An Algebra for List-Oriented Applications

Latha S. Colby*
Department of Computer Science
Indiana University
Bloomington, IN, 47405
colby@cs.indiana.edu

February 23, 1992

Abstract

Most data models and query languages, provide mechanisms for dealing with sets of objects. Many applications nowadays, however, are list-oriented, i.e., deal with collections or aggregates of objects in which their order is important. A formal model and an algebra for representing and manipulating list-oriented data are presented in this paper. We also give the criteria that were used in the design of the algebra and show how the algebra satisfies these criteria.

*The author was supported by a grant from the Indiana Corporation for Science and Technology.

1 Introduction

The relational data model [5] provides a simple yet powerful means of representing and querying data. However, the need for representing data with more complex structure and semantic information has resulted in a variety of complex data models. Models, like the nested relational model [12, 19], are simple extensions of the relational model, that allow a more natural representation of complex objects. Semantic and object-oriented models, such as, ER [4], SDM [9], FDM [18], and O_2 [3], on the other hand, were designed independent of the relational model and are equipped with various data modeling features that enable a variety of data applications to be modeled. However, none of these models is best suited for all applications. In particular, most of these models are inadequate for the following types of applications:

1. applications that are text-oriented, e.g., office documents, dictionaries, bibliographies and abstracts;
2. applications involving historical sequences of records, such as financial records and medical records;
3. experimental data that involve sequences or lists of data points.

There are two main reasons why most data models are not suitable for such applications. Firstly, the types of applications listed above all deal with sequences or lists of objects. Most models, however, are set-oriented. Models like the relational and the nested-relational models provide only sets as a grouping construct. Semantic data models often provide lists as a modeling construct. However, their query languages are not specialized enough to deal with list-oriented applications. Secondly, many of these applications require variable schema representation. For instance, a bibliography can have entries of many different formats. Traditional database models which require all instances of an entity (such as, all tuples in a relation) to correspond to the same schema, are unsuitable. Semantic models can often deal with variable schema by means of a construct called generalization. However, this construct most often comes bundled with other data modeling constructs such as, classification, specialization hierarchies and methods, with the result that these models are too complex for simple list-oriented applications.

A model that is designed for the types of applications mentioned earlier must be able to handle lists and variable schemas. Such a model would require a specialized query language

that provides list-oriented operations. Although sequencing is a feature of many database implementations, this feature is hidden from the modeling perspective. It is not our goal to abandon this important aspect of data independence. Rather the goal is to provide a modeling level construction for sequential information. The next section briefly reviews some of the data models and query languages that have been proposed for these applications. In Section 3, we give the description of a data model, the list-structure model, that is especially suited for the kinds of applications mentioned above. An algebra for the list-structure model is described in Section 4. In Section 5, we discuss some of the properties of the operators of the algebra and in Section 6, we discuss possible extensions to the model and the algebra.

2 Related work

Some of the early attempts at designing models for text-based applications were made in the area of office automation. A model for multimedia documents was proposed by Rabitti in [17]. The model is based on the idea of using context-free grammars to represent document structures. A schema defined in this model consists of a grammar and a set of restrictions. An instance of a schema is a tree (a derivation tree defined by the grammar) that satisfies the set of restrictions, if any. Operations for manipulating and retrieving documents are provided. Retrievals are specified by means of query filters. A query filter defines a portion of an instance tree and all trees that match the filter are retrieved.

Gonnet and Tompa [6] proposed a similar model based on grammars and a data-manipulation language. The language is oriented towards "text-dominated databases" and has the flavor of a programming language.

In [8], Gyssens, Paredaens and Van Gucht define an algebra and a calculus for a similar grammar-based model. While the languages are fairly simple and well defined, the operators of the languages are very primitive. Queries tend to become long and complicated when expressed in terms of such primitive operators.

A query language for manipulating text structures was proposed in [14] by Macleod. The model on which the language is based allows the hierarchical representation of text (using a grammar-based representation) as well as non-hierarchical linkages between objects (using references). The model and the language are specifically designed for text applications and

are not entirely suitable for other types of list-oriented applications. Moreover, the language does not provide enough flexibility to deal with arbitrary schema restructurings.

An extended nested relational model was proposed in [16], by Pistor and Traunmueller for dealing with both sets and lists. They extend the SQL language by providing some list-oriented operators and the equivalent of some nested relational algebra operators. The language and the data model that the language is based on are well suited for simple list-oriented applications, but not for complex applications involving variable schema.

An algebra for structured office documents is described in [7] by Güting, Zicari and Choy. The algebra, which is an extension of the relational algebra, is well-defined and has several features for dealing with sequences of objects. However, the data model upon which the algebra is defined uses a simplistic approach by combining aggregation and sequencing. It also does not provide variable schema representation.

3 A Model for List-Oriented Applications

3.1 Informal Description of the Model

In this section, we first outline the modeling features that are required in a data model that supports list-oriented applications like those that were mentioned in Section 1. We then give an informal description of the list-structure data model and show how the modeling requirements are satisfied by this model.

The data objects needed to represent the information in list-oriented applications are mainly of three types: (i) atomic, (ii) composite or aggregate, and (iii) collection. An atomic object is a simple data element, such as, an integer or a character. A composite object is one that is made up of several components. A collection of objects (of the same type) can be of one of the following categories – a sequence[1], an ordered set, an unordered set, or a bag. The elements in a sequence are all ordered and there can be duplicates among these elements. An unordered set (usually called a set) is a collection of distinct objects in which the order of the elements is irrelevant. In an ordered set the order of the elements is relevant. An ordered set can also be viewed as a sequence that has no duplicates. A bag is an unordered

[1]We use the term *list* to denote an ordered collection of objects that are either all of the same type or of different types; and the term *sequence* to denote an ordered collection of objects of the same type.

collection of elements in which duplicates are allowed (a set with duplicates).

In the case of most set-oriented applications, the order of the components of composite objects is irrelevant. However, for many list-oriented applications the order of the components is important. Hence, the data model that supports such applications should treat a composite object as being composed of a *list* of components rather than as a *set* of components.

Also, in many of these applications, a part of a schema (a sub-schema) may be repeated in several places in the schema description. For instance, a paragraph can occur in a chapter or in the preface of the book. In such cases, it would be better if the schema definition at a certain level referenced the sub-schema, rather than having the sub-schema definition repeated several times. This is, in fact, necessary in the case of recursive schema definitions. It must also be possible to associate different sub-schemas with a schema definition, thus allowing for variable schema representation.

The data model that we are about to describe is designed to meet the above requirements. However, for the sake of simplicity (particularly in the query language description), the model described here does not allow for sets (either unordered or ordered) or bags. We show in Section 6 how the model and the query language can be extended to deal with such types of collections, as well.

Given the above requirements for a data model, it seems natural to choose a hierarchical representation for the instances of the model. It would also be natural to describe the schemas for such hierarchical instances in terms of a set of productions, similar to the kinds of productions one would use to define a context free grammar. In the list-structure model a schema consists of a start symbol and a set of productions. A value corresponding to a scheme (known as a structure over the scheme) is a tree that corresponds to the productions in the schema and whose root node is the start symbol of the scheme[2]. Figure 1 is an example of a bibliography list-structure scheme and Figures 2 and 3 are examples of structures over that scheme.

For the sake of simplicity, we allow only certain kinds of productions in the schemes. A leaf production is one that is used to describe atomic objects. The right-hand side of such a production is either a primitive type name or a constant. In the book-collection scheme,

[2]The model deals with objects in which both the hierarchical structure of the objects and the order of the different components or elements of the objects is relevant – hence the name *list-structure* model.

$$
[\![\texttt{BOOKS},\quad
\left\{
\begin{array}{ll}
\texttt{BOOKS} \longrightarrow \texttt{BOOK*} & \texttt{MI} \longrightarrow \texttt{Char} \\
\texttt{BOOK} \longrightarrow \texttt{YEAR AUTHORS TITLE} & \texttt{L-NAME} \longrightarrow \texttt{String} \\
\texttt{BOOK} \longrightarrow \texttt{YEAR AUTHORS VOL TITLE} & \texttt{VOL} \longrightarrow \texttt{TITLE NO.} \\
\texttt{YEAR} \longrightarrow \texttt{Integer} & \texttt{VOL} \longrightarrow \texttt{NO.} \\
\texttt{AUTHORS} \longrightarrow \texttt{A*} & \texttt{NO.} \longrightarrow \texttt{Integer} \\
\texttt{A} \longrightarrow \texttt{F-NAME MI L-NAME} & \texttt{NO.} \longrightarrow \texttt{Char} \\
\texttt{A} \longrightarrow \texttt{F-NAME L-NAME} & \texttt{TITLE} \longrightarrow \texttt{WORD*} \\
\texttt{F-NAME} \longrightarrow \texttt{String} & \texttt{WORD} \longrightarrow \texttt{String}
\end{array}
\right\}
]\!]
$$

Figure 1: A book-collection list-structure scheme.

WORD $\longrightarrow$ String is an example of a leaf production. An aggregate production is used to specify composite objects, e.g., BOOK $\longrightarrow$ YEAR AUTHORS TITLE. A list production is used to specify objects that contain a list of similar objects, e.g., BOOKS $\longrightarrow$ BOOK*. Variable schema can be easily represented since two or more productions can have the same left-hand side but different right-hand sides. Although our example does not contain a recursive definition, one can see how recursive schema definitions can be easily represented. Also, since we are using productions to describe a schema, a sub-schema definition does not have to be repeated every time it is used in a schema declaration. For instance, in the example schema, TITLE appears in the productions that describe the BOOK part of the schema, as well as in the productions that describe the VOL part. However, one does not have to repeat the description (i.e., the productions that describe TITLE) for every occurrence of TITLE in the right-hand side of a production. Thus, all of the requirements that were outlined earlier, (except the requirement that the model must allow for sets and bags), are satisfied by the list-structure model.

The list-structure model is essentially the same as those of [6] and [8]. The main difference is in the kinds of productions that are allowed. For instance, both [6] and [8] allow productions that can have a combination of constants and variables on the right hand side, and [8] does not allow list productions. These variations do not affect the type of information that can be represented in these models. As we shall see, our choice of production types allows for simplicity in the definitions of the operators of the query language.

The list-structure model can also be viewed as a variation of the Format model [11]. The objects that can be represented in this model also have hierarchical structures that correspond

to types that are built from three type constructors – aggregate, set and union – and a set of base types. The union type construct allows several different types to be associated with one type which is in some sense equivalent to allowing different productions in the list-structure model to have the same left hand side. The main difference between the Format model and the list-structure model, then is that the grouping construct in the former is a set-construct and a list-construct in the latter.

