A Functional DBPL Revealing High Level Optimizations

Martin Erwig
Praktische Informatik IV
FernUniversität Hagen
Postfach 940, 5800 Hagen, Germany
erwig@fernuni-hagen.de

Udo W. Lipeck
Institut für Informatik
Universität Hannover
Lange Laube 22, 3000 Hannover 1, Germany
ul@informatik.uni-hannover.de

Abstract

We present a functional DBPL in the style of FP that facilitates the definition of precise semantics and opens up opportunities for far-reaching optimizations. The language is integrated into a functional data model, which is extended by arbitrary type hierarchies and complex objects. Thus we are able to provide the clarity of FP-like programs together with the full power of semantic data modelling. To give an impression of the special facilities for optimizing functional database languages, we point out some laws not presented before which enable access path selection already on the algebraic level of optimization. The algebraic way of access path optimization also gives new insights into optimization strategies.

1 Introduction

The design of new database programming languages is still a challenging task, for on the one hand it has to meet the conceptual requirements of programming languages and database systems at the same time, and on the other hand it has to present a suitable integration of both. Concerning database aspects, one observes that the drawbacks of the relational data model have led to a variety of so-called "semantic data models" [HK87] that more or less provide explicit methods for modelling classification, aggregation (tupling), association (grouping), and generalization. Unfortunately, some models grow very large (and thus become unwieldy) in trying to cover all shades of each concept by permanently introducing new modelling constructs; this does not come up to a good language design as described in [ABC^{2}M84]. Hence such models are not easy to use, and, not surprisingly, things will get even worse if one tries to combine such a model with a programming language. The functional data model [Shi81, BF79], however, with its few concepts is both easy to understand and to apply. Moreover, as we will show, it is nevertheless able to cover the above mentioned modelling concepts. Furthermore, a functional programming language can seamlessly be integrated into a functional data model [Nik88].

Concerning programming methodology, we believe that applicative languages are superior to imperative ones for a number of reasons [Sad87, Bac78]. Particularly, functional programs are shorter and much simpler, that is, easier to understand, than their imperative equivalents. Furthermore, because of referential transparency their semantics have simple mathematical descriptions. Special advantages of FP-like languages are [Bac78, Bac85]: (i) a hierarchic program structure which simplifies understanding of programs and enhances reusability, (ii) reasoning on function level (instead of object level), and (iii) an algebra of programs. Items (ii) and (iii) support a fine approach to program verification and optimization [KS81, Bac85, BK90], which lets FP be an excellent candidate for a functional DBPL, since optimization of queries to large databases is of particular interest [JK84].

The model we shall present in the following combines the modelling capabilities of semantic data models including complex objects and type hierarchies with the rigorous FP programming-style. Thus on the one hand, we significantly extend the models presented in [Shi81, BF79], and on the other hand, we obtain powerful optimization facilities that are not possible in models such as [PK90, ACo85]. Note that algebraic optimizations are not investigated in the work of [Shi81, BF79, BFN82] either.

Traditionally, the process of optimization is divided into two consecutive steps: algebraic optimization and access path optimization, which is not performed algebraically. In this paper we shall show that the functional model allows the latter kind of optimization to be done algebraically, too, thus bringing both steps onto a common (high) level. It is this uniform treatment that opens up the way to the investigation of new optimization strategies, for example, the interweaving of both optimization

stages. In fact, we will see that it is advisable to try access path optimization *before* the "algebraic" one.

The rest of the paper is structured as follows: In the next section we introduce a functional DBPL allowing for update and schema operations. Along with this we explain elements of the underlying extended functional data model. The definition of the semantics is outlined in Section 3. This enables us to describe an approach to algebraic optimizations in Section 4. There we first consider the "classical" algebraic optimization and the algebraic access path optimization separately (Subsections 4.1 and 4.2). Then we outline a possible way to join these two approaches in Subsection 4.3. Conclusions drawn in Section 5 will complete this paper.

2 ADAPLAN: A DBPL Based on an Extended Functional Data Model

The functional data model [Shi81] views a database as a set of *object and data types*, a type being a sort, that is, a set of data values or objects, together with some functions defined on that sort. Unlike data sorts (like NUM or BooL), object sorts change in time because their objects have to be created and may be deleted at some time. Likewise, data functions (such as + or $\wedge$) always return the same result when applied to the same data whereas functions defined on object types may be altered (for example, salaries may rise). Object attributes are modelled by data-valued object functions, and relationships between objects are captured by functions having object sorts as ranges (which may be complex ones, see below). Object functions may be *defined extensionally* (then we call them *mappings*), or they are *derived* by means of expressions (then we call them simply *functions*).

This basic model is extended in two directions.[1] First, we introduce a means to arrange object types into arbitrary hierarchies that can capture many different object class relationships. Inheritance of functions is not the only benefit gained from this, for one can also selectively apply different functions to a generalized object depending on its concrete subtype. Another reason for introducing generalizations is the opportunity to let explicit integrity constraints be captured by inherent ones. In defining new types one can express an inclusion or equality constraint between the set of new types and a set of already existing types (refer to [HNSE87] for details, and note that multiple inheritance is not captured by this method). In addition, each base type, that is, any type specialized out of the top sort `OBJECT`, is required to be defined along with at least one key mapping because we want to ensure that each object can be identified uniquely at any time. In [Bro84] this requirement is called *entity integrity*.

Second, the modelling of complex objects is supported by adding the type constructors *(set) union* $(A_1 \ldots A_n)$, *tuple* $[A_1 \ldots A_n]$, *set* $*A$, and *function* $(A \rightarrow B)$. Type constructors are used to describe the types of functionals (see below) and to allow the definition of complex range types of object functions. In the latter case function type constructors must not be used because "functions as ranges of functions" would complicate the application mechanism (though desirable in some cases [AH87] we feel that simplicity of the overall conception dominates). Since type constructors may be nested we achieve type-completeness [AB87, ABC²M84]. For a detailed description of types and constructors, see [oho90].

In the following we show how all the previous ideas can be turned into a concrete DBPL. Therefore, the different concepts of the **a**pplicative **da**tabase **p**rogramming **lan**guage ADAPLAN are presented by giving several examples. The language provides schema, update, and query operations and furthermore offers the ability to define explicit integrity constraints and transactions.

We begin with describing the schema for a parts database. A part is named and has a total price. It is either a base part or an assembly group. A base part has an associated price, and an assembly group has a parts list telling which parts are used how many times in its composition. Moreover, we assume that parts are delivered by suppliers located in a certain city. In addition, we keep track of orders each of which consists of a set of parts. Now, the definition of the object sorts looks like this:

```
TYPE Part ≤ OBJECT,     KEY name: Part |→ STRING
TYPE (Base Group) = Part
TYPE Supplier ≤ OBJECT, KEY name: Supplier |→ STRING
TYPE City ≤ OBJECT,     KEY name: City |→ STRING
TYPE Order ≤ OBJECT,    KEY who: Order |→ Supplier, no: Order |→ NUM
```

[1] The subtype relationships of DAPLEX capture only a small part of imaginable type structures, and being restricted to multivalued functions for grouping one is not able to model, for example, sets of sets.

