of a concrete operation is computed during compile time by the cost functions provided by the internal schema writer. Each path signature of a concrete operation may have a different cost, as it refers to a different portion of the database. The cost of any other operation is equal to the sum of costs of all operations called from its body. More specifically, for each path signature we assign a cost value, computed from the cost of the path signatures of the operations called. If this operation is directly or indirectly recursive, then we estimate the cost as the product of the sum of the sizes assigned to the input nodes, times the sum of the costs of the non-recursive calls. This is a very rough estimation, as it assumes that the recursion is linear.

Our approach for cost estimation is not complete: to compute the cost of an abstract function, we need to translate this function into a concrete program first. That is, when we compare alternative solutions to find the one with the minimum cost, we need to expand all of them in terms of concrete function calls, in order to get valid costs. This is not plausible, because some abstract functions may have a large number of possible implementations. To narrow the search, we need heuristics to prune out the alternatives that seem

very bad from the beginning. Pruning alternatives in the abstract level is a very difficult task and is a topic of further research.

7 Schema Evolution and Data Restructuring

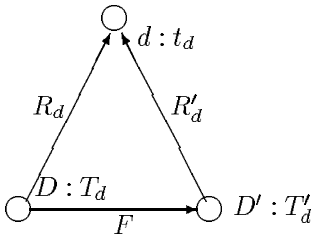
Most database application programs evolve during their lifetime to catch up with the users' needs and with current trends of technology. In our model these changes can be divided into two categories:

- Implementation evolution: changes to the implementation of the abstract model.
- Schema evolution: changes to the abstract model, including the abstract types and operations.

In either case, the actual data stored in the database need to be modified to reflect the new schema. Our transformation framework can be directly applied to this situation, as it is an adequate model for mapping types to types.

When we change the implementation of the database, it will be very convenient to have a function that gets the old concrete database as input and generates the new database of the new implementation. That way the new database does not have to be rebuilt from scratch. This is the problem of data restructuring in a database environment. The compiler can synthesize such a function using the type transformation model. The benefit of this is that these functions can be translated and optimized like any other operation.

Suppose that the abstract database object d of type t_d is assigned a concrete implementation D of type T_d via the representation function R_d. If we change the implementation of the database, we need to specify a new representation function R'_d that maps the database into its new implementation. We can specify R'_d in terms of R_d by changing the representation function of a few abstract objects that are part of the database, keeping the rest the same. That way R'_d will be the same as R_d, except for these modified parts.

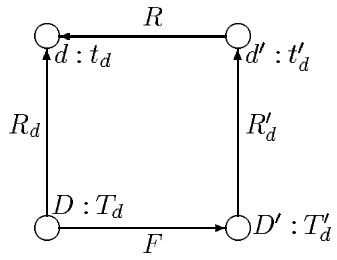


Function F is the function that rebuilds the database:

$$R'_d(F(D)) = R_d(D) \;\Rightarrow\; F(D) = C'_d(R_d(D), \ldots)$$

where C'_d is a coding function associated with R'_d, possibly needing some extra parameters.

If we change the database abstract schema t_d to t'_d via the representation function R, then the new abstract database object $d' : t'_d$ will have a different concrete implementation $D' : T'_d$ via the new representation function R'_d. Modifying the abstract database schema t_d into a new schema t'_d is not an abstract to concrete mapping but it can be made to fit in the same framework. This mapping is usually expressed using the coding function C instead of the representation function R. Note also that mapping an abstract database schema usually means making a small change to one of the types that constitute the database schema, such as adding a new component to a tuple. In that case, we need to define the mapping for this type only, leaving the rest as identity mappings.



Function F, the function that rebuilds the database, satisfies:

$$R(R'_d(F(D))) = R_d(D) \;\Rightarrow\; F(D) = C'_d(C(R_d(D), \ldots), \ldots)$$

where C'_d and C are coding functions associated with R'_d and R.