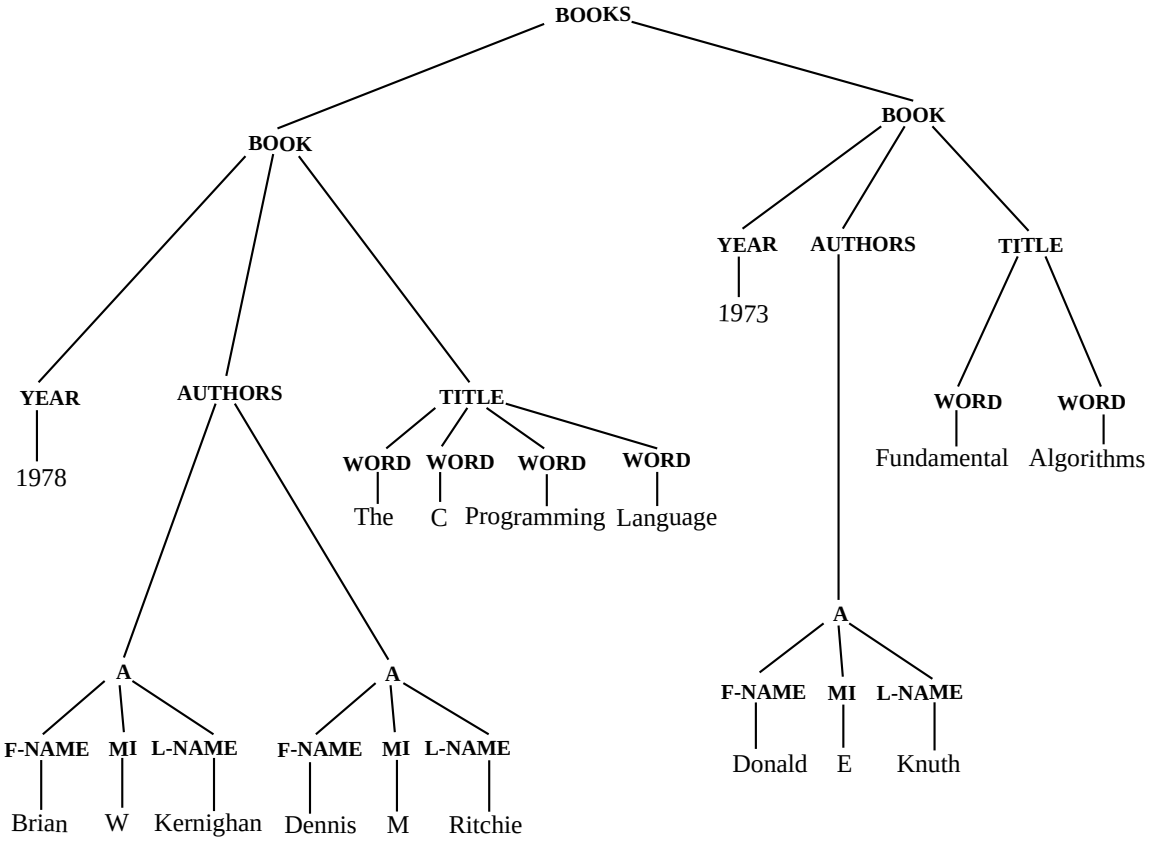


Figure 2: A structure over the book-collection scheme.

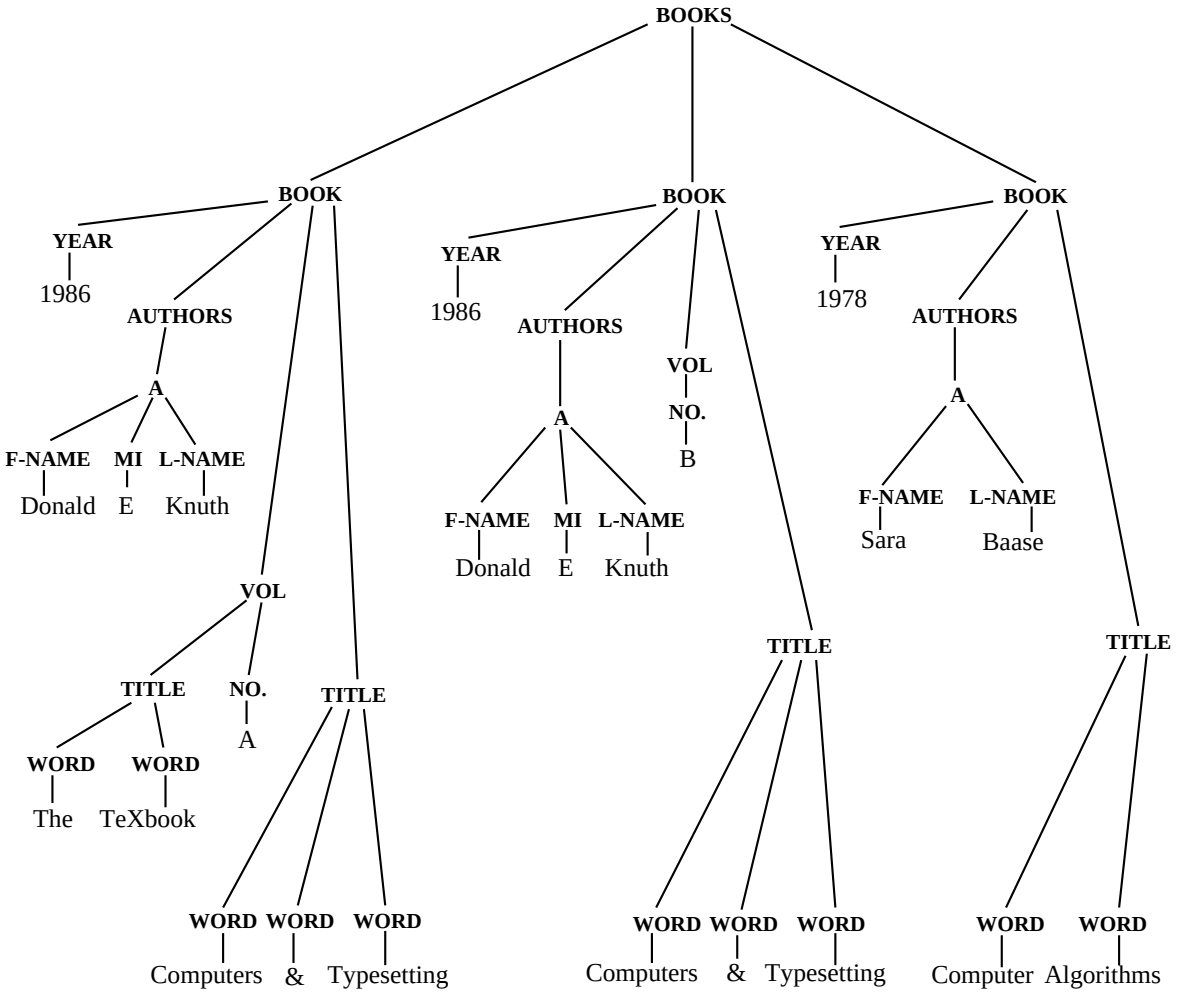


Figure 3: An alternate structure over the book-collection scheme.

3.2 Formal Definitions of the Model

Let $\mathcal{V}$ be the universal set of all variables. Let , be the set of basic types, such as, Integer, Real, etc., and for each type $\tau \in$, , let $Dom(\tau)$ denote the set of all the elements or constants in the domain of that type. Also, let $\mathcal{T}$ be the set of all the constants, i.e., $\mathcal{T} = \bigcup_{\tau \in \Gamma} Dom(\tau)$. The sets $\mathcal{V}$, , and $\mathcal{T}$ are pairwise distinct.

Definition 3.1 A *production* p is of the form $A \longrightarrow \beta$, where $A \in \mathcal{V}$, and β is of one of the following forms:

(i) $\beta = B_1...B_k$, $k \geq 1$, where each $B_i \in \mathcal{V}$, and the B_i's are distinct.
(ii) $\beta = B^*$, where $B \in \mathcal{V}$.
(iii) $\beta = c$, where $c \in \mathcal{T}$.
(iv) $\beta = \tau$, where $\tau \in$, . □

If p is a production, then $head(p)$ refers to the variable on the left hand side of the '$\longrightarrow$' in p and $tail(p)$ refers to the right hand side. For example, if $p = A \longrightarrow B_1...B_k$, then $head(p) = A$ and $tail(p) = B_1...B_k$. We use the symbol '$\in$' to denote the occurrence of a variable or constant or type in the tail of a production. For instance, 'B occurs in $tail(p)$' can be written as $B \in tail(p)$. If $tail(p)$ is of the form B^*, we say $B \in tail(p)$, and not $B^* \in tail(p)$. We use words or letters in upper case to denote variables.

Definition 3.2 A *list-structure scheme* $\mathcal{G}$, consists of a variable S, called the *start symbol* of $\mathcal{G}$, and a finite set of productions P, where $S = head(p)$ for some $p \in P$ if P is non-empty. We denote such a scheme by the ordered pair $[\![S, P]\!]$. $var(\mathcal{G})$ denotes the set consisting of S and all the variables that occur in the set of productions P. □

A production p is called

(i) an aggregate production if $tail(p) = B_1...B_k$,

(ii) a list production if $tail(p) = B^*$, and

(iii) a leaf production if $tail(p) = b$, where $b \in \mathcal{T}$ or $b \in$, .

In the scheme s, shown in Figure 4, the production $S \longrightarrow A\ B$ is an example of an aggregate production, $B \longrightarrow D^*$ is a list production and $A \longrightarrow a_1$ is a leaf production.

$$\left[\!\left[S, \left\{ \begin{array}{l} S \longrightarrow A\ B,\ S \longrightarrow A\ B\ C,\ A \longrightarrow a_1, \\ A \longrightarrow a_2,\ B \longrightarrow D^*,\ C \longrightarrow A\ S, \\ C \longrightarrow c_1,\ C \longrightarrow c_2,\ D \longrightarrow d_1, \\ D \longrightarrow d_2,\ D \longrightarrow E,\ D \longrightarrow E\ F, \\ E \longrightarrow e_1,\ E \longrightarrow e_2,\ F \longrightarrow f_1,\ F \longrightarrow f_2, \end{array} \right\} \right]\!\right]$$

Figure 4: The list-structure scheme s.

A production p is *non-terminating* in P if it is not a leaf production or a list production and at least one of the variables in its tail, say A, is such that (a) there is no production in P (other than p) whose head is A, or (b) every production whose head is A is non-terminating in $P - \{p\}$. If the scheme s contained the productions $A \longrightarrow B\ C\ G$ and $G \longrightarrow H$, then both of these productions would be non-terminating (in the set of all the productions of the scheme).