Here Part, City, Supplier, and Order are each defined as a subtype of the top sort OBJECT. A bar to the left of an arrow restricts a mapping to being total (a bar to the right specifies surjectivity), so with the above definitions we require the key mappings being defined for each object of the respective sort. Key mappings are missing in the definition of Base and Group because these types are subtypes of Part and therefore both inherit the key mapping name. In fact, this subtype specification defines a partition, that is, each part has to be in exactly one of the respective subsorts. Now consider the definition of the mappings:

```
MAP delivered: Part      |→| Supplier
MAP located:   Supplier  |→  City
MAP transport: Supplier  |→  STRING
MAP shipped:   Order     |→  *Part
MAP baseprice: Base      |→  NUM
MAP partslist: Group      →  *[Part NUM]
```

Note that shipped and partslist yield complex objects, namely sets and sets of tuples, respectively. Since partslist maps group parts to parts we in fact have defined a recursive type.

Before defining the derived functions price and total some principles of the FP programming-style should be outlined: Expressions (called *functional forms*) are built by applying *functionals* to *functions*. Functions can be data type functions (such as +, ∪), object type functions (like located, name), data type constants (for instance, 'Rome', true, {3, 17}), object type constants (for example, Order, Supplier('Cheap')), or some special functions such as

id	(identity function)
#i	(selects the ith component out of a tuple)
T?	(checks whether an object has type T)
split	(partitions a set into an element and the remainder of the set)

The following set of functionals has proved to be useful in DBPLs (functions are denoted in general by variables f, g, h, set-valued functions by s, t, predicates by p, q, constants by c, and mappings by m):

f ∘ g	(T → U) (S → T)	→ (S → U)	*composition*
*f	(S → T)	→ (*S → *T)	*map, apply-to-all*
f/g	([T T] → T) (S → T)	→ (*S → T)	*aggregate*
|p|	(S → BOOL)	→ (*S → *S)	*filter*
[f_1 … f_n]	(S → T_1) … (S → T_n)	→ (S → [T_1 … T_n])	*tuple*
{f_1 … f_n}	(S → T_1) … (S → T_n)	→ (S → *(T_1 … T_n))	*set construction*
~m	(S → T)	→ (T → *S)	*inverse*
p → f; g	(S → BOOL) (S → T) (S → U)	→ (S → (T U))	*conditional*

The semantics of the functionals and their use is illustrated by the following examples:

```
*located ∘ Supplier                                          (1)
max/baseprice ∘ Base                                         (2)
÷ ∘ [+/baseprice +/1] ∘ Base                                 (3)
|(located = City('Rome')) ∘ delivered| ∘ Part                (4)
∪/~delivered ∘ ~located ∘ City('Rome')                       (5)
```

Function composition can be imagined as pipelining the result of one function into the next and is needed in almost any expression. In (1) the function located is applied to each object in the set Supplier, so the result is a set of cities. In (2) the mapping baseprice is applied to each element of the set Base (temporarily preserving duplicates) and then the maximum value is computed by aggregating the results with the binary function max. Other examples for aggregation are: counting (summing up) a set of elements (numbers), which can be expressed by +/1 (+/id). The average price of base parts can be

determined as shown in (3) where a tuple, consisting of the sum of prices and the number of base parts, is constructed to which the division operator is applied. To enhance readability we often use infix notation (`x f y`) in place of the FP expression `f∘[x y]` as in (4) which asks for all parts delivered from Rome. The same query is expressed by (5) where first, all suppliers of Rome are determined (by the inverse `~located`) and second, all parts delivered by each of them are calculated (by the inverse `~delivered`) and aggregated using ∪. Set construction and partition are present mainly for technical reason; they will be defined formally in Section 3 where they are used in the definition of other functionals. An example for the use of the conditional is given by the definition of the derived function `price`[2]:

```
FUN price: Part → NUM =
    Base? → baseprice; +/(price ∘ #1 × #2) ∘ partslist
FUN total: Order → NUM = +/price ∘ shipped
```

The defining expression of `price` operates as follows: `Base?` checks whether the argument part is a base part. In this case the stored price is the result; otherwise for all elements of the parts list, the total price of the subpart is multiplied with its frequency and all values are summed up.[3] The expression defining `total` should be clear now. More query examples are presented in Section 4.

Now let us consider some update operations. We follow the idea of AST (*applicative state transition*) systems [Bac78] and consider each update as an atomic operation that transforms one (valid) database state into another.[4] Conceptually, an update happens all at once, that is, there is no undefined intermediate state to which queries may refer, and the update operation itself "sees" at any time nothing but the old, yet unchanged, database state. Moreover, an update is performed completely or not at all, that is, when an integrity constraint is violated by a part of the update, the database state does not change. Thus referential transparency is guaranteed for query and update expressions so that the applicative nature of our language is not touched by updates. These topics are discussed more thoroughly in [Nik88]; implementation issues are considered in [Tri89].

As already indicated, updates must not violate *integrity constraints* which include *inherent* and also *explicit* ones. Inherent integrity constraints are, for example, key integrity, referential integrity, invariants of mappings (like being total or surjective), or inclusion/equality constraints implied by type definitions. The latter are preserved by update operations, and thus due to the above type definition, for instance, inserting a new part object can easily be done by inserting it into any of its subtypes. Note that we have to supply the values for the key mappings.

```
NEW Base('Cylinder'):OBJECT
NEW Order(Supplier('Cheap'), 33):OBJECT
```

The type following the colon is the type the new object "comes from", so in this case two "completely new" objects are meant. In contrast, if we want to insert an already existing object into another type, for example, a part into a type `Selfmade` (not defined in the above schema), we write:

```
NEW Selfmade('Cylinder'):Base('Cylinder')
```

causing that the object is now element of three types. Next, consider the update of a mapping which is specified by a pair [*object key(s) mapping value*]:

```
LET baseprice: ['Cylinder' 100]
LET partslist: ['Engine' {[Part('Engine Block') 1] [Part('Cylinder') 4]}]
```

In the second update the constructors set and tuple were used to obtain complex objects. Update operations can cope with multiple values, too; then a set of update pairs has to be provided. If, for instance, the prices of all base parts delivered by supplier `Cheap` increase by 10%, the following update is per-

---

[2] This is similar to the cost and mass example of [AB87].

[3] Note that functionals have higher precedence than functions (conditional having lowest). Precedence can be changed using parentheses.

