SECONDO/QP: Implementation of a Generic Query Processor

Ralf Hartmut Güting [1]
Stefan Dieker [1]
Claudia Freundorfer [1]
Ludger Becker [2]
Holger Schenk [1]

[1] Praktische Informatik IV
FernUniversität Hagen
D-58084 Hagen, Germany

[2] Westfälische Wilhelms-Universität
FB 15 - Informatik, Einsteinstr. 62
D-48149 Münster, Germany

Abstract: In an extensible database system, evaluation of a *query plan* is done in cooperation between a collection of *operator implementation functions* and a component of the DBMS that we call the *query processor*. Basically, the query processor constructs an operator tree for the query plan and then calls an *evaluator* function which traverses the tree, calling the operator functions in each node. This seemingly simple strategy is complicated by the fact that operator functions must be able to call for the evaluation of parameter expressions (e.g. predicates), and must be able to process *streams* of objects in a pipelined manner.

Although query processing along these lines is implemented in most database systems, and certainly in all extensible database systems, the details of programming the parameter passing, organizing the interaction between stream operators, etc. are tricky, and seem to be buried in the code of the respective systems. We are not aware of any simple, crisp, clear published exposition of how one could implement such a query processor. This is what the paper offers.

Moreover, we feel the solution presented here is particularly simple, elegant, and general. For example, it is entirely independent from the data model being implemented, admits arbitrary parameter functions, and allows one to mix freely stream operators and other operators. The construction of the operator tree, shown in the paper, includes complete type checking and resolution of overloading. The query processor has been implemented within the SECONDO system; the source code is available.

1 Introduction

Extensible database systems have been studied since about the mid-eighties. The main motivation has been the support of new application areas, especially by making it possible to introduce application-specific data types (e.g. polygons or images), and the achievement of a cleaner system architecture which allows for easier changes such as the introduction of new access methods or query processing algorithms. Two main directions can be distinguished. The first one is to select a specific data model and to implement for it a system with well-defined interfaces for user-defined extensions. This approach was taken, for example, in the POSTGRES [StRH90], Starburst [Haas90], Gral [Gü89], PREDATOR [SeLR97], and Paradise [Pate97] projects as well as in commercial systems like Informix Universal Server [Inf98]; in these cases, the model is an extended relational, or as it is called today, object-relational model [St96].

The second approach strives for more generality and attempts to even leave the data model open. In that case, no system is offered; instead, a *toolkit*, a collection of powerful tools for building database systems is provided, for instance a general storage manager or an optimizer generator. Major proponents of this approach are EXODUS [Care86] and its successor SHORE [Care94], GENESIS [Bato88], DASDBS [Sche90] and Volcano [Gr94]. Toolkits "have essentially proven to be dead-ends"; this is at least the view of Carey and DeWitt [CaD96]. The main reason was that it was too difficult to construct a DBMS using the toolkit; also the tools offered were in some cases not flexible enough to entirely match what was needed in the application DBMS.

The goal of SECONDO is to offer a generic "database system frame" that can be filled with implementations of a wide range of data models, including, for example, relational, object-oriented, graph-oriented or sequence-oriented DB models. The strategy to achieve this goal is to separate the data-model independent components and mechanisms in a DBMS (the *system frame*) from the data-model dependent parts. Nevertheless, the frame and the "contents" have to work together closely. With respect to the different levels of query languages in a DBMS, we have to describe to the system frame:

- the *descriptive algebra*, defining a data model and query language,
- the *executable algebra*, specifying a collection of data structures and operations capable of representing the data model and implementing the query language,
- *rules* to enable a query optimizer to map descriptive algebra terms to executable algebra terms, also called *query plans* or *evaluation plans.*

A general formalism serving all of these purposes has been developed earlier, called *second-order signature (SOS)* [Gü93]. It is reviewed in Section 2.

At the system level, definitions and implementations of type constructors and operators of the executable algebra are arranged into *algebra modules*, interacting with the system frame through a small number of well-defined support functions for manipulation of types and objects as well as operator invocation.

This paper reports on the core part of the SECONDO system frame, the *query processor*. The task of a query processor is the evaluation of a query plan, that is, an expression of the physical algebra. The basic strategy is to first construct an operator tree for the expression, and then to call an *evaluator* function which traverses the tree, calling *operator functions* in each node. This is complicated by the fact that operator functions must be able to call for the evaluation of parameter expressions (e.g. predicates) and must be able to process streams of objects, working interleaved.

The paper is organized as follows. In Section 2, we review the concept of second-order signature. Section 3 describes the interaction between the query processor and algebra modules from the algebra implementor's point of view. Section 4 describes the evaluation of queries, explaining the construction of the operator tree, type checking, resolution of overloading, and evaluation control. Section 5 discusses related work, and Section 6 concludes the paper.

2 Second-Order Signature

We can give here only a very brief introduction to second-order signature. For a more complete exposition, see [Gü93]. A sophisticated OO and graph data model has been specified in SOS in [BeG95]. The idea is to use two coupled signatures. The first signature has so-called *kinds* as sorts and *type constructors* as operators. The terms of this signature are called *types*. This set of types will be the type system, or equivalently, the data model, of a SECONDO DBMS. The second signature uses the *types generated by the first signature* as sorts, and introduces *operations* on these types which make up a query algebra.

As an example data model specification we choose a small relational model at the execution level (physical model and algebra) which will be used further in the paper. Although the purpose of SECONDO is not to reimplement relational systems, it makes no sense to explain an unknown formalism using examples from an unknown data model.

kinds IDENT, DATA, NUM, TUPLE, REL
type constructors

	→ DATA	<u>*int*</u>, <u>*real*</u>, <u>*bool*</u>, <u>*string*</u>
	→ NUM	<u>*int*</u>, <u>*real*</u>
$(\text{IDENT} \times \text{DATA})^+$	→ TUPLE	<u>*tuple*</u>
TUPLE	→ REL	<u>*rel*</u>

Any term of this signature is a *type* of the type system. Hence <u>*int*</u> is a type, and

<u>*rel*</u>(<u>*tuple*</u>(<(name, <u>*string*</u>), (country, <u>*string*</u>), (pop, <u>*int*</u>)>))

is a type as well, describing the type of a relation for *cities*. Since this is a physical model, for each type constructor there must be a representation data structure in the system. – The second signature uses the types generated by the first signature as sorts, and defines a collection of *operators*, in this case for a query processing algebra. Since there are in general infinitely many sorts generated by the first signature, we use *quantification over kinds* to specify polymorphic operators. Hence kinds play a dual role: They control the applicability of type constructors, and they are used for the specification of operators.

operators
$\forall$ num_1 **in** NUM. $\forall$ num_2 **in** NUM.