A variable A is *reachable* from a variable B in a set of productions P, if (a) $A = B$, or (b) there is a path from B to A in P, i.e., if there is a sequence of productions $p_1, ..., p_n$ in P such that $head(p_1) = B$, $A \in tail(p_n)$, and $head(p_{i+1}) \in tail(p_i)\ \forall i, 1 \leq i < n$. A production p in P is *reachable* from a variable B if $head(p)$ is reachable from B. If the scheme s contained the production $H \longrightarrow h_1$, it would be a production that is not reachable from the start symbol S.

A production p is *redundant* in P, if it is a leaf production of the form $A \longrightarrow b$, where b is a constant, and P also contains the production $A \longrightarrow \tau$, where τ is a type and $b \in Dom(\tau)$.

A production p, of a scheme $\mathcal{G} = [\![S, P]\!]$, is a *useless* production if it can never occur in a structure over the scheme, i.e., if it is non-terminating in P, or if it is not reachable from the start symbol S. A scheme $\mathcal{G}$ is said to be *normalized* if it has no useless or redundant productions. The function *normalize* defined below, takes a scheme as input and produces a normalized scheme. Note that for a normalized scheme $\mathcal{G}$, $normalize(\mathcal{G}) = \mathcal{G}$. The schemes in both Figure 4 and Figure 1 are normalized schemes.

Let $\mathcal{G} = [\![S, P]\!]$. The function $normalize(\mathcal{G})$ first removes from P all non-terminating productions and then all productions that are not reachable from the start symbol. It then removes all redundant productions.

Definition 3.3 Let $\mathcal{G} = [\![S, P]\!]$ be a scheme. Then,

$normalize(\mathcal{G}) = \mathcal{G}'$, where $\mathcal{G}' = [\![S, P']\!]$ is determined as follows.

$$\begin{aligned} P' &= P'' - \{p \mid (p \in P'') \wedge (tail(p) \in \mathcal{T}) \\ &\quad \wedge (\exists q \in P'' \mid (head(p) = head(q)) \wedge \\ &\quad\quad (tail(q) \in ,) \wedge (tail(p) \in Dom(tail(q))))\}, \text{ where,} \\ P'' &= P''' - \{p \mid (p \in P''') \wedge (reachable(head(p), S, P''') = false)\} \\ &\quad \text{where, } P''' = P - \{p \mid (p \in P) \wedge (non\text{–}terminating(p, P) = true)\} \end{aligned}$$

If $\mathcal{G} = [\![S, P]\!]$ is a list-structure scheme and p is a production in P, then

$$\begin{aligned} non\text{–}terminating(p, P) &= true, \text{ if } (tail(p) = B_1 ... B_m) \wedge \\ &\quad\quad (\exists B \in tail(p) \mid (\forall q \in P - \{p\}\ (B \neq head(q)) \\ &\quad\quad\quad \vee\ (non\text{–}terminating(q, P - \{p\}) = true))), \\ &= false, \text{ otherwise} \end{aligned}$$

If P is a set of productions and A and B are variables, then

$$\begin{aligned} reachable(A, B, P) &= true, \text{ if } (A = B) \\ &\quad \vee (\exists p \in P \mid (A \in tail(p)) \wedge (B = head(p))) \\ &\quad \vee (\exists p \in P \mid (B = head(p)) \wedge ((\exists B_i \in tail(p)) \\ &\quad\quad \mid (reachable(A, B_i, P - \{p\}) = true))) \\ &= false, \text{ otherwise} \end{aligned}$$

□

Having defined list-structure schemes we are now ready to define *structures*. A structure over a list-structure scheme is a tree (a derivation tree in the usual grammar sense) and is represented by a pair, where the first element denotes the root of the tree and the second its subtrees.

Definition 3.4 t is a finite structure over a scheme $\mathcal{G} = [\![S, P]\!]$, if

1. $t = []$, the empty structure over $\mathcal{G}$,
2. $t = [S, l]$, where S, the start symbol of $\mathcal{G}$, is the root of the structure and l is a list of subtrees, such that
   (a) $l = (t_1, ..., t_k)$, $k \geq 1$, and for some $p \in P$, $head(p) = S$ and $tail(p) = B_1 ... B_k$, and each t_i is a non-empty structure over $\mathcal{G}_i$, where $\mathcal{G}_i = normalize([\![B_i, P]\!])$, or

(b) $l = (t_1, ..., t_m)$, $m \geq 0$, and for some $p \in P$, $head(p) = S$ and $tail(p) = B^*$, and each t_i is a non-empty structure over $\mathcal{G}'$, where $\mathcal{G}' = normalize([\![B, P]\!])$, or

3. $t = [S, b]$, where S is the root of the structure and b is a constant, and for some $p \in P$, $head(p) = S$ and (i) $tail(p) = b$ or (ii) $tail(p) = \tau$, where $\tau \in$, and $b \in Dom(\tau)$. □

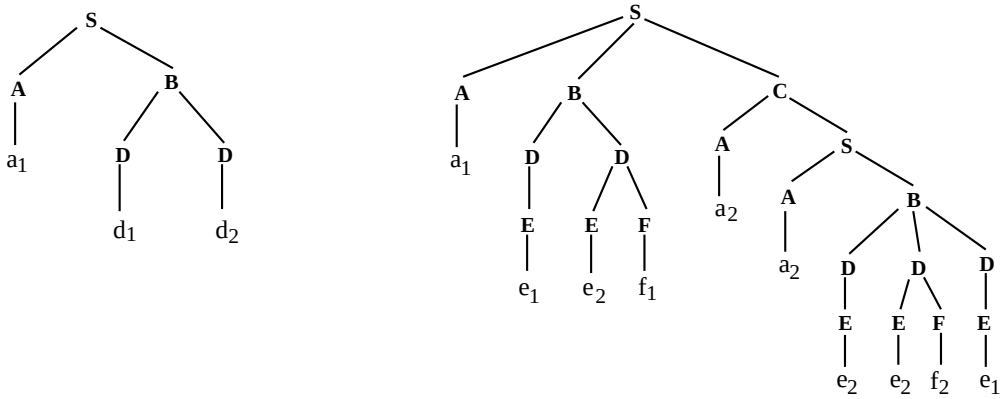


Figure 5: Example structures over the scheme s.

We will often refer to structures as trees. Figure 5 shows examples of structures over the scheme s, in Figure 4. If t is a structure over $\mathcal{G}$, and $t = [A, l]$, then $root(t) = A$, and $subtrees(t) = l$. If $subtrees(t) = b$, where $b \in \mathcal{T}$, or if $subtrees(t) = ()$, then $children(t) = subtrees(t)$. Otherwise, $children(t) = (root(t_1), ..., root(t_k))$ where $(t_1, ..., t_k) = subtrees(t)$. If $children(t) = (B_1, ..., B_k)$, where the B_i's are distinct, then $t[B_i]$ denotes the subtree of t with root B_i. We also refer to the B_i's, in this case, as attributes of t. $size(subtrees(t))$ denotes the number of subtrees of t.

Definition 3.5 A *list-structure instance* r is a pair $\langle t, \mathcal{G} \rangle$, where t is a structure over the scheme $\mathcal{G}$. $struct(r)$ refers to t, the first component of r, and $scheme(r)$ refers to $\mathcal{G}$, the second component. □

We will sometimes refer to the structure part of an instance as its value. For any scheme $\mathcal{G}$, the set of all the structures that can be defined over $\mathcal{G}$ is the same as that of $normalize(\mathcal{G})$. The proof is by induction on the number of productions in $\mathcal{G}$ and is very similar to the one given in [10] and is hence omitted here. In the rest of this paper we will assume that all structures are defined over normalized schemes.

4 An Algebra for the List-Structure Model

In this section we describe an algebra for the list-structure model. While the main purpose of any query language is to retrieve and manipulate databases, there are often specific objectives or criteria that influence the design of the language. For example, for some applications it might be essential that the language be Turing expressive. For other applications, Turing expressiveness might not be as important an objective as the ability to express a given class of queries succinctly in the language. In Section 4.1, we describe the set of criteria that were used in the overall design of the algebra for the list-structure model. Section 4.2 contains a brief, informal description of the algebra, while Section 4.3 contains the detailed formal description. In Section 5 we explain how the algebra meets the design criteria.

4.1 Criteria for a List-Structure Query Language

(i) *The language should be well-suited for list-oriented applications.*

A model that is designed to suit a specific class of applications is useful only when its associated query language is also well-suited for those applications. Hence, the language must provide list-oriented functions, such as searching and updating based on pattern match and list-position, and ordering and sorting elements in a list.

(ii) *The operators must be simple enough to understand and implement but powerful enough to enable queries to be expressed succinctly.*

The motivation for this should be fairly obvious. A language is more useful when the syntax and semantics of its components are simple enough to be easily understandable and implementable. On the other hand, if the operators of the language are too primitive, expressions representing even the simplest queries can become long and complicated.

(iii) *The number of basic operators required to express most reasonable queries must be kept to a minimum.*

A language consisting of a small set of basic operators is again easier to work with and to implement. However, the set of operators must be fairly expressive to be able to handle most queries. There is no agreement about what "fairly expressive" means and some researchers view Turing expressiveness as the standard for expressiveness. In the

design of this algebra greater emphasis was placed on the simplicity and ease of use of the language than on Turing expressiveness. The operators chosen for the algebra are those that are needed in typical queries or that can (in conjunction with other operators) be used to express operators that are not often used. For instance, we do not provide binary operators like union, difference, intersection, and join since they can be expressed in terms of other operators.

(iv) *The language must satisfy the closure property. In other words, the type of the objects returned by the operators must be the same as the type of the operands.*

The fact that our language is an algebra implies that it satisfies the closure property. All 'objects' that are manipulated by the algebraic operators are list-structure instances. Each operator takes one or two instances and returns an object which is again a list-structure instance. This closure property makes it easier to compose queries since one is always dealing with objects of a single type. It also increases the scope for query optimization since the order of the operators can be changed (whenever possible).

(v) *Except for operators that are designed specifically to change the scheme of an instance depending upon its value, all other operators, when applied to an instance, must return an instance whose scheme is independent of the value of the input instance.*

If the scheme of the result of any operation depends only on the scheme of the input instance, then users can compose queries consisting of several operators without having to examine the scheme of the result of each individual operator (assuming that the scheme of the input instance is known to the user). For example, suppose that $\langle t, \mathcal{G} \rangle$ is an instance and that $\mathcal{G}$ contains the production, say p, where $p = A \longrightarrow BCD$. Now let us suppose that we want to delete the D-subtree from all the trees in t that correspond to the production p and whose B-subtree has the value b_1. Should the scheme of the result, $\langle t', \mathcal{G}' \rangle$, contain p? Now, t' may or may not have trees that correspond to p. We could decide if p should belong in $\mathcal{G}'$ after examining t'. But suppose that we have a query composed of two operators, where the first one is the Delete that we described above and the second is an Insert that inserts a tree as a subtree in each tree that corresponds to the production $A \longrightarrow BCD$. If $\mathcal{G}'$ didn't contain p, the second operation would be illegal. Thus one would have to examine the scheme of the result of the first operation before applying the second. On the other hand, if we decided to let $\mathcal{G}'$ always contain p, the

second operation would be legal although it may not find any trees that correspond to p in its input instance.