[4] In fact, the old database state is preserved so that the new state does not simply replace the old but rather extends it. In particular, this opens up the possibility for dynamic integrity constraints, but this is not crucial to the following discussion. For details refer to [Erw89].

formed:

```
*LET baseprice: *[name baseprice × 1.1] ∘ |delivered=Supplier('Cheap')| ∘ Base
```

For each part object selected by the filter expression a tuple consisting of its name and the raised price is constructed resulting in the required update set. The deletion of objects is performed by giving a single object or a set of objects, for example, deleting a specific group object respectively all parts of a certain supplier is done by:

```
DROP Group('Engine')
*DROP ~delivered ∘ Supplier('Cheap')
```

There are cases in which one update operation alone cannot achieve the desired change in the database, for example, if a cylinder is not viewed as a base part any more, how can we make it become an element of the `Group` subtype? Simply adding it to the subtype is not possible since that would contradict the partition constraint; dropping it first is not possible either due to the reference from the above defined parts list. Now, a transaction can be written within which the referential integrity may temporarily be violated.

```
BEGIN
  DROP Base('Cylinder')
  NEW  Group('Cylinder'):OBJECT
END
```

In addition to inherent integrity constraints it is possible to state *explicit integrity constraints*, too:

```
ASSERT notfree: and/(price > 0) ∘ Part
```

which checks for all parts whether the price is positive. Subsequent updates are accepted only if the condition evaluates to `true`. Notice that the specification of a new integrity constraint is accepted only if it holds in the current state.

3 Query Language Semantics

Due to limitation of space we will not give definitions for all language constructs; we will rather concentrate on defining part of the query language thus justifying optimization laws which follow in Section 4. A complete treatment is given in [Erw89] where formal semantics of update operations and schema operations are provided, too. There the formal basis of FP, called FFP [Bac78], is extended by the notion of mappings. This facilitates the definition of an AST-system in which database state and history, integrity constraints, and transactions can formally be defined. The resulting persistent FP programming system is then used to give an operational semantics to the DBPL ADAPLAN.

A *constant* (*constant function*) is an element of a data or object type or a name of an object type denoting the set of objects belonging to that type. The set of *primitive functions* consists of constants, data and object type functions, and some miscellaneous functions (`id`, `split`, ...). A mapping inherently means a set of pairs; the meaning of a mapping application can be compared with a table lookup: A pair is sought in the extension of the mapping for which the first component equals the object to which the mapping is applied. If such a pair exists its second component is obtained as the result of the application, if not, the undefined object ($\perp$) is returned. An *expression* denoting a query or defining a derived function is given by a *functional form*, which is an application of a functional (sometimes called "program forming operator" or "combining form") to functions or expressions.

Since the set of functionals is fixed the expressive power of the query language is inferior to that of the λ-calculus. We feel that this is not a severe limitation because one can easily identify a set of functionals that meets the requirements of a DBPL. Extensions of FP to enable full higher order programming are considered, for example, in [Bel85, SS86].

For convenience the semantics of the language is defined by function level equations which also may serve as (basic) laws for optimizations. We will give only those equations explaining the essence of some functionals which are of special interest in database languages; a more complete collection of equations, including *composition* $\circ$, *tuple* `[]`, *selection* #, and *conditional* `( → ; )`, can be found in

[Bac85].

Since we are dealing with sets rather than lists we have to define a corresponding construction functional and a selection function (like [] and # for tuples). Likewise, we have do adapt the definition of the map functional to operate on sets.

set construction {}: (S → T) (S → U) → (S → *(T U))

$$\{f\ g\} \circ h := \begin{cases} \{f \circ h\} & \text{if } f \circ h = g \circ h \\ \{f \circ h\ g \circ h\} & \text{otherwise} \end{cases}$$

The special case (for two functions) given here can easily be generalized to more arguments. Elements can be selected out of a set by means of set partition, which is realized by the function split:

partition split: *S → [S *S]

split ∘ {} := ⊥

split ∘ s := [f t] where {f} ∪ t = s and f ∉ t

The second line means that an element of s is selected non-deterministically.

Now the map functional is defined referring to set construction and partition.

map *: (S → T) → (*S → *T)

*f ∘ {} := {}

*f ∘ s := ∪ ∘ [f ∘ #1 *f ∘ #2] ∘ split ∘ s

From this definition the following equation can immediately be derived:

$$*f \circ \{g\ h\} = \begin{cases} \{f \circ h\} & \text{if } f \circ h = g \circ h \\ \{f \circ h\ g \circ h\} & \text{otherwise} \end{cases}$$

Aggregation enables the application of binary functions to sets of arguments.

aggregation /: ([T T] → T) (S → T) → (*S → T)

f/g ∘ {} := unit ∘ f

f/g ∘ {h} := g ∘ h

f/g ∘ s := f ∘ [g ∘ #1 f/g ∘ #2] ∘ split ∘ s

Note that aggregation is defined properly only for functions f being associative and commutative. This aggregation yields deterministic functions since the order in which elements are taken from s is irrelevant to the semantics. The aggregation of an empty set is defined exactly if unit∘f is defined. (unit∘f yields the unit element for the operation f, for example, unit ∘ + = 0 and unit ∘ ∪ = {}.) We shall assume that unit is predefined by the system and that a user may update/extend it if necessary.

The filter functional takes a predicate and constructs a function which restricts a set to those elements for which the predicate holds.

filter ||: (S → BOOL) → (*S → *S)

|p| ∘ s := ∪/(p → {id}; {}) ∘ s

The inverse of a mapping m is a function selecting all objects of its domain that are mapped to the value to which the inverse is applied. (dom yields the domain of a mapping.)

inverse ~: (S → T) → (T → *S)

~f ∘ c := |f = c| ∘ dom ∘ f

~f ∘ s := |f ∈ s| ∘ dom ∘ f

Note that in the case of set-arguments (second line) the definition of inverse differs from the mathematical one. This was done because in the context of database queries the interpretation presented here is

needed more often. Consider, for example, a query which asks for all orders that contain expensive parts. With the above definition, the scan

```
||price > 1000| ∘ shipped ≠ {}| ∘ Order
```

can be replaced by the more intuitive inverse construction

```
~shipped ∘ |price > 1000| ∘ Part
```

which directly supports the use of indexes (see below). Note that if a multi-valued function is viewed as a relation, the inverse of such a function just corresponds to the inverse relation.

Bear in mind that, according to the FP-style, we are *manipulating functions* and not objects. Thus in general, the function variables used in equations stand for arbitrary functional forms which greatly extends the applicability of laws.

4 Optimization

The existence of an *algebra of programs* [Bac78, Bac85, Wil82] is founded on the combining forms being the only means for constructing programs and queries. This algebra can be used to reason about programs, and we will describe part of it by giving several equations containing combining forms and function variables. Apart from the laws defining the algebra of programs, there are derived rules (identities that can be inferred directly from the laws) and theorems (giving nonrecursive definitions for recursively defined functions, see [Wil82, KS81] for details). It has turned out that derived rules are suited best for our purpose.