$num_1 \times num_2$	→ <u>*bool*</u>	$<, >, \leq, \geq, =, \neq$

This defines comparison operators for any combination of <u>*int*</u> and <u>*real*</u> types. Here num_1 and num_2 are type variables that can be bound to any type in the respective kind. A type is *in* the result kind of its outermost type constructor.

$\forall$ *tuple in* TUPLE.

<u>*rel*</u>(*tuple*) × (*tuple* → <u>*bool*</u>)	→ <u>*rel*</u>(*tuple*)	**scanselect**
<u>*rel*</u>(*tuple*)	→ <u>*stream*</u>(*tuple*)	**feed**
<u>*stream*</u>(*tuple*) × (*tuple* → <u>*bool*</u>)	→ <u>*stream*</u>(*tuple*)	**filter**

These operators implement selection on a relation by scanning it[1], feed a relation into a stream, and filter a stream by a predicate, respectively. Note that the kind IDENT and the type constructor <u>*stream*</u> are not defined in the data model above. They can be viewed as predefined since their meaning is in fact hard-coded into the query processor, as we will show in Section 4. – Finally, we will use an example *attr* operator to access components of a tuple:

$\forall$ *tuple*: <u>*tuple*</u>(*list*) **in** TUPLE, *attrname* in IDENT, *member(attrname, attrtype, list).*
tuple $\times$ *attrname* $\rightarrow$ *attrtype* **attr**

Here *member* is a *type predicate* that checks whether a pair (x, y) with x = *attrname* occurs in the *list* making up the tuple type definition. If so, it binds *attrtype* to y. Hence **attr** is an operation that for a given tuple and attribute name returns a value of the data type associated with that attribute name. – Type predicates are implemented “outside” the formalism in a programming language. An example expression, or *query plan* using these operators, would be:

cities **scanselect**[**fun** (c: city) **attr**(c, pop) > 500000]

This assumes a *cities* relation of the type shown earlier, with tuple type *city*. Note that there is a general notation for function abstractions, of the form

fun $(x_1: t_1, \ldots, x_n: t_n)$ *expr*

where the x_i are free variables in *expr* and the t_i their types. – A SECONDO system executes a fixed set of commands. Its most important *basic commands* are:

type <identifier> = <type expression>	**update** <identifer> := <value expression>
delete type <identifier>	**delete** <identifier>
create <identifier> : <type expression>	**query** <value expression>

Here the first two commands manipulate named types, the next three named objects in the database. *Type expression* refers to a type of the current type system and *value expression* to an expression of the associated algebra. Note that this is the way how the fixed set of SECONDO commands cooperates with the variable data model. Hence we can create the *cities* relation by commands:

type city_rel = <u>*rel*</u>(<u>*tuple*</u>(<(name, <u>*string*</u>), (country, <u>*string*</u>), (pop, <u>*int*</u>)>))
create cities: city_rel

The implemented query processor manipulates all commands, type expressions, and value expressions, in the form of *list expressions*, as in the language *Lisp*. In this notation, type constructors and operators are written as the first element of a list, followed by their arguments. Hence the type of the *cities* relation is represented as:

```
(rel (tuple ((name string) (country string) (pop int))))
```

and the query shown above is represented as

```
(scanselect cities (fun (c city) (> (attr c pop) 500000)))
```

1. Notice that we do not advocate the creation of intermediate relations as selection results, but rather introduce operator **scanselect** in this paper to give a simple example of an operator using a parameter function, with an evaluation function implementation as brief as presented in Section 3.2.

3 Interaction of Query Processor and Algebra Modules

3.1 Overview

Type constructors and operators are implemented within *algebra modules*. The algebra implementor must provide a fixed set of support functions which are registered with the system frame, thus becoming callable by the query processor. In the scope of this paper, we are only interested in the support functions presented in Table 1. An overview of the entire SECONDO system, including a complete list of support functions, can be found in [DiG99].

`Evaluate`	Computes the result value from input values. In case of overloading, several evaluation functions exist.
`Select`	Selects the correct evaluation function in case of overloading by means of the actual argument types.
`TransformType`	Computes the operator's result type from given argument types (*type mapping*).

Table 1: Operator support functions (subset)

The query processor, on the other hand, offers primitives which can be used within implementations of evaluation functions as listed in Table 2.

Handling of parameter functions	`argument`	Gives access to the argument vector of a parameter function.
	`request`	Calls a parameter function.
Handling of stream operators	`open`	Initializes a stream operator.
	`request`	Triggers delivery of the next stream element.
	received	Returns true if the preceding request was successful, otherwise false.
	`close`	Closes a stream operator.

Table 2: Query processor primitives

In the rest of this section we show how evaluation functions are implemented using the query processor primitives (Section 3.2) and explain the basic techniques for the implementation of type mapping functions (Section 3.3). We omit a detailed description of the implementation of `Select` functions, because they are an easier variant of the `TransformType` functions.

3.2 Implementation of Evaluation functions

The interface for `Evaluate` operator support functions written in Modula-2 is defined as follows.[2]

2. SECONDO supports algebra module implementations in C and C++, too. We prefer Modula-2 for code examples because it is best suited to show technical details while not requiring specific programming language expertise.

```
PROCEDURE alpha (arg    : ArgVector;
             VAR result : WORD;         (* out *)
                 message: INTEGER;
             VAR local  : ADDRESS       (* in/out *)) : INTEGER;
```

Here *arg* is a vector of operands; each function knows how many operands it has. Each operand is represented in one word of storage.[3] The result value is returned in the same form in the parameter *result*. The *message* and *local* parameters are needed only for stream operators; they are explained below. The return value of the function is 0 if no error occurred; for stream operators it has an additional meaning explained below. This interface is sufficient to implement "simple" operators that just take some arguments and return a value, operators with parameter functions, and stream operators. We give an example implementation for each kind of operator.