4.2 Informal Description of the Algebra

The algebra has a set of thirteen value-changing operators and a value-preserving operator. The value-changing operators manipulate the value part, i.e., the structure, of instances. The corresponding schemes of the instances may or may not be affected. The value-preserving operator changes the scheme of an instance without affecting the value part of an instance. The following is a listing of the operators.

Value-Changing					Value-Preserving
Retrieve	Delete	Replace	UnGroup	Apply	Change-Scheme
Find	Substitute	Reorder	Number		
Insert	Rename	Group	Sort		

Any query language must have a means of searching for data according to some given condition. In most languages the searching mechanism is built into operators that perform some other action after the searching (such as, retrieving the results of the search, as in a selection operator). In the list-structure model, since the concept of ordering is important, one can expect queries that involve selecting, as well as inserting and deleting elements, depending upon positions, patterns, and values of the elements. In other words, one can expect to have to search for elements satisfying a given condition, not only before retrieving those elements, but also before inserting or deleting. It would, hence, be convenient to have a separate operator that looks, within an instance, for trees that satisfy a given condition and marks those trees. The *Find* operator is used for searching and marking trees in an instance. Trees that satisfy the search condition are marked by the introduction of a new node as the child of the root. The *Retrieve* operator extracts trees that have a given root node. The result is a tree whose subtrees are the trees that have been extracted.

Inserting and deleting elements in a tree are performed by the *Insert* and *Delete* operators. These operators have some limited searching mechanism built into them, since one must be able to specify the location where an element is to be inserted or the value of the elements that are to be deleted. However, one might have to use the *Find* operator first for more

complicated searches. The *Substitute* operator can often be used to perform the action of an *Insert* followed by a *Delete*.

The *Rename* operator is used for renaming nodes in a tree. It is often used to restrict the application of other operators to trees that correspond only to a given production. It can also be used to make the elements of a list production correspond to an aggregate production and vice-versa. The *Replace* operator is used to add or remove internal nodes from a tree. It is often used to remove internal nodes introduced by other operators, such as, *Find*. The *Reorder* operator is used to rearrange trees that correspond to aggregate productions.

The operator *Group* is used to group elements of trees that correspond to list productions. This is very similar to the *Nest* operation in the nested relational algebra. The *UnGroup* operator does the opposite of the *Group* operator and is similar in function to the *Unnest* operator of the nested algebra.

Although the *Find* operator can be used to locate elements in a given position, it is sometimes necessary to be able to number the elements in a list, explicitly. The *Number* operator numbers the subtrees of a list production (assuming that the subtrees all correspond to aggregate productions) by adding an attribute to each subtree such that the value of the attribute is the position of the subtree in the list. The *Sort* operator orders the elements of a given list production. This operator is important for the kinds of applications supported by this model, since the concept of ordering is crucial to these applications.

The *Apply* operator is used to perform actions on a local level. Any of the operators listed so far (or any sequence of these) can be supplied as an argument to the *Apply* operator, which will then 'apply' this operator (or sequence) to trees that satisfy a given aggregate production. The result of the application on each tree is stored as an additional attribute within the same tree. This is similar to the *extend* (λ) operator of [7].

The operators listed so far are the value-changing operators. *Change–Scheme* is a value-preserving operator that is used to modify the scheme of an instance without affecting the structure by adding new productions or deleting extraneous ones. For instance, if p_1 is a production in a scheme $\mathcal{G}$ and t is a structure over $\mathcal{G}$, then if there are no trees in t that correspond to p_1, one can delete p_1 from the set of productions in $\mathcal{G}$ without affecting t. The value-preserving operator is not generally needed for querying, but serves as a data-manipulation function that is used for schema modification.

The list-structure algebra is a query language and not a data manipulation language in the sense that if an operation Q is performed on an instance i, then the result of the operation is a new instance i' such that $i' = Q(i)$. In other words, the original instance i is not affected as a result of the operation which means that updates cannot be made to the instances. However, the language has been designed in such a way that it can be easily extended and used as a data manipulation language as well. For instance, it should be possible to have a system in which there is a query mode and an update mode (similar to the concept of modes in the context of the GOOD model [2]). In the query mode the result of an operation Q on an instance i will be a different instance i', whereas in the update mode, i will be replaced by the result of $Q(i)$.

Some of the operators of the algebra, like *Insert*, and *Change–Scheme* are not likely to be used very often for querying purposes but they were still provided in the language so that they may be used for updates. Many of the operators of the language, however, can be expected to be used for both querying and updating. For example, the query "*Display the* `TITLE` *and* `AUTHORS` *of all the books in the instance* d_1*, shown in Figure 2.*" would involve the use of the *Delete* operator to project out the `TITLE` and `AUTHORS` attributes of the books. The same operator can also be used to perform the update "*Remove all books published before 1970 from* d_1". We would, of course, need to add additional data manipulation operators, such as *Create–Scheme* and *Create–Instance* to make the language a full-fledged data-manipulation language.

4.3 Description of the Algebra

This section provides a complete description of the algebra including the formal definitions for the operators. In all of the definitions we will use the following notation. If $\langle t, \mathcal{G} \rangle$ is a list-structure instance then $(t_1, ..., t_k)$ are the subtrees of t, unless indicated otherwise. If $subtrees(t) \notin \mathcal{T}$, then each t_i is a structure over $\mathcal{G}_i$, where $\mathcal{G}_i = normalize([\![root(t_i), P]\!])$. Similarly, if $\langle t_1, \mathcal{G}_1 \rangle$ is an instance, then the t_{1i}'s denote the subtrees of t_1 and the $\mathcal{G}_{1i}$'s denote the corresponding schemes. Also, if $\mathcal{G}$ is a scheme then S and P denote the start symbol and the set of productions of $\mathcal{G}$.

The symbols $\subset$, $\subseteq$, $\supset$, and $\supseteq$, denote list comparisons. For instance, $l_1 \subseteq l_2$ should be read as 'l_1 is a sublist of l_2'. For example, if $l_1 = (a_1, ..., a_m)$ and $l_2 = (b_1, ..., b_n)$, then $l_1 \subseteq l_2$

if l_2 has the sequence $b_k, ..., b_{k+m-1}$ such that $\forall a_i$, $a_i = b_{k+i-1}$. The symbols $\sqsubset$, $\sqsubseteq$, $\sqsupset$, and $\sqsupseteq$ also denote list comparisons. However, these operations check if the elements of one list are contained in another in the same order. So, if $l_1 = (a_1, ..., a_m)$ and $l_2 = (b_1, ..., b_n)$, then $l_1 \sqsubseteq l_2$ if $(\forall a_i \ (a_i \in l_2))$ and $(\forall a_i, a_j \ (a_i = b_p) \wedge (a_j = b_q) \wedge (i < j) \Rightarrow (p < q))$.

We first define some functions that will be used in the definitions. These are not part of the algebra and are used only to make the definitions of the operators succinct and readable. In order to make these functions easily distinguishable from the operators of the algebra, we will denote the names of the operators beginning with upper case letters, and function names beginning with lower case letters. We also define some data structures that are used for pattern matching. Most definitions are written in the form shown below and we assume that condition i is tested only if conditions 1 through $i-1$ fail.

$$\begin{aligned} f(x) &= y_1, \text{ if condition 1,} \\ &= y_2, \text{ if condition 2,} \\ &\ \ \vdots \\ &= y_n, \text{ otherwise.} \end{aligned}$$

$\diamond$ The function *concat* takes zero or more lists as arguments and returns a single list which is the concatenation of the lists. For instance, $concat(l_1, l_2, l_3)$, where $l_1 = (a, b, c)$, $l_2 = ()$ and $l_3 = (c, d)$ is (a, b, c, c, d), $concat() = ()$ and $concat((), l_1) = l_1$. $\diamond$

$\diamond$ The function *append* takes an element and a list as arguments and appends the element to (the left of) the list, e.g., $append(t, l)$, where $t = a$ and $l = (b, c, d)$ is (a, b, c, d). $\diamond$

$\diamond$ *has–leaf* takes as input a non-empty structure and returns true or false depending upon whether the child of the tree is a constant or not.

$$\begin{aligned} has\text{–}leaf(t) &= true, \text{ if } (subtrees(t) = b) \wedge (b \in \mathcal{T}), \\ &= false, \text{ otherwise.} \end{aligned}$$

$\diamond$

$\diamond$ The function *substitute* takes two variables and a production as arguments and replaces all occurrences of the first variable in the production by the second variable. For example, $substitute(A, B, A \longrightarrow AC)$ will return $B \longrightarrow BC$. $\diamond$

$\diamond$ The function $satisfies$ determines if a structure corresponds to a production. If t is a structure and p is a production, then $satisfies(t,p)$ is defined as follows.

$$\begin{aligned} satisfies(t,p) \;=\; & true, \text{ if } (t \neq []) \wedge (root(t) = head(p)) \wedge \\ & (((has\text{-}leaf(t) = false) \wedge \\ & \quad ((tail(p) = B_1 ... B_k) \wedge (children(t) = (B_1, ..., B_k))) \\ & \quad \vee ((tail(p) = B^*) \wedge (\forall B_i \in children(t)\ (B_i = B)))) \\ & \vee ((subtrees(t) = b) \wedge (tail(p) = b)) \\ & \vee ((subtrees(t) = b) \wedge (tail(p) = \tau) \wedge (b \in Dom(\tau)))) \\ =\; & false, \text{ otherwise.} \end{aligned}$$

In the structure, x_1, shown in Figure 5, $satisfies(x_1,p)$ is true for $p = S \longrightarrow A\ B$, while $satisfies(x_1,p)$ is false for all other productions. In some cases, a structure can correspond to more than one production. If the scheme in Figure 4 contained the production $B \longrightarrow C^*$ in addition to all the other productions, then the tree $[B,()]$ would correspond to both $B \longrightarrow D^*$ and $B \longrightarrow C^*$. Section 6.3 contains some ideas on how the model may be modified to ensure that a structure always corresponds to a unique production. $\diamond$

We now define a *struct-list* over a variable and functions, *str–list* and *compatible*. A struct-list over a variable A essentially denotes a structure tree over A without the interior node labels. The function *str–list* returns the struct-list corresponding to a given structure. The function *compatible* determines whether two variables can have the same struct-list or not.