The *completeness* of a set of rules can be judged from three different points of view: First, a complete set of rules in the sense that it explicitly contains all valid rules is impossible to give since each law has infinitely many instantiations. (This is due to the fact that a variable in an equation stands for an almost arbitrary program.) Second, a set of rules may be considered complete if it describes the intended semantics of the functionals. More specifically, such a complete set of rules may then serve as a rewrite system capable of reducing any functional form to an object. Such a treatment is given in [HW^{3}85]. Third, due to [Bac85]: "… the object of a complete description would be to give a practically useful set of equations, some of which might be redundant …". It is this view we adopt here, and we shall provide a powerful set of rules which goes beyond the rules of relational algebra, especially being concerned with employing access paths on the algebraic level of optimization. This set of rules is partially redundant when promising good optimizations and not complete in the second sense, that is, ignoring rules that cannot be used for optimization. The need for redundant rules can be seen from the following example: A general filter `|p|` can at best be "moved to the right", but a selection `|f = c|`, which is an instance of the former, can sometimes be transfomed directly into a table lookup (see rules (I3) and (I4) in Subsection 4.2). Some of the rules to be presented are touched partially in [BK90] where a general algebraic optimization framework is described.

one principal goal of query optimization in database systems is to minimize the number of I/o-operations making up the prevailing amount of processing time. This is traditionally achieved by first reducing the number and extent of intermediate results and after that employing access paths. Since in our approach both steps of optimization are performed on an algebraic level it seems to be an interesting question whether an interweaving scheme of *plain optimization*[5] with index optimization, or, access path optimization, will be advantageous over the traditional two-step approach. As it turns out, this indeed seems to be the case.

So in a first attempt we will present here an optimization scheme that considers both steps separately: For each step we first present a set of rules, then we outline a possible optimization strategy, and finally we give several examples of query optimization. Where possible, we compare our approach with that of relational algebra. This is done in Subsections 4.1 and 4.2, respectively. Finally, we indicate in Subsection 4.3 how both steps could be joined together.

---

[5] From now on we will call the "classical algebraic optimization" *plain optimization* to distinguish it from index optimization which is also performed algebraically within our framework.

4.1 Reducing Intermediate Results

We start by giving laws supporting plain optimization. Note that in this place we have to consider only those functionals yielding expressions that operate on sets, namely, *filter*, *map*, and *aggregate*.

A well-known heuristic is to perform selections as early as possible, which can be achieved in our language by applying laws to move selections to the right.

$$|p| \circ *f \quad \rightrightarrows \quad *f \circ |p \circ f| \qquad \text{(F1)}$$
$$|p| \circ \cup/id \quad \equiv \quad \cup/id \circ *|p| \qquad \text{(F2)}$$

Rule (F2) can be viewed as a generalization of the law in relational algebra for commuting selection with union.

After being moved right, cascades of selections should be grouped together. We present only two rules; the reader may easily discover others.

$$|p| \circ |q| \quad \equiv \quad |p \wedge q| \qquad \text{(F3)}$$
$$(|p| \cup |q|) \quad \equiv \quad |p \vee q| \qquad \text{(F4)}$$

Finally, the expressions inside the filter might be simplified like, for example, $p \wedge p \rightrightarrows p$.

Since functional joins of the relational model can be expressed as mapped function applications (for example, $\pi_{City.name}(Part \bowtie Supplier \bowtie City) \equiv *name \circ *located \circ *delivered \circ Part$) it is possible to reduce the number of intermediate results of such join cascades, which is not possible in the relational model. Similarly, we can reduce a map followed by a filter to a conditional aggregate (that rule is closely related to (A3), see below).

$$*f \circ *g \quad \rightrightarrows \quad *(f \circ g) \qquad \text{(M1)}$$
$$*f \circ |p| \quad \equiv \quad \cup/(p \rightarrow \{f\}; \{\}) \qquad \text{(M2)}$$

Powerful rules exist for the optimization of aggregates that are totally missing in the relational model. Provided that the binary function of an aggregate is idempotent and, in the case of filter, the unit for f is defined, a cascade of aggregates can be eliminated (A1), likewise a map (A2) or filter (A3) preceding an aggregate. Moreover, a number of "parallel" aggregates can be turned into one (A4), which means that just one iteration over the same set suffices to produce a tuple of aggregate values.

$$f/g \circ \cup/id \quad \rightrightarrows \quad f/(f/g) \qquad \text{(A1)}$$
$$f/g \circ *h \quad \equiv \quad f/(g \circ h) \qquad \text{(A2)}$$
$$f/g \circ |p| \quad \equiv \quad f/(p \rightarrow g; \; unit \circ f) \qquad \text{(A3)}$$
$$[f_1/g_1 \; \ldots \; f_n/g_n] \quad \equiv \quad [f_1 \circ [\#1\_1 \; \#1\_2] \; \ldots \; f_n \circ [\#n\_1 \; \#n\_2]]/[g_1 \; \ldots \; g_n]^6 \qquad \text{(A4)}$$

To see the correctness of (A2) it is helpful to have the following theorem:

Theorem. *For all functions* $f\colon [T\ T] \rightarrow T$, $g\colon S \rightarrow T$, *and* $s\colon \rightarrow *S$ *with*

(i) f *being associative and commutative*

(ii) $\forall x \in S\colon f \circ [x\ x] = x$

the following equation holds:

$$f/g \circ s = f/id \circ *g \circ s$$

The proof is given in the Appendix. Now the validity of (A2) can be shown as follows:

$$\begin{array}{ll} & f/g \circ *h \\ \rightrightarrows & f/id \circ *g \circ *h \qquad \text{(Theorem)} \\ \rightrightarrows & f/id \circ *(g \circ h) \qquad \text{(M1)} \\ \equiv & f/(g \circ h) \qquad \text{(Theorem)} \end{array}$$

An optimization strategy employing the above rules might consist of the following steps:

---

[6] $\#i\_j$ is simply an abbreviation for $\#i \circ \#j$.

Reduce Intermediate Results

(1) Expand function definitions.
(2) Move filters to the right.
    Group filters.
(3) Group maps.
(4) Group aggregates.

The steps (2) - (3) may be followed by an additional optimization of subexpression. Note that the above scheme is amenable to enhancements, for example, it is possible that after step (3) rules of step (2) may apply again so that a loop back might be introduced.