Example (Simple Operator). Consider a simple addition operation on integers. The signature is:

```
int x int    ->   int  +
```

The evaluation function is trivial, but shows how the generic interface is used:

```
PROCEDURE add (arg    : ArgVector;
           VAR result : WORD;         (* out *)
               message: INTEGER;
           VAR local  : ADDRESS       (* in/out *)) : INTEGER;
BEGIN
  result := INTEGER(arg[1]) + INTEGER(arg[2]); RETURN 0
END add;
```

Type casting is necessary to consider the arguments as integers when accessing the argument vector.

Example (Operator with Parameter Function). The parameter function will be given by an address within the argument vector *arg* in the proper position (e.g. *arg*[2], if the second argument is a function). Hence the argument vector contains values as well as what we call *suppliers*; a supplier is either a function argument or a stream operator. (In the implementation, the parameter function is represented by a subtree of the operator tree, and the supplier is a pointer to the root of that subtree).

A procedure implementing an operator with a parameter function will at some point have to call this function explicitly (perhaps several times); for each call it has to set the arguments and to receive the result using the `argument` and `request` primitives, respectively.

Let us illustrate this by a procedure implementing the *scanselect* operator of Section 2 with signature:

```
forall tuple in TUPLE.
   rel(tuple) x (tuple -> bool) -> rel(tuple)   scanselect
```

We assume the existence of the following operations for reading and updating the relation representation:

```
CreateRelation        Append          CreateScan          Next
DestroyRelation                       DestroyScan         EndOfScan
                                                          Get
```

3. To achieve uniformity and simplicity in the query processor, we assume that any type of any algebra can be represented in a single word of storage. For simple values such as an integer, boolean, etc., this is the case; for larger structures, it can be a pointer; for persistent values currently not in memory, it can be an index into a catalog table describing how the persistent value can be accessed.

Then the evaluation function could be coded as follows.

```
PROCEDURE scanselect (arg    : ArgVector;
                  VAR result : WORD;         ( * out *)
                      message: INTEGER;
                  VAR local  : ADDRESS       (* in/out *)) : INTEGER;
VAR r: relation_scan; s: relation; t: tuple; value: INTEGER;
  valid: BOOLEAN; vector: ArgVectorPointer;
BEGIN
  r := CreateScan(relation(arg[1])); s := CreateRelation();
  ALLOCATE(t, TSIZE(TupleRec)); vector := argument(arg[2]);
  WHILE NOT EndOfScan(r) DO
    t := Get(r); vector^[1] := t;
    request(arg[2], value); valid := VAL(BOOLEAN, value);[4]
    IF valid THEN Append(s, t) END; Next(r)
  END;
  DEALLOCATE(t, TSIZE(TupleRec)); DestroyScan(r); result := s; RETURN 0
END scanselect;
```

The essential point is the treatment of the parameter function of type (*tuple* → *bool*) which is the second argument of *scanselect* and therefore represented by *arg*[2]. This function has a single argument of type *tuple*; hence the first argument of its argument vector is set to *t*.

Example (Stream Operator). A stream operator somehow manipulates a stream of objects; it may consume one or more input streams, produce an output stream, or both. For a given stream operator α, let us call the operator receiving its output stream its *successor* and the operators from which it receives its input streams its *predecessors*. The basic way how stream operators work is as follows. Operator α is sent an *open* message to initialize its stream state. After that, each *request* message makes α try to deliver a stream object. Operator α returns a *yield* message in case of success, or a *cancel* message if it does not have an object any more. A *close* message indicates that α's successor has abandoned processing stream elements provided by α.

When we try to simulate stream operators by algebra functions, one can first observe that a function should not relinquish control (terminate) when it sends a message to a predecessor. This can be treated pretty much like calling a parameter function. However, the function needs to terminate when it sends a *yield* or *cancel* message to the successor. This requires the function to be re-entrant, using some local memory to store the state of the function.

In the general evaluation function interface given above, parameter *message* is used to send to this operator one of the messages *open*, *close*, or *request*, coded as integers. Upon *request*, this procedure will send in its return parameter a message *yield* or *cancel* to the successor. The parameter *local* can be used to store local state variables that must be maintained between calls to *alpha*. If more than a one-word variable is needed, one can define and allocate a record (type) within *alpha* and let *local* point to it.

Let us now consider the implementation of the stream operator *filter* using the primitives for stream processing introduced above.

```
forall tuple in TUPLE.
   stream(tuple) x (tuple -> bool) -> stream(tuple)   filter
```

4. VAL is a type transfer function in Modula-2; it converts from INTEGER (compatible with WORD) to BOOLEAN.

The *filter* operation has an input stream, an output stream, and a parameter function. It transmits from its input stream to the output stream only tuples fulfilling the condition expressed by the parameter function.

```
PROCEDURE filter (arg    : ArgVector;
              VAR result : WORD;         ( * out *)
                  message: INTEGER;
              VAR local  : ADDRESS       (* in/out *)) : INTEGER;
VAR t: tuple; found: BOOLEAN; value: INTEGER; vector: ArgVectorPointer;
BEGIN
  CASE message OF
    OPEN:    open(arg[1]); RETURN 0
    REQUEST: request(arg[1], t); found := false;
             vector := argument(arg[2]);
             WHILE received(arg[1]) AND NOT found DO
               vector^[1] := t; request(arg[2], value);
               found := VAL(BOOLEAN, value);
               IF NOT found THEN request(arg[1], t) END
             END;
             IF found THEN result := t; RETURN YIELD
             ELSE RETURN CANCEL
             END
    CLOSE:   close(arg[1]); RETURN 0
  END
END filter;
```

3.3 Implementation of Type Mapping Functions

In SECONDO, we exploit the concept of nested lists, well known from functional programming languages, as a generic means to represent type expressions, queries, and constant values. Nested lists are very useful for the representation of query expressions or type expressions, since they are flexible and easy to manipulate. We have written a little tool called *NestedList* which represents lists as binary trees and offers the basic operations like *Cons*, *First*, *Rest*, etc., as in the language Lisp. This tool has a Modula-2 as well as a C, C++, and Java interface. The tool considers the *atoms* occurring in the lists as typed, and there is a fixed set of atom types:

• *Int*	70, -5	• *String*	"Hagen"
• *Real*	3.14	• *Symbol*	filter, >
• *Bool*	FALSE		

Actually, there is a sixth atom type called *Text* to represent long texts which is not relevant here. The *NestedList* tool allows one to read a list from a character string or a file.