Definition 4.1 A *struct-list* over a variable A of scheme $\mathcal{G}$ is defined as follows. β is a struct-list over A, if

1. β is a constant b and for some $p \in P$, $head(p) = A$ and either $tail(p) = b$ or $tail(p) = \tau$, where $\tau \in$, , and $b \in Dom(\tau)$, or
2. $\beta = (\beta_1, ..., \beta_k)$, $k \geq 1$, and for some $p \in P$, $head(p) = A$, $tail(p) = B_1 ... B_k$ and each β_i is a struct-list over B_i, or
3. $\beta = (\beta_1, ..., \beta_m)$, $m \geq 0$, and for some $p \in P$, $head(p) = A$, $tail(p) = B^*$ and each β_i is a struct-list over B. $\square$

$\diamond$ The function *str–list* takes as input a non-empty structure and returns the struct-list corresponding to that structure. For example, $str\text{–}list(x_1) = (a_1, (d_1, d_2))$ (x_1 is shown in Figure 5).

$$
\begin{aligned}
\mathit{str\text{–}list}(t) &= \mathit{subtrees}(t), \text{ if } \mathit{has\text{–}leaf}(t), \\
&= (t'_1, ..., t'_k), \text{ otherwise, where } t'_i = \mathit{str\text{–}list}(t_i) \text{ and } (t_1, ..., t_k) = \mathit{subtrees}(t).
\end{aligned}
$$

◇

◇ Two variables A and B are said to be *compatible* if it is possible to have the same struct-list over both variables. The function *compatible* takes two variables and a set of productions as input and returns true or false depending on whether the variables are compatible in the set of productions or not.

$$
\begin{aligned}
\mathit{compatible}(A, B, P) &= \mathit{true}, \text{ if } \exists p_1, p_2 \in P \mid (\mathit{head}(p_1) = A) \wedge (\mathit{head}(p_2) = B) \\
&\quad \wedge((\mathit{tail}(p_1) = \mathit{tail}(p_2)) \\
&\quad\quad \vee((\mathit{tail}(p_1) \in \mathcal{T}) \wedge (\mathit{tail}(p_2) \in ,) \wedge (\mathit{tail}(p_1) \in \mathit{Dom}(\mathit{tail}(p_2)))) \\
&\quad\quad \vee((\mathit{tail}(p_2) \in \mathcal{T}) \wedge (\mathit{tail}(p_1) \in ,) \wedge (\mathit{tail}(p_2) \in \mathit{Dom}(\mathit{tail}(p_1)))) \\
&\quad\quad \vee((\mathit{tail}(p_1) = B_1 ... B_m) \wedge (\mathit{tail}(p_2) = C_1 ... C_m) \\
&\quad\quad\quad \wedge(\mathit{compatible}(B_i, C_i, P - \{p_1, p_2\}), \forall i, 1 \leq i \leq m)) \\
&\quad\quad \vee((\mathit{tail}(p_1) = C^*) \wedge (\mathit{tail}(p_2) = D^*))) \\
&= \mathit{false}, \text{ otherwise.}
\end{aligned}
$$

◇

Note that A and B are compatible if they are the heads of two list productions, regardless of whether the variables in the tails are compatible or not. This is because the empty list () can be a structure over both productions. The concept of a struct-list enables comparisons between structures that do not take the labeling of the interior nodes into account. It would also be useful to be able to ignore the entire structure altogether and examine only the leaf nodes of a structure.

◇ The function *leaves* takes a non-empty structure as input and returns the list of all the constants in the leaf nodes of the structure. For example, $\mathit{leaves}(x_1) = (a_1, d_1, d_2)$.

$$
\begin{aligned}
\mathit{leaves}(t) &= (\mathit{subtrees}(t)), \text{ if } \mathit{has\text{–}leaf}(t) \\
&= \mathit{concat}(t'_1, ..., t'_k), \text{ otherwise, where } t'_i = \mathit{leaves}(t_i).
\end{aligned}
$$

◇

In comparisons involving the *leaves* of a structure any list of constants can be used in the comparison operation (see Definition 4.4). Note that unlike in the case of a *struct-list* we do not have the concept of a *leaf-list* since determining if a list is a leaf-list over a variable can be a time consuming operation.

Finally, we describe *structure patterns* and *value patterns*. These are used in operations that involve pattern matching. We also define two functions, *match* and *matchlist* that determine when these patterns match a structure and a list of structures, respectively.

Definition 4.2 Let $\mathcal{G} = [\![S, P]\!]$ be a scheme. Then, β is a structure pattern over a variable A of scheme $\mathcal{G}$, if it is of one of the following forms.

1. $\beta = [A, *]$, and for some $p \in P$, $head(p) = A$.
2. $\beta = [A, -]$, and for some $p \in P$, $head(p) = A$, and $tail(p) \in \mathcal{T}$ or $tail(p) \in$, .
3. $\beta = [A, b]$, and for some $p \in P$, $head(p) = A$, and $tail(p) = b$ or $tail(p) = \tau$, where $\tau \in$, and $b \in Dom(\tau)$.
4. $\beta = [A, (\beta_1, ..., \beta_m)]$, $m \geq 1$, and for some $p \in P$, $head(p) = A$, $tail(p) = B_1 ... B_k$, and
   (a) each β_i is either a $*$, or a $-$[3], or a structure pattern over some $B_j \in tail(p)$, and
   (b) for any two consecutive β_i, β_{i+1}, they cannot both be $*$'s, and
   (c) if $\beta_i = [B_p, l_p]$, $\beta_j = [B_q, l_q]$, then $i < j \Rightarrow p < q$, $i = j - 1 \Rightarrow p = q - 1$, and
   (d) $\beta_1 = *$, or $-$, or $[B_1, l_1]$, and
   (e) $\beta_m = *$, or $-$, or $[B_k, l_k]$.
5. $\beta = [A, (\beta_1, ..., \beta_m)]$, $m \geq 0$, and for some $p \in P$, $head(p) = A$, $tail(p) = B^*$, and
   (a) each β_i is either a $*$, or a $-$, or a structure pattern over B, and
   (b) for any two consecutive β_i, β_{i+1}, they cannot both be $*$'s.

α is a *value pattern* over a variable A if $[A, \alpha]$ is a structure pattern over A. □

◇ The function *match* determines when a structure matches a given structure pattern. For example, $match(x_2, \beta)$, (x_2 is shown in Figure 5), returns true when β is any of the following structure patterns.

1. $[S, *]$ 2. $[S, (*)]$ 3. $[S, (*, [B, *], *)]$ 4. $[S, ([A, a_1], [B, *], [C, ([A, *], [S, *])])]$

5. $[S, (-, -, [C, *])]$ 6. $[S, (-, [B, ([D, ([E, e_1], *)], *)], *)]$

[3]The symbols $*$ and $-$ denote wildcards. The symbol $*$ matches with none or any number of subtrees of a structure, while $-$ matches with exactly one subtree.

Let t be a structure over some $\mathcal{G} = [\![S, P]\!]$ and let $\beta = [A, l]$ be a structure pattern over some variable A of $\mathcal{G}$. Then,

$$\begin{aligned} match(t, \beta) &= true, \text{ if } ((t = []) \wedge (A = S)) \vee ((root(t) = A) \wedge (l = *)) \\ &= true, \text{ if } (root(t) = A) \wedge (has\text{–}leaf(t)) \wedge ((l = -) \vee (subtrees(t) = l)) \\ &= true, \text{ if } (root(t) = A) \wedge (l \neq -) \wedge (l \notin \mathcal{T}) \wedge (has\text{–}leaf(t) = false) \\ &\qquad \wedge (matchlist(subtrees(t), l) = true) \\ &= false, \text{ otherwise.} \end{aligned}$$

Let $\mathcal{G} = [\![S, P]\!]$ be a scheme and $l_1 = (t_1, ..., t_k)$ be a sublist of some list s, where for some $A \in var(\mathcal{G})$, $[A, s]$ is a structure over $normalize([\![A, P]\!])$. Let $l_2 = (\beta_1, ..., \beta_m)$ be a sublist of some list α where α is a value pattern over A.

$$\begin{aligned} matchlist(l_1, l_2) &= true, \text{ if } (l_1 = l_2) \vee (l_2 = (*)) \\ &= true, \text{ if } (l_1 \neq ()) \wedge (l_2 \neq ()) \wedge (\beta_1 \neq *) \\ &\qquad \wedge ((\beta_1 = -) \vee (match(t_1, \beta_1) = true)) \\ &\qquad \wedge (matchlist(l_1', l_2') = true), \\ &\quad \text{where, } l_1' = (t_2, ..., t_k) \text{ and } l_2' = (\beta_2, ..., \beta_m) \\ &= true, \text{ if } (l_1 \neq ()) \wedge (l_2 \neq ()) \wedge (\beta_1 = *) \\ &\qquad \wedge ((\beta_2 = -) \vee (match(t_i, \beta_2) = true)) \\ &\qquad \wedge (matchlist(l_i'', l_3') = true), \text{ for some } t_i \in l_1 \\ &\quad \text{where, } \forall j < i, \ ((\beta_2 \neq -) \wedge (match(t_j, \beta_2) = false)) \\ &\qquad \vee (matchlist(l_j'', l_3') = false), \\ &\quad l_p'' = (t_{p+1}, ..., t_k), \ 1 \leq p \leq i, \text{ and } l_3' = (\beta_3, ..., \beta_m) \\ &= false, \text{ otherwise} \end{aligned}$$

◇

We now give the complete description of the operators of the algebra. For each operator, we first give an informal description of its operation and its input parameters. We then give descriptive examples and finally its formal definition.

4.3.1 Retrieve

Given variables A and B as parameters, *Retrieve* extracts from the input instance, trees that have A as the root. The result is a tree with B as the root and the list of trees extracted

as its list of subtrees. The trees are extracted in depth-first left-to-right order.

Example 4.1 Consider the query "*Retrieve the titles of all the books in* d_1" where d_1 denotes the instance whose structure is shown in Figure 2.

This query can be expressed as, *Retrieve*(TITLE,ANSWER)d_1. The resulting structure has ANSWER as the root node with the trees subtended by all the TITLE nodes in d_1 as its subtrees, as shown in Figure 6. ■

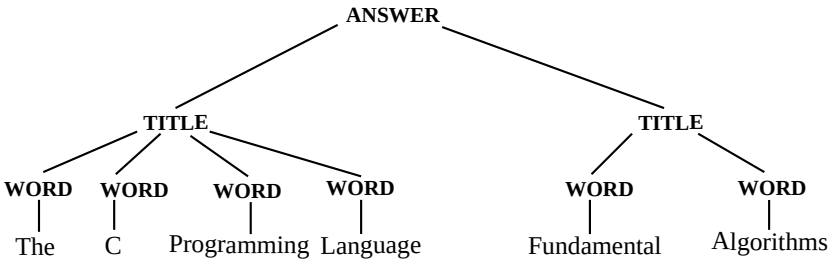


Figure 6: The result of a *Retrieve* operation on the instance d_1.