Let us consider some examples. If we want to check transportation costs we might look at cheap parts and find out the cities of those suppliers that ship their parts by airplane. The query would be transformed as follows:

```
    *located ∘ |transport='air'| ∘ *delivered ∘ |price<10| ∘ Part
⇉   *located ∘ *delivered ∘ |(transport='air') ∘ delivered| ∘ |price<10| ∘ Part    (F1)
⇉   *located ∘ *delivered ∘ |(transport='air') ∘ delivered ∧ (price<10)| ∘ Part   (F3)
⇉   *(located ∘ delivered) ∘ |(transport='air') ∘ delivered ∧ (price<10)| ∘ Part  (M1)
⇉   ∪/((transport='air') ∘ delivered ∧ (price<10) →
        {located ∘ delivered}; {}) ∘ Part                                        (M2)
```

Here we were able to eliminate three intermediate sets. Another example is the query for all currently ordered parts that are delivered by supplier `Cheap`:

```
    |delivered = Supplier('Cheap')| ∘ ∪/id ∘ *shipped ∘ Order
⇉   ∪/id ∘ *|delivered = Supplier('Cheap')| ∘ *shipped ∘ Order                    (F2)
⇉   ∪/id ∘ *(|delivered = Supplier('Cheap')| ∘ shipped) ∘ Order                   (M1)
⇉   ∪/(|delivered = Supplier('Cheap')| ∘ shipped) ∘ Order                         (A2)
```

Query (3) of Section 2 can be optimized using rule (A4):

```
    ÷ ∘ [+/baseprice +/1] ∘ Base
⇉   ÷ ∘ [+ ∘ [#1_1 #1_2] + ∘ [#2_1 #2_2]]/[baseprice 1] ∘ Base                    (A4)
```

The resulting query scans the type `Base` is only once, building tuples of aggregate values immediately, whereas the original query needs two scans to build the tuple of aggregates.

Note that optimization may also apply to updates, for instance, the price-raising update of Section 2 can be transformed as follows:

```
    *LET baseprice: *[name baseprice × 1.1] ∘ |delivered= Supplier('Cheap')| ∘ Base
⇉   *LET baseprice: ∪/(delivered= Supplier('Cheap') →
        {[name baseprice × 1.1]}; {}) ∘ Base                                     (M2)
```

Finally, let us complete the comparison with relational algebra. Since the functional model differs significantly from the relational model a comparison on the whole is not possible. This was already indicated in the discussion preceding rule (M1) and has its climax in the fact that a cartesian product operator is missing in the functional model. Nevertheless, if we would add a cartesian product operator `cp: [*S *T] → *[S T]` to our algebra, laws corresponding to relational algebra could, of course, be formulated. Consider, for example, commuting selection with cartesian product (cf. [Ull89], p. 665):

```
|p ∘ #1| ∘ cp ∘ [s t]          ⇉   cp ∘ [|p| ∘ s t]
|p ∘ #1 ∧ q ∘ #2| ∘ cp ∘ [s t] ⇉   cp ∘ [|p| ∘ s |q| ∘ t]
|p ∘ #1 ∧ q| ∘ cp ∘ [s t]      ⇉   |q| ∘ cp ∘ [|p| ∘ s t]
```

Since function application and tuple selection correspond to projection we also can give laws for removing cascades of projection or for commuting projections with other operators. Such rules can easily be defined, for example:

```
[#1 … #n] ∘ [f₁ … fₘ]    ⇉   [f₁ … fₙ]              (if n ≤ m, cascade of projection)
*f ∘ (s ∪ t)             ⇉   *f ∘ s ∪ *f ∘ t        (commuting projection with union)
```

Readers specifically interested in these laws may refer to [BK90] where a framework related more closely to relational algebra is considered.

4.2 Algebraic Access Path optimization

As already indicated we want to focus our discussion on performing access path optimizations already on the algebraic level. An approach to make this feasible in the relational model was outlined in [SS90] where the relational model was formally extended by the notion of tuple identifier.

We start by giving some general observations. First, we note that an index in the functional data model is nothing but a materialized inverse mapping. Second, the general objective of the use of indexes is to replace a scan of the domain of a mapping by a scan of the range of that mapping. Surely, the profit of any such transformation depends on the type of scan and on the structure of the domain and range. (This will be explained soon.) Third, since the information about domains and ranges of functions are recorded in the schema corresponding schema equations have to be added to the optimization rules. These are of the form:

```
dom ∘ f = S                                   (if f total)
rng ∘ g = T                                   (if g surjective)
```

for mappings/functions `f: S |→ T`, `g: S →| T`. For instance, in our schema holds:

```
dom ∘ located   = Supplier                    (S1)
dom ∘ baseprice = Base                        (S2)
dom ∘ delivered = Part                        (S3)
rng ∘ delivered = Supplier                    (S4)
dom ∘ shipped   = Order                       (S5)
```

To keep the subsequent discussion more compact we from now on identify a type name with the equivalence class of expressions induced by schema equations as those above. So in order to match a rule against an expression `f` possibly each expression of `f`'s equivalence class has to be considered.

Next we present a schema that captures all possible (as far as we have studied) index optimizations in the framework of the presented data model. The general rule may be formulated as:

$domscan_i \circ \mathtt{dom} \circ \mathtt{f} \quad \stackrel{\rightrightarrows}{\equiv} \quad rngscan_i \circ \mathtt{rng} \circ \mathtt{f}$ (I1) - (I5)

where $domscan_i$ and $rngscan_i$ can be one of the following forms:

i	$domscan_i$	$rngscan_i$	*comment*
1	`*f`	`id`	
2	`g/f`	`g/id`	(only if `g` is idempotent)
3	`|p∘f|`	`∪/~f∘|p|`	
4	`|f = c|`	`~f∘c`	
5	`id`	`∪/~f`	

Note that (I4) and (I5) are special cases of rule (I3) with `p = |f = c|` and `p = true`. In fact, the resulting expression of (I4) reduces to `~f∘c` since `c` denotes a constant.

Let us briefly consider the conditions under which the above rules may be applied together with the expected savings in execution time. The first rule is especially helpful if the range of a mapping is smaller than its domain and if it is stored separately from the domain or perhaps additionally. The latter condition is typically fulfilled if there is an index on `f`. Generally, the same preconditions also apply to the rules (I2) - (I4). Concerning the second rule, if the binary function is `max` or `min` and the index is stored as a B-Tree, this rule can replace a full scan of the range by a sublinear (logarithmic) one. Rule (I3) can be applied very often since the left side denotes the selection of objects due to a property (`p`) of a certain attribute (`f`). Besides the range-size, the gain of efficiency depends on the selectivity of `p`. In the extreme, only constantly many objects qualify, as it is often the case for rule (I4),

and so the full scan sometimes can be reduced to a simple table lookup; it is obvious that the case (I4) should be checked (and, of course, applied if possible) before (I3) is tried. Rule (I5) applies if a scan of a small subtype (being located very low in the hierarchy) is to be performed and the complete hierarchy is stored together, that is, not clustered horizontally [CDFLNR82]. Then scanning the whole hierarchy may require the loading of several blocks for each element of the subtype, and thus a lookup driven by the range seems much more appropriate.