An operator's `TransformType` function is called during query annotation with a parameter of type *ListExpr* which is a list of the types of the arguments. The tasks of `TransformType` are

(i) to check that all argument types are correct, and
(ii) to return a result type, again as a list expression.

In addition to that, a `TransformType` function may

(iii) extract argument type information which might be useful for the operator's evaluation function.

Since query expressions may be entered by users directly, `TransformType` should be prepared to handle all kinds of wrong argument type lists, and return a symbol atom *typeerror* in such a case. – As an example, here is the `TransformType` function for the *filter* operator. The transformation of type lists to be performed is

```
((stream x) (map x bool))  -> (stream x)
```

This is implemented in procedure `filtertype` as follows.

```
PROCEDURE filtertype (args: ListExpr) : ListExpr;
VAR first, second : ListExpr;
BEGIN
  IF ListLength(args) = 2 THEN
    first := First(args); second := Second(args);
    IF (ListLength(first) = 2) AND (ListLength(second) = 3) THEN
      IF (TypeOfSymbol(First(first)) = stream) AND
         (TypeOfSymbol(First(second)) = map) AND
         (TypeOfSymbol(Third(second)) = bool) AND
         Equal(Second(first), Second(second))
      THEN RETURN first
      END
    END
  END; RETURN SymbolAtom("typeerror")
END filtertype;
```

Here `Equal` is a procedure which checks for deep equality of its argument lists.

In `filtertype` we did not need to make use of the possibility to pass argument type information to the `TransformType` function (task (iii) above). But consider, for instance, relational operators which allow the user to identify attribute values by name like the *attr* operator or a projection operator. For example, using *attr* the user would like to write an attribute name, as in `(attr c pop)`. However, the `Evaluate` function does not have access to the tuple type any more, and needs a number, the position of the attribute called *pop* within the tuple, to access that component efficiently. For that purpose, the `TransformType` function of *attr* computes the information needed and returns it in addition to the pure result type *t* using the keyword APPEND: If a `TransformType` function returns a list `(APPEND inf t)` rather than just *t*, where *inf* is a sublist containing the additional information, *inf* will be appended to the original list of arguments during annotation of the query, explained below, and will be treated as if it had been written by the user. In Appendix B the implementation of this mechanism is shown.

4 Evaluation of Query Plans

4.1 Structure of the Operator Tree

The operator tree has three kinds of nodes called *Object*, *Operator* and *IndirectObject*. An *Object* node represents some kind of value (a database object or a constant), an *Operator* node an algebra operator. An *IndirectObject* node represents a value accessible through the argument vector of a parameter function, hence, a parameter to that function. The actual definition of the data structure is given in Appendix A.

Each kind of node has a field *evaluable* which is true iff this node is an object, an indirect object, or an operator which is neither (the root of a subtree representing) a function argument nor a stream argument. This means the value of this subtree can be computed directly. A node can then have one of the three forms mentioned above. It can represent an object (a simple value or a pointer); in that case a field *value* contains that object. It can be an "indirect object" accessible through an argument vector attached to a subtree representing a function argument; then a field *vector* points to that argument vector and a field *argIndex* is the position of the object within the argument vector.

Finally, the node can represent an operator with information as follows. Fields *algebraId* and *opFunId* identify the operator's evaluation function, *noSons* is the number of arguments for this operator, *sons* is an array of pointers to the sons, *isFun* is true iff the node is the root of a function argument, and *funArgs* is a pointer to the argument vector for a function node, only used, if *isFun* is true. Field *isStream* is true if this operator produces an output stream, *local* is used to keep the *local* parameter of a stream operator between calls, and *received* is true iff the last call of this stream operator returned YIELD.

The three kinds of nodes are represented graphically as shown in Figure 1. For an operator node, the top left field shows the operator rather than its *algebraId* and *opFunId*. The other fields in the top row are *noSons*, *isFun*, *funArgs*, and *isStream*; the last two fields are omitted in this representation. The bottom row, of course, shows the *sons* array.

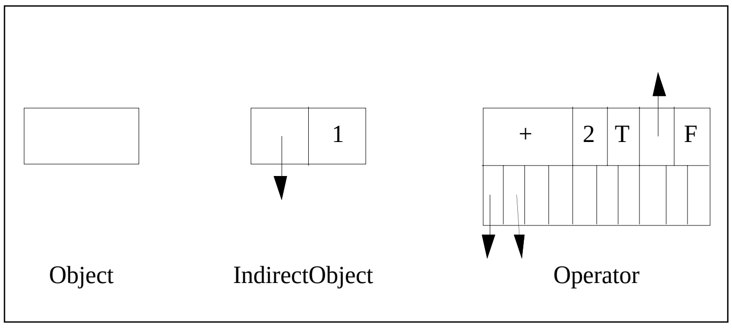

Figure 1: Types of operator nodes

The structure of the operator tree is illustrated by the representation of the following query plan:

```
(filter (feed cities)
        (fun (c city)
             (>   (attr c pop .)
                  500000)))
```

Here *attr* is an operator with three arguments, namely, a tuple, an attribute name within that tuple, and a number giving the position of the attribute within the tuple. However, the user does not have to supply this number; it will be inferred and added in type checking (see Section 3.3). This is indicated by the dot, which is not written, only shown here to indicate the missing argument. The operator tree for this query is shown in Figure 2. Here oval nodes represent data objects represented externally from the operator tree. At the bottom an argument vector is shown.