The same query when applied to the instance d_2 shown in Figure 3 will give the result shown in Figure 7. The result in this case contains not only the subtrees corresponding to the TITLE nodes of the BOOK trees, but also those that correspond to the TITLE nodes of the VOL subtrees. If we wanted to restrict the result to contain only the book-titles or only the volume-titles, we can either rename or delete the TITLE nodes of the VOL trees before performing the *Retrieve* operation.

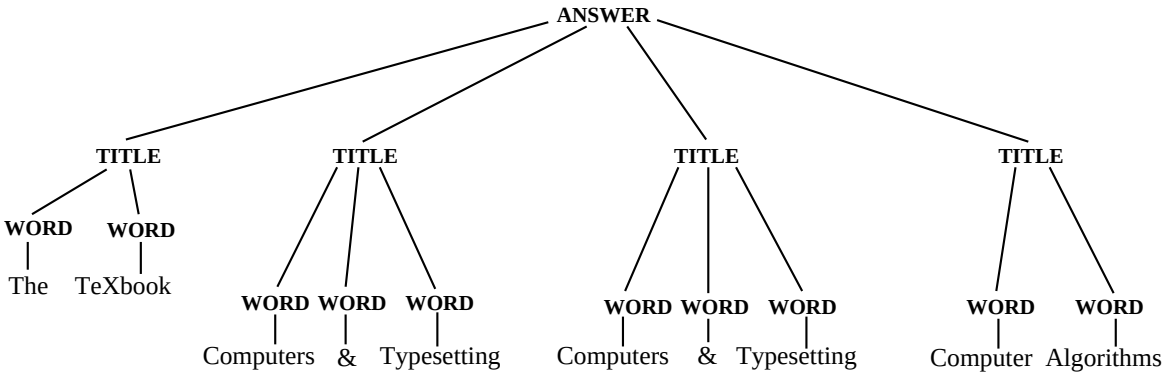


Figure 7: The result of a *Retrieve* operation on the instance d_2.

Definition 4.3 Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance, where $\mathcal{G} = [\![S, P]\!]$, and let A and B be variables such that $A \in var(\mathcal{G})$. Then,

$Retrieve(A, B)\langle t, \mathcal{G} \rangle = \langle t', \mathcal{G}' \rangle$, where,

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \\
&= [B, ()], \text{ if } ((root(t) \neq A) \wedge ((subtrees(t) = ()) \vee (has\text{-}leaf(t)))) \\
&= [B, concat(l_1, ..., l_k)], \text{ if } (root(t) \neq A) \\
&\quad \text{where, } subtrees(t) = (t_1, ..., t_k), \text{ and} \\
&\quad\quad l_i = subtrees(struct(Retrieve(A, B)\langle t_i, \mathcal{G}_i \rangle)), \text{ if } (reachable(A, root(t_i), P)) \\
&\quad\quad\quad = (), \text{ otherwise} \\
&= [B, (t)], \text{ otherwise,} \\
\mathcal{G}' &= normalize([\![B, P \cup \{B \longrightarrow A^*\}]\!]).
\end{aligned}
$$

□

4.3.2 Find

The *Find* operator is used for identifying trees or lists of subtrees, in an instance, that satisfy certain conditions. We will first define a *condition* and the meaning of evaluating a condition, before describing the *Find* operator.

Definition 4.4 c is a *simple* condition on a variable A of scheme $\mathcal{G}$ if for some $p \in P$, $head(p) = A$ and

(a) $c = \phi$, i.e., the empty condition, or

(b) $c = (\theta\ \beta)$, and

1. $\theta \in \{=, \neq, \subset, \subseteq, \supseteq, \supset\}$[4] and β is a list $(b_1, ..., b_n)$, $n \geq 0$, where each $b_i \in \mathcal{T}$, or
2. $\theta \in \{\ni, \not\ni\}$ and β is a structure pattern over some $B \in var(\mathcal{G})$, where for some $p \in P$, $head(p) = A$ and $B \in tail(p)$, or
3. $\theta \in \{\cong, \not\cong\}$ and β is a struct-list over A, or
4. $\theta \in \{\equiv, \not\equiv\}$ and β is a value pattern over A, or
5. $\theta \in \{<, \leq, >, \geq\}$ and β is some constant b, where for some $p \in P$, $head(p) = A$ and either (i) $tail(p) = d$, where d and b are of the same domain, or (ii) $tail(p) = \tau$, where τ is a type and $b \in Dom(\tau)$.

[4]The comparison operators, $\subset$, $\subseteq$, etc., denote list comparisons, such as, sublist, and not the usual set comparisons.

(c) $c = (B\theta C)$, $B \neq C$, and for some $p \in P$, $head(p) = A$, and $B, C \in tail(p)$, and

1. $\theta \in \{=, \neq, \subset, \subseteq, \supset, \supseteq\}$, or
2. $\theta \in \{\cong, \ncong\}$ and $compatible(B, C, P)$, or
3. $\theta \in \{\equiv, \not\equiv\}$ and for some $q_1, q_2 \in P$, $head(q_1) = B$, $head(q_2) = C$, and (i) $tail(q_1) = tail(q_2)$, or (ii) $tail(q_1) \in \mathcal{T}$, $tail(q_2) \in$, and $tail(q_1) \in Dom(tail(q_2))$, or vice-versa, or
4. $\theta \in \{<, \leq, >, \geq\}$ and for some $q_1, q_2 \in P$, $head(q_1) = B$, $head(q_2) = C$, and (i) $tail(q_1) = tail(q_2) = \tau$, where $\tau \in$, , or (ii) $tail(q_1) = b_1$ and $tail(q_2) = b_2$, b_1 and b_2 are constants of the same domain, or (iii) $tail(q_1) \in \mathcal{T}$ and $tail(q_2) \in$, , and $tail(q_1) \in Dom(tail(q_2))$, or vice-versa.

c is a *condition* on variable A of scheme $\mathcal{G}$, if

1. c is a simple condition on A.
2. c is of the form $[B_j c_j]$, where for some $p \in P$, $head(p) = A$ and $B_j \in tail(p)$ and c_j is a condition on B_j.
3. c is of the form $\forall[B_j c_j]$, where for some $p \in P$, $head(p) = A$ and $tail(p) = B_j^*$ and c_j is a condition on B_j.
4. $c = c_1 \wedge c_2$, or $c = c_1 \vee c_2$, or $c = \neg c_1$, where c_1 and c_2 are conditions on A. □

We now give the meaning of evaluating a condition on a structure. If c is a condition on a variable A of scheme $\mathcal{G}$, and $\langle t, \mathcal{G}\rangle$ is a list-structure instance, $(t \neq [])$, then

$c(t) =$ true, if one of the following is true.

1. c is ϕ.
2. c is $(\theta\ \beta)$, and
   1. $\theta \in \{=, \neq, \subset, \subseteq, \supseteq, \supset\}$ and $leaves(t)\theta\beta =$ true.
   2. θ is $\ni$, $has\text{–}leaf(t) =$ false and $match(t_i, \beta) =$ true for some $t_i \in subtrees(t)$.
   3. θ is $\not\ni$ and $has\text{–}leaf(t) =$ true or $match(t_i, \beta) =$ false for all $t_i \in subtrees(t)$.
   4. θ is $\cong$ and $str\text{–}list(t) = \beta$.
   5. θ is $\ncong$ and $str\text{–}list(t) \neq \beta$.
   6. θ is $\equiv$ and $match(t, [root(t), \beta]) =$ true.

7. θ is $\not\equiv$ and $match(t, [root(t), \beta]) =$ false.
8. $\theta \in \{<, \leq, >, \geq\}$, $has\text{–}leaf(t) =$ true, and $subtrees(t)\theta\beta =$ true.

3. $c = (B\theta C)$, $(B \in children(t))$, $(C \in children(t))$, and
   1. $\theta \in \{=, \neq, \subset, \subseteq, \supset, \supseteq\}$ and $(leaves(t[B])\ \theta\ leaves(t[C])) =$ true.
   2. θ is $\cong$, and $str\text{–}list(t[B]) = str\text{–}list(t[C])$.
   3. θ is $\not\cong$, and $str\text{–}list(t[B]) \neq str\text{–}list(t[C])$.
   4. θ is $\equiv$, and $subtrees(t[B]) = subtrees(t[C])$.
   5. θ is $\not\equiv$, and $subtrees(t[B]) \neq subtrees(t[C])$.
   6. $\theta \in \{<, \leq, >, \geq\}$, $has\text{–}leaf(t[B]) =$ true, $has\text{–}leaf(t[C]) =$ true, and $(subtrees(t[B])\ \theta\ subtrees(t[C])) =$ true.
4. c is $[B_j c_j]$, $has\text{–}leaf(t) =$ false and for some $t_i \in subtrees(t)$, $root(t_i) = B_j$ and $c_j(t_i) = true$.
5. c is $\forall[B_j c_j]$, $has\text{–}leaf(t) =$ false and for each $t_i \in subtrees(t)$, $root(t_i) = B_j$ and $c_j(t_i) = true$.
6. $c = c_1 \wedge c_2$ and both $c_1(t)$ and $c_2(t)$ are true, or $c = c_1 \vee c_2$ and either $c_1(t)$ or $c_2(t)$ is true, or $c = \neg c_1$ and $c_1(t)$ is not true.

We now describe the *Find* operator. *Find* has three parameters – a variable A, which is in the schema of the instance, a search parameter c, and a variable B. *Find* selects trees that have A as the root node and that satisfy the condition specified by the search parameter. The variable B (usually chosen so that it is not one of the variables of the scheme) is used to mark the selected nodes. The search parameter, c, can be one of four different forms and *Find* has different actions corresponding to each of the four cases.

1. c can be a condition on A. In this case *Find* searches for trees in the instance whose root nodes are A's and that satisfy the condition c. In each tree t_i that satisfies the search condition, B is introduced as its (only) child and as the parent of its subtrees.

   Example 4.2 *Find all the books in* d_1 *that were published in* 1973.

   The expression *Find*(`BOOK, [YEAR (≡ 1973)], B`)d_1 will locate all the `BOOK` nodes that have a `YEAR` node as a child, whose value (leaf-node) is `1973`. Each `BOOK` tree that satisfies

this condition will be marked by the introduction of a B node as its child. We can then perform a *Retrieve*(B,C) on the result of the previous expression to select out the marked books. We list below other ways of expressing the *Find* part of the same query.