We observe that in order to apply rules (I1) - (I4) a function at the right of the scanform has to match the `f` in the expression `dom`$\circ$`f`. Since `f` is sometimes involved in tuple expressions it is recommendable to move `f` out of tuples to the right so that the index rules can be applied. This can be done with the help of the following *tuple factorization* rule:

$$[E_1 \ \ldots \ E_n] \quad \supseteq \quad [f_1 \ \ldots \ f_n] \circ g \qquad \text{(T1)}$$
where $E_i = f_i \circ g$ or $E_i = f_i = c$

Likewise, it is sometimes necessary to undo previously performed optimizations to let an index scheme apply. Examples are the following *undo* rules:

$$*f \circ |p \circ f| \quad \supseteq \quad |p| \circ *f \qquad \text{(U1)}$$
$$*(f \circ g) \quad \supseteq \quad *f \circ *g \qquad \text{(U2)}$$
$$\cup/f \circ |p \circ f| \quad \supseteq \quad \cup/id \circ |p| \circ *f \qquad \text{(U3)}$$

Rule (U1) undoes (F1), (U2) undoes (M1), and (U3) is a combination of (U1) and (A2). This indicates that an interweaving scheme of plain and index optimization is highly useful because there are cases in which F-, M-, and A-rules have to be avoided if index optimization should apply. To see the need for U-rules consider the following diagram in which a situation for the application of (U1) is given (we assume $p \neq$ `(id = c)` so that (I4) cannot be applied):

$$*f \circ |p \circ f| \circ dom \circ f \xrightarrow{(I3)} *f \circ \cup/\sim f \circ |p| \circ rng \circ f$$
$$\downarrow (U1)$$
$$|p| \circ *f \circ dom \circ f \xrightarrow{(I1)} |p| \circ rng \circ f$$

It is immediately clear that the expression resulting from the transformation (U1, I1) is more efficient than the one resulting from (I3). Note carefully that a rule $*f \circ \cup/\sim f \supseteq$ `id` is true only for single-valued functions (refer to the definition of ~ in Section 3) so that the rule (U1) is needed in general to achieve the above optimization. Unlike the reader may perhaps suppose, this rule does not contradict the general heuristic to perform selections as soon as possible. As can be seen from the result it rather has to be understood as an intermediate step in a transformation that finally allows index access.

The strategy for index optimization may now be described as follows:[7]

Index optimization

(1) Identify the subset of mappings *M* (occurring in the query) on which an index is defined.
(2) Try to match an index rule with $f \in M$. If none matches, apply T- and U-rules and try again. If a matching index rule is found, apply it.
(3) Repeat step (2) until no more index rules are applicable.

Let us consider some examples. To illustrate the use of the equivalence classes we give explicit transformations referring to the corresponding schema equations when necessary. The queries (1) and (2) of Section 2 can be directly transformed as follows:

```
    *located ∘ Supplier
⊇   *located ∘ dom ∘ located        (S1)
⊇   rng ∘ located                   (I1)
```

[7] To be precise, we had to add conditions to the index rules (which are based on cost-estimates) so that substitutions are performed only when they are expected to result in more efficient queries. But the preceding discussion of the index rules has indicated that these considerations are very similar to those of the relational model. So we do not investigate that topic any further.

```
    max/baseprice ∘ Base
⇉   max/baseprice ∘ dom ∘ baseprice      (S2)
≅   max/id ∘ rng ∘ baseprice             (I2)
```

Note that the resulting expressions are more efficient not only when the range is smaller than the domain but also if the functions are not clustered with their domain, that is, stored as a separate table (see [CDFLNR82]). Next we show how query (4) of Section 2 can be optimized to query (5):

```
    |(located = City('Rome')) ∘ delivered| ∘ Part
⇉   |(located = City('Rome')) ∘ delivered| ∘ dom ∘ delivered                  (S3)
≅   ∪/~delivered ∘ |(located = City('Rome'))| ∘ rng ∘ delivered              (I3)
≅   ∪/~delivered ∘ |(located = City('Rome'))| ∘ dom ∘ located                (S4, S1)
≅   ∪/~delivered ∘ ~located ∘ City('Rome')                                  (I4)
```

Finally, we give an example showing the need for T- and U-rules in the process of index optimization. Suppose we want to know for each part, which is transported by airplane, the city from which it is delivered.

```
    *(located ∘ delivered) ∘ |transport ∘ delivered = 'air'| ∘ Part
≅   *(located ∘ delivered) ∘ |(transport = 'air') ∘ delivered| ∘ Part        (T1)
≅   *located ∘ *delivered ∘ |(transport = 'air') ∘ delivered| ∘ Part         (U2)
≅   *located ∘ |transport = 'air'| ∘ *delivered ∘ dom ∘ delivered           (U1, S3)
≅   *located ∘ |transport = 'air'| ∘ rng ∘ delivered                        (I1)
≅   ∪/(transport = 'air' → {located}; {}) ∘ rng ∘ delivered                 (M2)
```

The last step shows that it might be possible and desirable to optimize an expression that has passed index optimization even further with respect to plain optimization. Another example is given in the next subsection.

4.3 Putting Things Together: A Proposal for a Combined Optimization

As we have seen in the last example, some of the rules for plain optimization and index optimization in a certain way "contradict" each other which means that rules to be applied in one direction supporting, for example, plain optimization, have to be applied in the opposite direction to facilitate index optimization. A solution to this could be a strategy that first chooses a primary optimization goal (plain optimization or index optimization) and then proceeds along a specific strategy as outlined above. Since the right goal may often be difficult to choose alternatively both goals could be followed and of the two results the one expected to be more efficient could finally be choosen, but this would reduce the efficiency of the optimization process itself. We instead suggest the following procedure:

Combined Index & Plain Optimization

(1) Expand function definitions.
(2) Identify the subset of mappings M (occurring in the query) on which an index is defined.
(3) If M is empty or does not "promise" a sufficient increase in performance, go to step (5).
(4) Try to match an index rule with $\mathtt{f} \in M$. If none matches, apply T- and U-rules and try again.
    If a matching index rule is found, apply it.
    Repeat step (4) until no more index rules are applicable.
(5) Perform plain optimization, that is:
    Move filters to the right. Group filters. Optimize resulting subexpressions.
    Group maps. Optimize resulting subexpressions.
    Group aggregates. Optimize resulting subexpressions.

Clearly, a crucial point is the judgement of the performance opportunities of identified index mappings (cost estimation will be similar to the relational model).