4.2 Constructing the Operator Tree

The operator tree is constructed following two major steps:

1. The query is analyzed and *annotated*, i.e. an expanded list expression is returned with additional

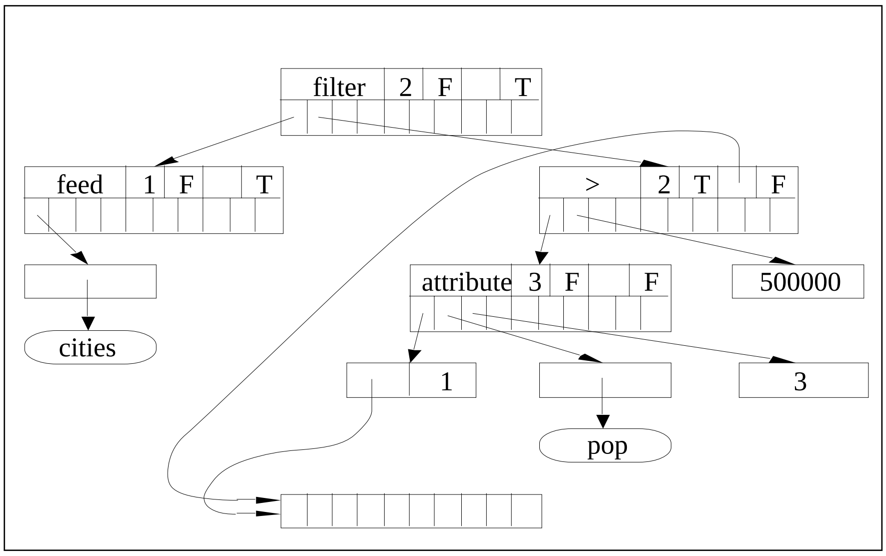

Figure 2: Sample operator tree

information about all symbols and substructures occurring in the query. Annotating the query requires interaction with the algebra modules in two ways. First, whenever an application of an operator α to a list of arguments is recognized, the *types* (type expressions) of the arguments are collected and the `TransformType` function for operator α is called with this list of types. `TransformType` checks that the argument types are correct and returns a result type (or the symbol “typeerror”). Type expressions are also represented and manipulated as nested lists.

Second, *overloading* of operators is resolved. An operator always has at least one, but may have several evaluation functions for different combinations of argument types; in the latter case we call it overloaded. Hence, whenever an application of operator α to a list of arguments is recognized, α’s `Select` support function is called. It is given the list of types of the arguments and the *index* of the operator (the number of the operator within its algebra); it returns the index of the appropriate evaluation function.

2. The operator tree is constructed from the annotated query.

In the rest of this section, we discuss annotating the query, type mapping, and finally the actual building of the operator tree from the annotated query.

4.2.1 Annotating the Query Expression

A query or query plan is given as a nested list. Here is once more the example introduced above:

```
(filter (feed cities)
        (fun (c city)
             (>   (attr c pop)
                  500000)))
```

It may have been entered by a user directly in this form, or be the result of processing of a parser front-end, or of the optimizer. Given a query in the form shown above, we can distinguish between the five types of atoms. The first four will be interpreted as constants of four basic types offered in a *StandardAlgebra* within SECONDO. The fifth, *Symbol*, is used to represent operators, object names, special symbols like *fun*, etc. In the query above, only 500000 is an *Int* atom; all other are *Symbol* atoms.

In annotating the query, the meaning of every atom and of every sublist is analyzed and returned explicitly together with that atom or list. What are the possible structures of a query? An *atom* can be:

(i) a *constant* of one of the four data types *int*, *real*, *bool*, or *string*
(ii) an *operator*
(iii) the name of an *object* in the database (which is not a function object)
(iv) the name of a *function* object in the database (see [Gü93]).
(v) the name of a *variable* defined as a parameter in some enclosing *abstraction* (function definition) in the query
(vi) an *identifier* (used, for example, as an attribute name)

A *list* can be:

(vii) an *application of an operator* α to some arguments of the form `(alpha a1 ... an)`
(viii) an *application of a database function* object *f* to some arguments: `(f a1 ... an)`
(ix) an *application of an abstraction* to some arguments:
`((fun (x1 t1) ... (xn tn) expr) a1 ... an)`
(x) an *abstraction* (function definition) of the form `(fun (x1 t1) ... (xn tn) expr)`
(xi) a *list of arguments* for some operator: `(a1 ... an)`. Operators in SECONDO may have arguments that are themselves varying length lists of arguments (for example, a list of attribute names for a projection operator).
(xii) an empty list: this is interpreted as an empty *list of arguments*.

A procedure *annotate* does the annotation; for a given subexpression (atom or list) *s*, it returns in general a structure of the form

```
((s <descriptor> …) <type>)
```

The descriptor is a keyword identifying the cases (i) through (xii) above, keywords are

```
constant, operator, object, function, variable, identifier,
applyop, applyfun, applyabs, abstraction, arglist
```

After the descriptor there can be further list elements giving specific information for this descriptor. For example, a constant is annotated as

```
<value>      ->   ((<value> constant <index>) <type>),

7            ->   ((7 constant 1) int)
```

Whenever a constant or a database object is annotated, its value, given as a WORD, is entered into a global ARRAY OF WORD *Values* at an index *valueno*, and *valueno* is incremented. The 1 after the descriptor *constant* is the index under which this constant was stored.

The second element of the annotation is always the type of the element annotated, except for operators, where no type is needed. Operators are annotated as

```
<op>         ->   ((<op> operator <algebraId> <operatorId>) none)
```

For an operator + as in Section 4.2 this would result in

```
+            ->   ((+ operator 1 1) none)
```

The application of an operator to some arguments is annotated as

```
(<op> <arg1> … <argn>)
        ->   (    (none applyop (ann(<op>) ann(<arg1>) … ann(<argn>))))
                  <resulttype>
                  <opFunId>)
```

If a list *s* is annotated, then the original list *s* is not repeated in the annotation, instead we have a symbol *none* in the first position of the first sublist. The third element of the first sublist is the annotated version of the operator application. The *resulttype* was determined by calling the `TransformType` function of the operator. The *opFunId* is the index of the evaluation function for this type combination and has been determined by a call of the `Select` function for this operator.