Find(BOOK, [YEAR (= (1973))], B)d_1

Find(BOOK, (≡ ([YEAR, 1973],*)), B)d_1

Find(BOOK, (∋ [YEAR, 1973]), B)d_1

Figure 8 shows the structure of the result of the above query. ■

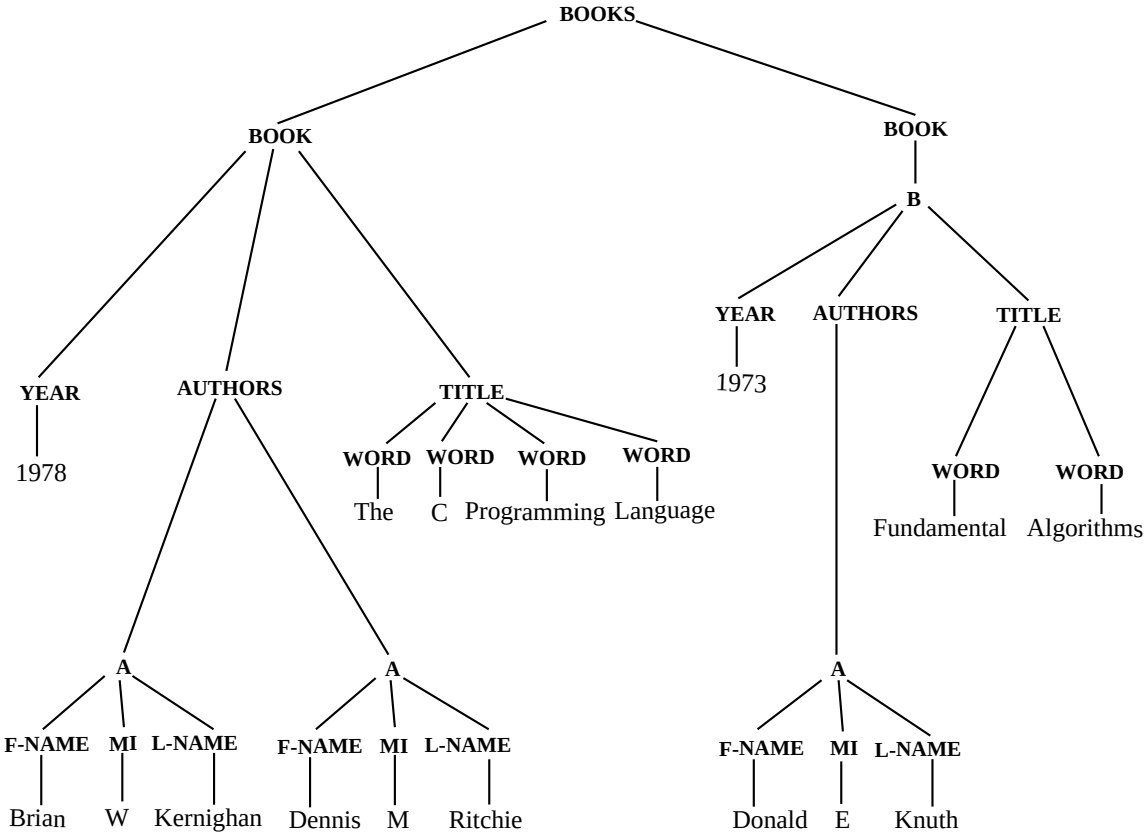


Figure 8: The result of applying a *Find* operation.

2. c can be of the form $(c_1, <)$ or $(c_1, >)$, where c_1 is a condition on A. *Find* looks for trees whose children are all A's and for each such tree it marks either the left-most or the right-most subtree that satisfies the condition c_1, depending on whether the second element in c is '<', or '>'.

Example 4.3 *For each book in d_1, find the last author.*

Find(A, (ϕ, >), B)d_1.

As before, this must be followed by a *Retrieve* operation. ■

3. c can be a positive integer i, in which case *Find* looks for trees whose children are all A's and marks the ith subtree by introducing B as its child and as the parent of its subtrees.

   Example 4.4 *Find all the books authored by two or more persons.*

   Find(BOOK, [AUTHORS ($\ni$ [A,([C,*])])], B)(*Find*(A, 2, C)d_1)

   The above expression will first look for trees that have all A nodes as children and then mark the second subtree (if there are at least two subtrees) of such trees by introducing C as the subtree's child. Next, trees that have BOOK as the root node and whose AUTHORS attribute has an A subtree whose child is C, are marked by B. This must then be followed by a *Retrieve* to extract the marked trees. The following is an alternate way of expressing (the *Find* part) of the same query.

   Find(BOOK, [AUTHORS ($\equiv$ ([A,*],[A,*],*))], B)d_1. ■

4. c can be a value pattern of the form $(\beta_1, ..., \beta_m)$, where each β_i is a structure pattern on A. *Find* searches the instance for trees whose children are all A's. In each such tree it then searches, in left to right order, for lists of subtrees that match the pattern c. Each such list is replaced by a tree whose root node is A and whose list of subtrees is the list being replaced, but with the A nodes in the list renamed to B. This case is useful when subtrees corresponding to some patterns have to be identified in order to be deleted or modified, as illustrated by the following example.

   Example 4.5 Let us suppose that we want to replace every occurrence of 'Data Base' in the titles of books and volumes in the instance d_3, whose structure is shown in Figure 9(a), with 'Database'. We will need to identify this pattern in the TITLE trees in the instance. The following expression, however, will not serve our purpose since it will mark those TITLE trees that contain the pattern in their values, but not mark the pattern itself.

   Find(TITLE, ($\equiv$ (*, [WORD, Data], [WORD, Base], *)), B)d_3.

The expression shown below, on the other hand, will look for two consecutive `WORD` trees that have `'Data'` and `'Base'` as their respective values, and will then mark both trees by changing their root nodes to `B`, and replacing the two subtrees by a single tree that has `WORD` as the root and the two trees as its subtrees.

$Find$(`WORD, ([WORD, Data], [WORD, Base]), B`)d_3.

Figure 9(b) shows the structure of the result of the above query. ■

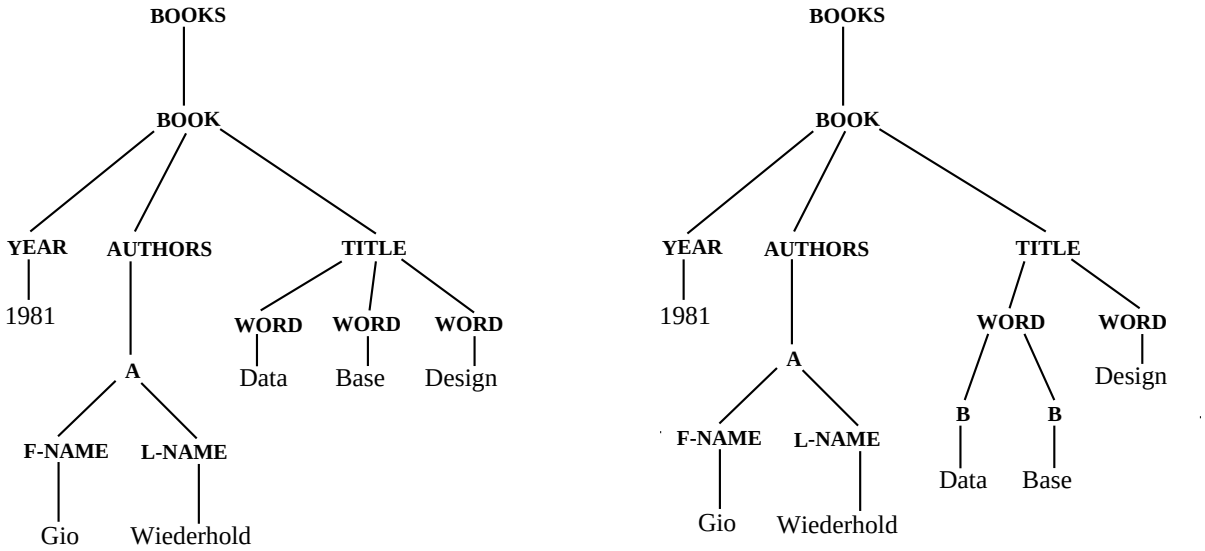


Figure 9: The result of a $Find$ operation on patterns.

We now give the formal definition for the $Find$ operator corresponding to each of the four different cases.

Definition 4.5 Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let A and B be variables such that $A \in var(\mathcal{G})$, then

$Find(A, c, B)\langle t, \mathcal{G} \rangle = \langle t', \mathcal{G}' \rangle$, where, t' and $\mathcal{G}'$ are as follows.

Case 1: c is a condition on A.

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (((root(t) \neq A) \vee (c(t) = false)) \\
&\qquad \wedge((subtrees(t) = ()) \vee (has\text{–}leaf(t))))^5 \\
&= [root(t), ([B, subtrees(t)])], \text{ if } (root(t) = A) \wedge (has\text{–}leaf(t)) \\
&\qquad \wedge(c(t) = true) \\
&= [root(t), ([B, (t'_1, ..., t'_k)])], \text{ if } (root(t) = A) \wedge (c(t) = true) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ otherwise,} \\
\text{where, } t'_i &= struct(Find(A, c, B)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}
$$

Case 2: c is of the form $(c_1, <)$ or $(c_1, >)$, where c_1 is a condition on A and for some production p in P, $tail(p) = A^*$.

$$
\begin{aligned}
t' &= t \text{ if } (t = []) \vee (subtrees(t) = ()) \vee (has\text{–}leaf(t)) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (\exists t_i \in subtrees(t) \mid root(t_i) \neq A) \\
&= [root(t), (t''_1, ..., t''_k)], \text{ otherwise, where,} \\
&\qquad t''_i = t'_i, \text{ if } (c_1(t_i) = false) \\
&\qquad\qquad \vee((c = (c_1, <)) \wedge (\exists t_j \mid (j < i) \wedge (c_1(t_j) = true))) \\
&\qquad\qquad \vee((c = (c_1, >)) \wedge (\exists t_j \mid (j > i) \wedge (c_1(t_j) = true))) \\
&\qquad\quad = [root(t_i), ([B, subtrees(t'_i)])], \text{ otherwise} \\
\text{where, } t'_i &= struct(Find(A, c, B)\langle t_i, \mathcal{G}_i\rangle), \text{ if} \\
&\qquad (\exists p \in P \mid (tail(p) = A^*) \wedge (reachable(head(p), root(t_i), P))) \\
&\qquad = t_i, \text{ otherwise.}
\end{aligned}
$$

[5] In the definitions of all the operators, in a disjunctive condition of the form $(t = []) \vee \cdots$, we assume that the test for the empty structure is done first, and if the test succeeds the rest of the clauses are not tested.