It is interesting to observe that the above scheme suggests, differently from the relational model, that index optimization should be carried out *before* plain optimization. As already discussed in connection with the rule (U1) (see Subsection 4.2), this is not inconsistent with the well-known heuristic to perform, in any case, selections as early as possibly. It rather seems that such transformations are obtained as a by-product of the index optimization. A nice consequence of this is that if step (4) is applied,

in the subsequent step (5) only few transformations are expected to be performed. This can be seen from the fact that the index rules transform expressions, which are possibly amenable to plain optimization, into forms for which only few rules exist to be applied further.

Let us consider a final example that needs preprocessing, that is, application of T- and U-rules, before index optimization can take place and a plain optimization after that. We want to find those parts that are contained in very important orders, that is, with a total of more than 10000.

```
    ∪/shipped ∘ |total > 10000| ∘ Order
≡⃗   ∪/shipped ∘ |+/price ∘ shipped > 10000| ∘ Order                (def. of total)
≡⃗   ∪/shipped ∘ |(+/price > 10000) ∘ shipped| ∘ Order                        (T1)
≡⃗   ∪/id ∘ |(+/price > 10000)| ∘ *shipped ∘ Order                            (U3)
≡⃗   ∪/id ∘ |(+/price > 10000)| ∘ *shipped ∘ dom ∘ shipped                    (S4)
≡⃗   ∪/id ∘ |(+/price > 10000)| ∘ rng ∘ shipped                               (I1)
≡⃗   ∪/((+/price > 10000) → id; {}) ∘ rng ∘ shipped                           (A3)
```

Since optimization schemes as the one above are hardly investigated yet, it seems to be a promising topic for further research.

5 Conclusions and Further Research

We have indicated that functional programming can be utilized profitably in the domain of database programming languages: easy description of semantics, simple and well-structured programs, and, above all, good opportunities for verification and optimization are the basic advantages of FP-like languages. In particular, we have presented algebraic optimization rules that cover and even extend the range of algebraic transformations in the classic relational database model. It is an inherent property of the functional data model that an inverse corresponds very closely to an index access. This in turn makes the algebraic approach to access path optimization possible at all. We have shown how a uniform treatment of the traditional two-step approach to optimization enables the integration of plain optimization and index optimization on an algebraic level. We believe that some new understandings of the optimization process can be gained from this (for instance, the heuristic "selection as early as possible" is partially covered by index optimization) and that the investigation of new optimization strategies is very promising.

The heuristics forming the basis of the optimization strategy presented in Subsection 4.3 do not result from experiments nor from a translation of, for instance, the relational model. They are rather founded on the set of presented rules together with the sketched cost model in mind. So it would be interesting to verify the strategy with a larger set of examples. Along with this, a thorough comparison with the relational algebra could be performed by means of a mapping between the data models.

We have not yet considered optimizations for parallel execution. Since filter, map, and aggregation commute with set-union (the latter again only if the binary function (f) is idempotent) this can be handled easily on the algebraic level, too. For example, the following laws can be employed:

```
*f ∘ (s ∪ t)              ≡⃗  *f ∘ s ∪ *f ∘ t
|p| ∘ (s ∪ t)             ≡⃗  |p| ∘ s ∪ |p| ∘ t
f/g ∘ (s ∪ t)             ≡⃗  f ∘ [f/g ∘ s ∪ f/g ∘ t]
*(s? → f; g) ∘ (s ∪ t)    ≡⃗  *f ∘ s ∪ *g ∘ t
```

(The extensions to unions of more than two arguments are straightforward.) The fact that functional programming is especially amenable to parallel execution is well-understood, but the interesting question is now how to integrate the above rules with the two other optimization strategies. Will access path optimization still be the first to try?

Additional potential for optimization lies in taking a lazy query evaluator [BFN82], thus leaving the strict semantics of FP programs: (i) unnecessary evaluations and database accesses are avoided because of call-by-need parameter passing and (ii) pipelining is already included in the execution of such an interpreter; hence some rules become redundant in the optimization cycle, for example, (M1). These ideas were incorporated into a prototype that was implemented within the work of [Erw89] on a SUN 3/60. This was done similar to the proposals of [BFN82] but was based on a relational instead of a network database system. Notice that the semantics may change in case of recursive function defini-

tions because the existence of a unique least fixpoint is guaranteed only if functions are strict [Wil82].

In [BF79, BFN82] FP was used only as a query language and algebraic optimization was not considered at all, and in [BK90] FP was considered as a target language into which object-oriented languages should be compiled. We have demonstrated that beyond these ideas an FP-like language can definitely serve as a "full" DBPL allowing for sophisticated optimizations.

References

[AB87] ATKINSON, M.P. & BUNEMAN, P.: Types and Persistence in Database Programming Languages, *ACM Computing Surveys Vol. 19*, No. 2, 1987, pp. 105-190.

[ABC^{2}M84] ATKINSON, M.P., BAILEY, P., COCKSHOTT, W.P., CHISHOLM, K.J. & MORRISON, R.: Progress with Persistent Programming, in: STOCKER, P.M., GRAY, P.M.D. & ATKINSON, M.P. (edd.) *Databases - Role and Structure*, Cambridge University Press, 1984, pp. 245-310.

[ACo85] ALBANO, A., CARDELLI, L. & ORSINI, R.: Galileo: A Strongly-Typed, Interactive Conceptual Language, *ACM Transactions on Database Systems Vol. 10,* No. 2, 1985, pp. 230-260.

[AH87] ABITEBOUL, S. & HULL, R.: IFo: A Formal Semantic Database Model, *ACM Transactions on Database Systems Vol. 12*, No. 4, 1987, pp. 525-565.

[Bac78] BACKUS, J.W.: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, *Comm. of the ACM Vol. 21*, No. 8, 1978, pp. 613-641.

[Bac85] BACKUS, J.W.: From Function Level Semantics to Program Transformation and Optimization, *Mathematical Foundations of Sofware Development*, LNCS 185, Springer 1985, pp. 60-91.

[Bel85] BELLOT, P.: Higher Order Programming in Extended FP, *Functional Programming Languages and Computer Architecture*, LNCS 201, Springer 1985, pp. 65-80.

[BF79] BUNEMAN, P. & FRANKEL, R.E.: FQL - A Functional Query Language, *Proc. ACM SIGMoD Conf. on Management of Data*, 1979, pp. 52-58.

[BFN82] BUNEMAN, P., FRANKEL, R.E. & NIKHIL, R.: An Implementation Technique for Database Query Languages, *ACM Transactions on Database Systems Vol. 7,* No. 2, 1982, pp. 164-186.

[BK90] BEERI, C. & KORNATZKY, Y.: Algebraic Optimization of Object-Oriented Query Languages, *Proc. 3rd Int. Conf. on Database Theory*, LNCS 470, Springer 1990, pp. 72-88.