Hence, for example, the complete annotation of a query expression `(- 7 1.100)` would be

```
(   (none applyop
        (   ((- operator 1 2) none)
            ((7 constant 1) int)
            ((1.100 constant 2) real)))
    real
    6)
```

Here the result type is *real* and the evaluation function for *int* and *real* arguments has index 6. A shortened version of procedure *annotate* is shown in Appendix B. Most cases are omitted, but the overall structure is given, and annotation of an operator application, including the calls of the operator's type mapping function and selection function, is shown completely. – To annotate abstractions of the form `(fun (x1 t1) … (xn tn) expr)`, *annotate* calls a procedure *annotate_function*. This procedure first increments a global counter for function definitions *fno*. Then it processes each parameter definition (x_i t_i) by entering name x_i, position *i*, function number *fno*, and type expression t_i into a table for variables. Finally, it calls *annotate* again to annotate *expr*. Within *expr*, occurrences of the names x_i will be recognized and annotated as *variables* with all the information mentioned above, so that variables can be translated into *IndirectObject* nodes of the operator tree later. *Annotate_function* also collects the types t_1, …, t_n and the result type *t* of *expr* (returned by *annotate*) and builds the type expression of the whole abstraction which is `(map t1 … tn t)`.

4.2.2 Building the Operator Tree

The operator tree is constructed from the annotated query expression by a procedure *subtree*. This is a relatively simple matter, since all necessary information has been collected in the annotation. A part of procedure *subtree* is shown in Appendix C; that part gives the flavor and should be sufficient to process the little annotated query:

```
(   (none applyop
        (   ((- operator 1 2) none)
            ((7 constant 1) int)
            ((1.100 constant 2) real)))
    real 6)
```

After annotating the query, it is known how many *abstractions* (function definitions) it contains. Before *subtree* is called, a corresponding number of argument vectors is allocated and assigned to the fields of a global array *ArgVectors*. Since the annotations of *abstractions* as well as those of *variables* contain function numbers, procedure *subtree* can set pointers to the argument vectors indexed by that number when nodes for *IndirectObjects* and root nodes of function expressions are processed.

4.3 The Evaluation Procedure

Evaluation is done in close cooperation between a recursive function *eval* of the query processor, applied to the root of the operator tree, and the operators' evaluation functions. Function *eval* has the basic structure as follows.

```
function eval (t : node): WORD;
input: a node t of the operator tree;
output: the value of the subtree rooted in t;
method:
  if t is an object or indirect object node then lookup the value and return it
  else {t is an operator node}
    for each subtree $t_i$ of t do {evaluate all subtrees that are not functions or streams}
      if the root of $t_i$ is marked as function (F) or stream (S) then $arg_i := t_i$
      else $arg_i := eval(t_i)$
      end
    end;
    Call the operator's evaluation function with argument vector arg and return its result
  end
end eval
```

The actual source code for *eval* can be found in Appendix D. The basic strategy is, of course, to evaluate the tree bottom-up, calling an operator's evaluation function with the values of the argument subtrees that have been determined recursively by *eval*. In case of parameter functions (F) or stream operators (S), however, evaluation control is passed to the evaluation function of the superior operator, which in turn uses the *request* primitive to call for specific subtree evaluation, as demonstrated in Section 3.2.

5 Related Work

We briefly compare with four projects that have studied generic query processors more deeply, namely GENESIS, EXODUS, Starburst, and Volcano. All these projects have lots of interesting concepts and results, but we can only consider the issue of this paper here. GENESIS [BaLW88] early on emphasized stream processing; they even adapted the data model level, a functional model similar to DAPLEX and FQL to use stream rewriting rules ("productions") as a querying primitive. A production is implemented by a *stream translator*, a box with input and output streams; such boxes are arranged into a translator network to describe a query plan. Translators communicate via mailboxes (shared variables). A translator is implemented by a function which branches on the various kinds of input *tokens*. An important feature not found in the other approaches is that streams have a nested structure; delimiters (braces) are part of the stream. In contrast to SECONDO, *all* operators have to be expressed as stream operators, even simple arithmetic or comparison operators. Parameter expressions are treated as streams as well. Mechanisms for setting up the translator network and for controlling execution are not shown.

EXODUS [RiC87, RiCS93] offers a language E, an extension of C++ designed for implementing database systems. E provides an *iterator* construct which allows one to initialize a stream, then request elements in a loop, and terminates when the stream is exhausted. An iterator actually consists of the *iterator loop* construct and some *iterator functions* called in the loop. An iterator function sends results to the caller by means of a *yield* command. This is all very similar to our stream protocol which was certainly inspired by EXODUS. Operator functions written in E are slightly more elegant than those of SECONDO. However, the environment for executing such functions is much more complex since it relies on compiling the language E. Implementing operators that need access to type/schema information requires a rather complex interplay between the operator function written by the database implementor, and pieces of code generated by the E compiler which has access to schema information. E is also firmly tied to the C++ environment and the compiling strategy, which would make it difficult or impossible to design a query processing system as a collection of algebras which can be written independently and in various languages (currently Modula-2, C, and C++ are supported in SECONDO, we plan extensions for Java and possibly other languages).

Starburst [Haas90, Haas89] uses query plans composed of algebraic operators called *LOLEPOPs* (low level plan operators). A LOLEPOP has several associated routines (e.g. a cost function, property mapping) and especially an interpretation routine (= evaluation function). These take one or more input streams and produce an output stream. However, the implementation of this concept, and the interface for writing interpretation routines is not shown in the papers. A specialty is that parameter expressions (predicates) are translated into programs for a stack machine.

Finally, Volcano [Gr94] is the closest in spirit to SECONDO. Volcano also emphasizes data model independence (as already EXODUS did). Operators are implemented as iterators observing an *open - next – close* protocol equivalent to our stream protocol. A slight technical difference here is that an operator is realized by three different functions, e.g. *open-filter*(), *next-filter*(), and *close-filter*() instead of our branching on messages in a single procedure. Parameter expressions are made available to operator functions as so-called *support functions*. However, it is not clear how support functions for expressions are constructed, and the precise mechanism for calling them is not given. In Volcano, *all* operators are stream operators (operators like +, > in expressions seem to be viewed as a different category). A difference to SECONDO is also that the environment recursively calls all *open* (and later *close*) functions, hence this is not controlled by the operator implementations as in SECONDO. Mixing stream and non-stream operators is not possible. Constructing operator trees from textual query plans is not shown. On the other hand, Volcano offers very interesting concepts such as *dynamic query plans* and transparent switching to a parallel execution, and is already a much more complete system than SECONDO, including an optimizer generator [GrM93].