All changes to the database schema and assignments of storage structures to parts of the database need to be done incrementally. It would be very convenient to have a language to express only one piece of information at a time, such as adding a new component to a tuple, and let the compiler use all these changes to derive the necessary representation function for implementing the database type. For example, we could be provided with a macro `EXTEND` that extends a tuple with a new component:

```
EXTEND(struct ms ( a: integer, b: integer )
       WITH c: integer
       WHERE(x) x.c=x.a+x.b )
```

This implements the type `struct ms ( a: integer, b: integer )` as:

```
struct ms ( a: integer, b: integer, c: integer )
       where(x) x.c=x.a+x.b
```

using the representation function `[y]->ms(y.a,y.b)`. The compiler could use all these generated representation functions to synthesize the representation function that implements the database.

8 Summary

Data independence in database management systems is the separation of conceptual data from its concrete implementations. Its aim is to simplify database system design by separating logical data design from implementation details. This separation supports different optimizations of database programming. Among the optimizations supported by data independence are query execution optimization, data restructuring to tune performance without changing programs, and minimizing the reprogramming required when systems evolve.

Here we have described an approach to bringing data independence to database or persistent programming wherein database processing is specified in a strongly typed language in the same way that non-persistent data is manipulated. We argued that the usual way of achieving data independence in programming languages, via layered abstraction, is too rigid to meet the needs of data-intensive application development. Our approach is to adapt transformational techniques developed by Darlington and others to the database programming environment.

The salient features of this proposal are

- the extension of the type transformation model to include parametric high-order functions and the extension of the definition of the coding function to cover a wider class of homomorphisms;
- the development of a type transformation model that is used in a database context;
- the integration of the type transformation model with query optimization methods, including the use of semantics in the optimization process;
- bringing the data restructuring problem into the type transformation framework;

- the ability to validate type transformations using a mechanical theorem prover .

The current state of this work is

- A program transformation system that serves also as the core for the Boyer-Moore theorem prover has been implemented;
- A concrete layer has been implemented and specified formally.

Our current plans include

- the use of reflection to achieve flexibility and control over the translation process (schema modification macros);
- the use of restricted patterns of recursion, in the form of generic reductions, to perform plausible theorem proving and program synthesis;
- applying this theory to a relational environment, with optimization techniques found in a real database system;
- formulating a framework for the interaction between the internal schema writer and the compiler.

9 Acknowledgments

We are grateful for the help given by Graham Kirby, Ron Morrison, and Tim Sheard.

References

[1] M. P. Atkinson and O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[2] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1729, November 1988.

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. Wise. Genesis: A Reconfigurable Database Management System. Technical report, Department of Computer Science, University of Texas at Austin, March 1986. TR-86-07.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[5] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[6] J. Darlington. The Synthesis of Implementations for Abstract Data Types. In *Computer Program Synthesis Methodologies*, pages 309–334. D. Reidel Publishing Company, 1983.

[7] L. Fegaras, T. Sheard, and D. Stemple. The ADABTPL Type System. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 243–254, 1989.

[8] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 160–171, 1987.

[9] D. Gries and D. Volpano. The Transform - a New Language Construct. *Structured Programming*, 11:1–10, 1990.

[10] C. A. Hoare. Recursive Data Structures. *Journal of the ACM*, 4(2):105–132, June 1975.

[11] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[12] N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. *Proceedings of the Tenth ACM Symposium on Principles of Database Systems, Denver, Colorado*, pages 37–52, May 1991.

[13] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[14] D. Kapur and M. Srivas. A Rewrite Rule Based Approach for Synthesizing Abstract Data Types. In *Mathematical Foundations of Software Development*, volume 1, pages 188–207. Springer-Verlag, March 1985.

[15] 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*, pages 23–34, 1979.

[16] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 12(3), September 1989.

[17] D. Stemple, S. Mazumdar, and T. Sheard. On the Modes and Meaning of Feedback to Transaction Designers. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 374–386, 1987.