[Bro84] BRODIE, M.L.: on the Development of Data Models, in: BRODIE, M.L., MYLOPOULOS, J. & SCHMIDT, J.W. (edd.) *on Conceptual Modelling*, Springer 1984, pp. 19-48.

[CDFLNR82] CHAN, A., DANBERG, S., FOX, S., LIN, W.K., NORI, A. & RIES, D.: Storage and Access Structures to Support a Semantic Data Model, *Proc. 8th Int. Conf. on Very Large Data Bases*, 1982, pp. 122-130.

[Erw89] ERWIG, M.: Semantic Concepts and Prototype Implementation of a Functional Database Programming Language (in German), Diploma Thesis, Fachbereich Informatik, Universität Dortmund, 1989.

[HK87] HULL, R. & KING, R.: Semantic Database Modelling: Survey, Applications and Research Issues, *ACM Computing Surveys Vol. 19*, No. 3, 1987, pp. 201-260.

[HNSE87] HOHENSTEIN, U., NEUGEBAUER, L., SAAKE, G. & EHRICH, H.D.: Three-Level-Specification of Databases using an Extended Entity-Relationship Model, *Proc. GI-Fachtagung Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen*, Springer 1987, pp. 58-88.

[HW^{3}85] HALPERN, J.Y., WILLIAMS, J.H., WIMMERS, E.L. & WINKLER, T.C.: Denotational Semantics and Rewrite Rules for FP, *Proc. Conf. 12th ACM Symp. Principles of Programming Languages*, 1985, pp. 108-120.

[JK84] JARKE, M. & KOCH, J.: Query Optimization in Database Systems, *ACM Computing Surveys Vol. 16*, No. 2, 1984, pp. 111-152.

[KS81] KIEBURTZ, R.B. & SHULTIS, J.: Transformation of FP Program Schemes, *Proc. ACM Conf. on Functional Programming and Computer Architecture*, 1981, pp. 41-48.

[Nik88] NIKHIL, R.S.: Functional Databases, Functional Languages, in: ATKINSON, M.P., BUNEMAN, P. & MORRISON, R. (edd.) *Data Types and Persistence*, Springer 1988, pp. 51-67.

[oho90] OHORI, A.: Semantics of Types for Database Objects, *Theoretical Computer Science, Vol. 76*, No.1, 1990, pp. 53-93.

[PK90] POULOVASSILIS, A. & KING, P.: Extending the Functional Data Model to Computational Completeness, *Proc. Int. Conf. on Extending Database Technology*, 1990, pp. 75-91.

[Sad87] SADLER, C.: Why Functional Programming, in: EISENBACH, S. (ed.) *Functional Programming: Languages, Tools and Architectures*, Ellis Horwood 1987, pp. 9-17.

[Shi81] SHIPMAN, D.W.: The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems Vol. 6,* No. 1, 1981, pp. 140-173.

[SS86] SRINIVAS, Y.V. & SANGAL, R.: A Generalization of Backus' FP, *Foundations of Software Technology and Theoretical Computer Science*, LNCS 241, Springer 1986, pp. 124-143.

[SS90] SIEG, J.,JR. & SCIORE, E.: Extended Relations, *Proc. IEEE 6th Int. Conf. on Data Engineering*, 1990, pp. 488-494.

[Tri89] TRINDER, P.W.: Referentially Transparent Database Languages, *Proc. Glasgow Workshop on Functional Programming*, 1989, pp. 142-156.

[Ull89] ULLMAN, J.D.: *Principles of Database and Knowledge-Base Systems Vol. II*, Computer Science Press, 1989.

[Wil82] WILLIAMS, J.H.: on the Development of the Algebra of Functional Programs, *ACM Transactions on Programming Languages and Systems Vol. 4,* No. 4, 1982, pp. 733-757.

Appendix

We present the proof to the theorem of Section 4.1 to show how easily one can reason about functional equations.

Proof. (By induction on the number of elements of s).
If s = {} = *g∘s, then f/g∘{} = unit∘f = f/id∘{} = f/id∘*g∘s (by def. of /) .
If s ={h}, then f/g ∘{h} = g ∘h = id∘g ∘h = f/id∘{g ∘h} = f/id∘*g∘{h} (by defs. of /,id,and *).
For s = {h i} (h ≠ i) we have:

f/g ∘{h i} =
f ∘[g ∘h g ∘i] — def. of /

Now we distinguish two cases:

Case 1: g ∘h = g ∘i. Then

f ∘[g ∘h g ∘i] =
g ∘h = — (ii)
f/id∘*g∘{h} = — (see Case s = {h})
f/id∘*g∘{h i} — def. of *

Case 2: g ∘h ≠ g ∘i. Then

f ∘[g ∘h g ∘i] =
f ∘[g ∘h f/id∘{g ∘i}] = — def. of /
f/id∘{g ∘h g ∘i} = — def. of /
f/id∘*g∘{h i} — def. of *

Now we assume that the theorem is true for t where [h t] := split∘s (s ≠ {} and t ≠ {}). Note that t has exactly one element less than s. We let [i u] := split∘t. Since the choice of a concrete partition does not affect the result of an aggregation we can assume w.l.o.g.

(iii) split∘*g∘t = [g∘i *g∘v] with v = |g ≠ g∘i| ∘u
(iv) split∘*g∘s = [g∘h *g∘t] if g∘h ∉ *g∘t, see Case 2 below

In equation (iii) v denotes those elements being mapped by g to a different value than i. This ensures the validity of the equation. We conclude further:

f/g ∘ s =
f ∘ [g ∘ #1 f/g ∘ #2] ∘ [h t] = def. of /
f ∘ [g ∘ h f/id ∘ *g ∘ t] by induction hypothesis
Case 1: ∃ i ∈ t: g ∘ i = g ∘ h
f ∘ [g ∘ h f/id ∘ *g ∘ t] =
f ∘ [g ∘ h f ∘ [id ∘ #1 f/id ∘ #2] ∘ split ∘ *g ∘ t] = def. of /
f ∘ [g ∘ h f ∘ [g ∘ i f/id ∘ *g ∘ v]] = (iii)
f ∘ [f ∘ [g ∘ h g ∘ i] f/id ∘ *g ∘ v] = (i)
f ∘ [g ∘ i f/id ∘ *g ∘ v] = (ii)
f ∘ [id ∘ #1 f/id ∘ #2] ∘ split ∘ *g ∘ t = (iii)
f/id ∘ *g ∘ t = def. of /
f/id ∘ *g ∘ (t ∪ {h}) = def. of *
f/id ∘ *g ∘ s
Case 2: ∀ i ∈ t: g ∘ i ≠ g ∘ h
f ∘ [g ∘ h f/id ∘ *g ∘ t] =
f ∘ [id ∘ #1 f/id ∘ #2] ∘ split ∘ *g ∘ s = (iii)
f/id ∘ *g ∘ s def. of /

■