6 Conclusions

SECONDO is a generic development environment for non-standard multi-user database systems. At the bottom architecture level, SECONDO offers tools for efficient and comfortable handling of nested lists and catalogs, a simplified interface to the underlying SHORE storage manager, a tool for efficient management of tuples with embedded large objects [DiG98], and an SOS-to-NestedList-Compiler. Algebra modules for standard and relational data types and operators as well as simple user interface clients have been implemented. [DiG99] gives a more detailed overview of the entire system.

In this paper, we have described the core part of SECONDO: the extensible query processor. Its main new aspects are the following:

- Other approaches lack a formal basis like SOS to describe a generic query plan algebra. This makes it impossible for them to give a clear algorithm for translating a query plan into an operator tree, as we do here.
- Functional abstraction is a well-defined concept in SOS. This leads to a very clean, simple and general treatment of parameter expressions of operators.
- *stream* is a built-in type constructor in SECONDO. Simply writing the keyword *stream* in the type mapping of an operator lets the query processor automatically set up calls of the evaluation function for this operator in stream mode. For this reason, SECONDO can handle uniformly streams of anything, not just tuples. Also, a query plan can mix freely stream and non-stream operators.
- SECONDO includes complete type checking, type mapping, and resolution of operator overloading.

For the time being, algebra support functions dealing with type expressions, like `TransformType`, have to be coded manually. In the future, an SOS specification compiler will create those functions automatically. Other future work will focus on the completion of an extensible rule-based query optimizer.

References

[Bato88] Batory, D.S., J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, GENESIS: An Extensible Database Management System. *IEEE Trans. on Software Engineering 14 (1988)*, 1711-1730.

[BaLW88] Batory, D.S., T.Y. Leung, and T.E.Wise, Implementation Concepts for an Extensible Data Model and Data Language. *ACM Trans. on Database Systems 13 (1988)*, 231-262.

[BeG95] Becker, L., and R.H. Güting, The GraphDB Algebra: Specification of Advanced Data Models with Second-Order Signature. FernUniversität Hagen, Praktische Informatik IV, Informatik-Report 183, 1995.

[Care86] Carey, M.J., D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, The Architecture of the EXODUS Extensible DBMS. In [Di86], 52-65.

[Care94] Carey, M.J., D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, M.J. Zwilling, Shoring Up Persistent Applications. Proc. ACM SIGMOD Conf. 1994, 383-394.

[CaD96] Carey, M.J., and D.J. DeWitt, Of Objects and Databases: A Decade of Turmoil. Proc. of the 22nd Intl. Conf. on Very Large Data Bases, Mumbai, India, 1996, 3-14.

[Di86] Dittrich, K.R., Proc. of the IEEE/ACM International Workshop on Object-Oriented Database Systems, Pacific Grove, California, 1986.

[DiG98] S. Dieker and R. H. Güting, Efficient Handling of Tuples with Embedded Large Objects. FernUniversität Hagen, Informatik-Report 236, 1998.

[DiG99] Dieker, S., and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. FernUniversität Hagen, Praktische Informatik IV, Informatik-Report 249, 1999.

[Gr94] Graefe, G., Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowledge and Data Engineering 6 (1994)*, 120-135.

[GrM93] Graefe, G., and W.J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search. Proc. of the 9th Intl. Conf. on Data Engineering, 1993, 209-218.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Data Bases, 1989, 33-44.

[Gü93] Güting, R.H., Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. Proc. ACM SIGMOD Conf. (Washington, 1993), 277-286.

[Haas89] Haas, L.M., J.C. Freytag, G.M. Lohman, and H. Pirahesh, Extensible Query Processing in Starburst. Proc. ACM SIGMOD Conf. 1989, 377-388.

[Haas90] Haas, L.M., W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowledge and Data Engineering 2 (1990)*, 143-160.

[Inf98] Informix Software, Inc. *Extending INFORMIX-Universal Server: Data Types, version 9.1*, 1998. Online version (http://www.informix.com).

[Pate97] Patel, J.M., J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N.E. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, D.J. DeWitt, J.F. Naughton, Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. Proc. ACM SIGMOD Conf. 1997, 336-347.

[RiC87] Richardson, J.E., and M.J. Carey, Programming Constructs for Database System Implementation in EXODUS. Proc. ACM SIGMOD Conf. 1987, 208-219.

[RiCS93] Richardson, J.E., M.J. Carey, and D.T. Schuh, The Design of the E Programming Language. *ACM Trans. on Programming Languages and Systems 15 (1993)*, 494-534.

[Sche90] Schek, H.J., H.B. Paul, M.H. Scholl, and G. Weikum, The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE Trans. on Knowledge and Data Engineering 2:1 (1990)*, 25-43.

[SeLR97] Seshadri, P., M. Livny, and R. Ramakrishnan, The Case for Enhanced Abstract Datatypes. Proc. of the 23rd Intl. Conf. on Very Large Data Bases, Athens 1996, 66-75.

[St96] Stonebraker, M., Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann Publishers, 1996.

[StRH90] Stonebraker, M., L.A. Rowe, and M. Hirohama, The Implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering 2 (1990)*, 125-142.

Appendix

A Definition of the Operator Tree

```
TYPE
  OpTree      = POINTER TO OpNode;
  OpNodeType  = (Object, IndirectObject, Operator);

  OpNode      = RECORD
     evaluable: BOOLEAN;
    CASE nodetype : OpNodeType OF
        Object:
          value: WORD;
      | IndirectObject:
          vector : ArgVectorPointer;
          argIndex : INTEGER;
      | Operator:
          algebraId: INTEGER;
          opFunId: INTEGER;
          noSons: INTEGER;
          sons: ARRAY [1..MAXARG] OF OpTree;
          isFun: BOOLEAN;
          funArgs: ArgVectorPointer;
          isStream: BOOLEAN;
          local: ADDRESS;
          received: BOOLEAN
    END (* CASE *)
  END; (* OpNode *)
```

B Structure of Procedure *annotate*

```
PROCEDURE annotate (expr    : ListExpr;
                VAR varnames: NameIndex.NameIndex;    (* in/out *)
                VAR vartable: varentryCTable;         (* in/out *)
                VAR defined : BOOLEAN                 (* in/out *))
                                                              : ListExpr;
VAR …
BEGIN
  IF IsEmpty(expr) THEN …     (* empty arg. list, case (xii), omitted *)
  ELSIF IsAtom(expr) THEN …   (* treatment of atoms, cases (i) - (vi), omitted *)
  ELSE (* expr is a nonempty list *)
    IF NOT (TypeOfSymbol(First(expr)) = fun) THEN (* not an abstraction *)

      (* first annotate recursively each element of this list: *)
      first := First(expr); rest := Rest(expr);
      list := OneElemList(annotate(first, varnames, vartable, defined)); lastElem := list;
      WHILE NOT IsEmpty(rest) DO
        lastElem := Append(lastElem, annotate(First(rest), varnames, vartable, defined));
        rest := Rest(rest)
      END;
      last := lastElem;       (* remember the last element to be able to
                                 append further arguments, see below *)

(* At this point, we may have for a given expr (+ 3 10) a list such as
       (((+ operator 1 6) none) ((3 ...) int) (( 10 ...) int))         *)

      first := First(list);               (* first = ((+ operator 1 6) none) *)
      IF ListLength(first) > 0 THEN
        first := First(first);            (* first = (+ operator 1 6) *)
        IF ListLength(first) >= 2 THEN
          CASE TypeOfSymbol(Second(first)) OF
            operator:                     (* operator application, case (vii) *)
              alId := IntValue(Third(first)); opId := IntValue(Fourth(first));

              (* extract the list of types into "typeList" *)
              rest := Rest(list);
              IF NOT IsEmpty(rest) THEN
                typeList := OneElemList(Second(First(rest)));
                rest := Rest(rest); lastElem := typeList
              END;
              WHILE NOT IsEmpty(rest) DO
                 lastElem := Append(lastElem, Second(First(rest))); rest := Rest(rest)
              END;
              (* apply the operator's type mapping: *)
              resultType := TransformType[alId]^[opId](typeList);

              (* use the operator's selection function to get the index
              (opFunId) of the evaluation function for this operator: *)
              opFunId := SelectFunction[alId]^[opId](typeList, opId);

              (* check whether the type mapping has requested to append
              further arguments: *)
              IF (ListLength(resultType) = 3) AND
                (TypeOfSymbol(First(resultType)) = APPEND)
              THEN
                lastElem := last; rest := Second(resultType);
                WHILE NOT IsEmpty(rest) DO
                  lastElem := Append(lastElem,
                  annotate(First(rest), varnames, vartable, defined));
                  rest := Rest(rest)
                END;
                resultType := Third(resultType)
              END;

              RETURN ThreeElemList(
                ThreeElemList(SymbolAtom("none"), SymbolAtom("applyop"), list),
              resultType, IntAtom(opFunId))

          | function: …          (* case (viii), omitted *)

          | abstraction …        (* case (ix), omitted *)

          ELSE …                 (* argument list, case (xi), omitted *)
          END (* CASE *)
        ELSE RETURN SymbolAtom ("exprerror") END
      ELSE RETURN SymbolAtom ("exprerror") END

    ELSE (* is an abstraction, case (x) *)
      RETURN annotate_function(expr, varnames, vartable, defined, 0,
        TheEmptyList(), TheEmptyList())
    END
  END (* nonempty list *)
END annotate;
```

C Structure of Procedure *subtree*

```
PROCEDURE subtree(expr : ListExpr) : OpTree;

VAR …
BEGIN
  CASE TypeOfSymbol(Second(First(expr))) OF
    constant, object:
      ALLOCATE(node, TSIZE(OpNode));
      WITH node^ DO evaluable := TRUE; nodetype := Object;
        valNo := IntValue(Third(First(expr))); value := Values[valNo]
      END; RETURN node;
  | operator:
      ALLOCATE(node, TSIZE(OpNode));
      WITH node^ DO evaluable := TRUE; nodetype := Operator;
        algebraId := IntValue(Third(First(expr)));
        opFunId := IntValue(Fourth(First(expr)));
        (* next three fields may be overwritten later *)
        noSons := 0; isFun := FALSE; funNo := 0; isStream := FALSE;
      END; RETURN node;
  | applyop:
      node := subtree(First(Third(First(expr))));
      WITH node^ DO evaluable := TRUE; opFunId := IntValue(Third(expr));
        noSons := 0; list := Rest(Third(First(expr)));
        WHILE NOT IsEmpty(list) DO INC(noSons);
          sons[noSons] := subtree(First(list)); list := Rest(list)
        END;
        IF NOT IsAtom(Second(expr)) AND
          (TypeOfSymbol(First(Second(expr))) = stream)
        THEN isStream := TRUE; evaluable := FALSE
        END
      END;
  | …                          (* other cases, omitted *)
END subtree;
```

D Procedure *eval*

```
PROCEDURE eval(tree   : OpTree;
           VAR result : WORD;        (* out *)
               message: INTEGER);
VAR i: INTEGER; status: INTEGER; arg: ArgVector;
BEGIN
  IF tree = NIL THEN Error ("eval called with tree = NIL!"); HALT;
  ELSE
    WITH tree^ DO
      CASE nodetype OF
        Object:         result:= value; RETURN
        IndirectObject: result := vector^[argIndex]; RETURN
        Operator:       (* First evaluate all subtrees that are not
                           functions or streams. Then call this
                           operator's evaluation procedure. *)
          FOR i := 1 TO noSons DO
            IF sons[i]^.evaluable THEN eval(sons[i], arg[i], message)
            ELSE arg[i] := sons[i]
            END
          END;
          status :=
                Execute[algebraId]^[opFunId](arg, result, message, local);
                (* Execute is an array of pointers to all operator evaluation
                  functions, maintained by the system frame. *)
          IF isStream THEN received := (status = YIELD)
          ELSIF status # 0 THEN  Error ("eval: operator failed"); HALT;
           END; RETURN
      END (* CASE *)
    END
  END
END eval;
```