Case 3: c is an integer, $c > 0$, and for some p in P, $tail(p) = A^*$.

$$
\begin{aligned}
t' &= t \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (\exists t_i \in subtrees(t) \mid root(t_i) \neq A) \vee (c > k) \\
&= [root(t), (t'_1, ..., t'_{c-1}, t''_c, t'_{c+1}, ..., t'_k)], \text{ otherwise,} \\
&\quad \text{where, } t''_c = [root(t_c), ([B, subtrees(t'_c)])] \\
\text{where, } t'_i &= struct(Find(A, c, B)\langle t_i, \mathcal{G}_i\rangle), \text{ if} \\
&\qquad (\exists p \in P \mid (tail(p) = A^*) \wedge (reachable(head(p), root(t_i), P))) \\
&= t_i, \text{ otherwise.}
\end{aligned}
$$

Case 4: c is of the form $(\beta_1, ..., \beta_m)$, $m \geq 1$, where each β_i is a structure pattern on A, and for some $p \in P$, $tail(p) = A^*$.

$$
\begin{aligned}
t' &= t \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (\exists t_i \in subtrees(t) \mid root(t_i) \neq A) \vee (k < m), \\
&= [root(t), l], \text{ otherwise, where,} \\
&\quad l = append(t'', l'_1), \text{ if } matchlist((t_1, ..., t_m), c) = true, \text{ where} \\
&\qquad t'' = [A, (t''_1, ..., t''_m)], \text{ where} \\
&\qquad\quad t''_i = [B, subtrees(t'_i)], \\
&\qquad l'_1 = subtrees(struct(Find(A, c, B)\langle [root(t), (t_{m+1}, ..., t_k)], \mathcal{G}\rangle)) \\
&\quad = append(t'_1, l'_2), \text{ if } matchlist((t_1, ..., t_m), c) = false, \text{ where} \\
&\qquad l'_2 = subtrees(struct(Find(A, c, B)\langle [root(t), (t_2, ..., t_k)], \mathcal{G}\rangle)) \\
\text{where, } t'_i &= struct(Find(A, c, B)\langle t_i, \mathcal{G}_i\rangle), \text{ if} \\
&\qquad (\exists p \in P \mid (tail(p) = A^*) \wedge (reachable(head(p), root(t_i), P))) \\
&\quad = t_i, \text{ otherwise}
\end{aligned}
$$

We have given the definitions for the structure part of the result of the $Find$ operator, i.e., the definition for t', where $Find(A, c, B)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$. We now give the definition for the scheme part of the result, i.e., $\mathcal{G}'$, for all four cases.

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S, P \cup P' \cup P'']\!]), \text{ where,} \\
P' &= \{A \longrightarrow B\}, \text{ for Cases 1, 2, and 3} \\
&= \{A \longrightarrow B^*\}, \text{ for Case 4, and} \\
P'' &= \{p \mid (head(p) = B) \wedge (\exists p_1 \in P \mid (tail(p) = tail(p_1)) \wedge (head(p_1) = A))\}. \qquad \square
\end{aligned}
$$

We give a few more examples to illustrate the use of the *Find* operator. In each example we give only the part of the expression corresponding to the *Find* part, i.e., the marking part of the query.

Example 4.6 *Find all the books that do not have the word* 'Database' *in the book-title.*

$Find$(BOOK, [TITLE $\forall$[WORD ($\neq$ (Database))]], NON–DB)d_1 ■

Example 4.7 *Find all the books that do not have the word* 'Database' *in either the book-title or the volume-title.*

$Find$(BOOK, [TITLE $\forall$[WORD ($\neq$ (Database))]]
$\wedge$ (($\not\ni$ [VOL,([TITLE,*],*)])
$\vee$ [VOL, [TITLE $\forall$[WORD ($\neq$ (Database))]]]), NON–DB)d_1 ■

Example 4.8 *Find all the books written by two or more authors, where one of the authors is* 'Dennis C. Ritchie'.

$Find$(BOOK, [AUTHORS ($\equiv$ (-,-,*))
$\wedge$ [A (= (Dennis C Ritchie))]], B)d_1 ■

Example 4.9 *Find all the books written by two or more authors, where the second author is* 'Dennis C. Ritchie'. We list several possible ways of expressing the *Find* part of this query.

1. $Find$(BOOK, [AUTHORS, ($\equiv$
(-, [A,([F–NAME, Dennis],[MI,C],[L–NAME,Ritchie])],*))], B)d_1.

2. $Find$(BOOK, [AUTHORS [A, [C, (= (Dennis C Ritchie))]]], B)
($Find$(A, 2, C)d_1).

3. $Find$(BOOK, [AUTHORS ($\equiv$ (-,[A,([C,*])],*))], B)
($Find$(A, (= (Dennis C Ritchie)), C)d_1).

■

4.3.3 Insert

The *Insert* operator takes two instances as input and inserts the second in the first. The structure of the second instance is added, as a subtree, to trees in the first instance that either correspond to aggregate productions, thereby adding a new attribute, or to trees that correspond to list productions. *Insert* has two parameters – a variable A and an *insert pattern* over A – which determine where the tree is inserted. The variable A specifies the trees in the first instance in which the second instance is to be inserted, while the insert pattern specifies the position amongst the subtrees where it is to be inserted. An insert pattern is defined as follows:

Definition 4.6 Let $\mathcal{G} = [\![S, P]\!]$ be a scheme and let $A \in var(\mathcal{G})$ be such that for some production $p \in P$, $head(p) = A$ and $tail(p) = B^*$ or $tail(p) = B_1...B_k$. Also, let P be such that for any two productions p and q in P, if $head(p) = head(q) = A$, and $tail(p) = B^*$ and $tail(q) = B_1...B_m$, then $B \neq B_i$, for each $B_i \in tail(q)$. Then, β is an *insert pattern* on A if for some $p \in P$, $head(p) = A$, p is either an aggregate or a list production and

1. $\beta = (*, -)$, or $\beta = (-, *)$ and for every pair of productions p_1, p_2 in P, if $head(p_1) = head(p_2) = A$, then both p_1 and p_2 are aggregate productions, or they are both list productions, or either p_1 or p_2, or both, are leaf productions, or

2. $\beta = (\beta_1, \beta_2)$, and

   (a) $\beta_1 = *$ and β_2 is a structure pattern over some variable $B \in tail(p)$, or

   (b) $\beta_2 = *$ and β_1 is a structure pattern over some variable $B \in tail(p)$, or

   (c) β_1 and β_2 are structure patterns over some B where $tail(p) = B^*$, or

   (d) β_1 is a structure pattern over some variable B_i and β_2 is a structure pattern over some variable B_{i+1}, where B_i and B_{i+1} occur consecutively in $tail(p)$.

If β is an insert pattern on A and for some $p \in P$, $head(p) = A$ and β and p satisfy one of the conditions above then we say that β *corresponds to* p. □

Let $\mathcal{G}_1 = [\![S_1, P_1]\!]$ and $\mathcal{G}_2 = [\![S_2, P_2]\!]$ be two schemes and let $\langle t_1, \mathcal{G}_1 \rangle$ and $\langle t_2, \mathcal{G}_2 \rangle$ be two list-structure instances. Let A be a variable in $var(\mathcal{G}_1)$ such that A is the head of some non-leaf production in P_1 and let β be an insert pattern over A. Also, if β corresponds to a list production p, then $S_2 \in tail(p)$ and if β corresponds to an aggregate production p, then $S_2 \notin tail(p)$. Then, $Insert(A, \beta)(\langle t_1, \mathcal{G}_1 \rangle, \langle t_2, \mathcal{G}_2 \rangle)$ inserts the structure t_2 in t_1. The positions in t_1 where t_2 is inserted are determined by the insert pattern as follows.

1. If $\beta = (*, -)$ or $\beta = (-, *)$, then P_1 contains at least one aggregate production with A as the root or exactly one list production with A as the root (but not both)[6]. For every tree in t_1, whose root is A and whose subtree is not a leaf, t_2 is inserted as the left-most subtree or as the right-most subtree depending on whether β is $(-, *)$ or $(*, -)$, respectively.

   Example 4.10 Let P be the set of productions in the book-collection-scheme (Figure 1). Let d_4 represent the book instance whose structure, shown in Figure 10, corresponds to the scheme $normalize([\![BOOK, P]\!])$ and let us suppose that we want to insert d_4 in the instance d_1.

   $Insert$(BOOKS, (-,*))(d_1,d_4).

   As a result of the above operation, d_4 will be inserted as the left-most subtree of the BOOKS tree in d_1. ■

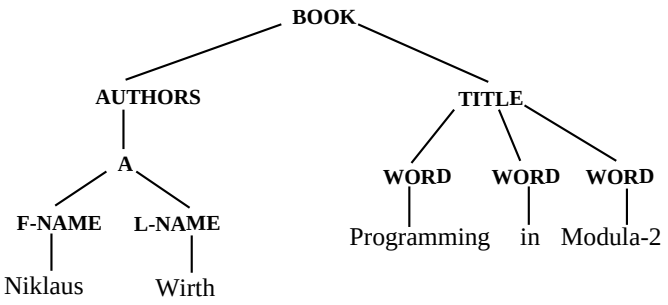


Figure 10: A BOOK instance.

2. If $\beta = (\beta_1, \beta_2)$, then for every tree in t_1 that has A as the root, t_2 is inserted between every pair of adjacent subtrees s_i, s_{i+1} that match the insert pattern. If $\beta_1 = *$, it looks for a match with β_2 alone and t_2 is inserted before it. If $\beta_2 = *$, it is inserted after the subtree that matches β_1. If β corresponds to a list production then several subtrees (or pairs of subtrees) could satisfy the insert pattern. Again, by definition, β can correspond to exactly one list production or to several aggregate productions (but not both).

   Example 4.11 Let *p–info* represent the following list-structure instance.

   $\langle$[PUBLISHER, Addison–Wesley], $[\![$PUBLISHER, {PUBLISHER $\longrightarrow$ String}$]\!]\rangle$.

[6]The *Find* operator and the *Rename* and *Replace* operators, which are described in Section 4.3.6 and 4.3.7 can be used to make the scheme of an instance satisfy this requirement.

Insert the above publisher information for the book, in instance d_1, whose book-title is `'Fundamental Algorithms'` *and insert this information to the right of the* `YEAR` *node.*

Insert(`B, ([YEAR, *],*)`)
`((Find(BOOK, [TITLE, (= (Fundamental Algorithms))], B)`d_1`)`, *p–info*)

Figure 11 shows the result of the above query. In Section 4.3.7, we will show how the extra nodes introduced by the *Find* operator, such as the `B` node in this example, can be removed by means of the *Replace* operator. ■

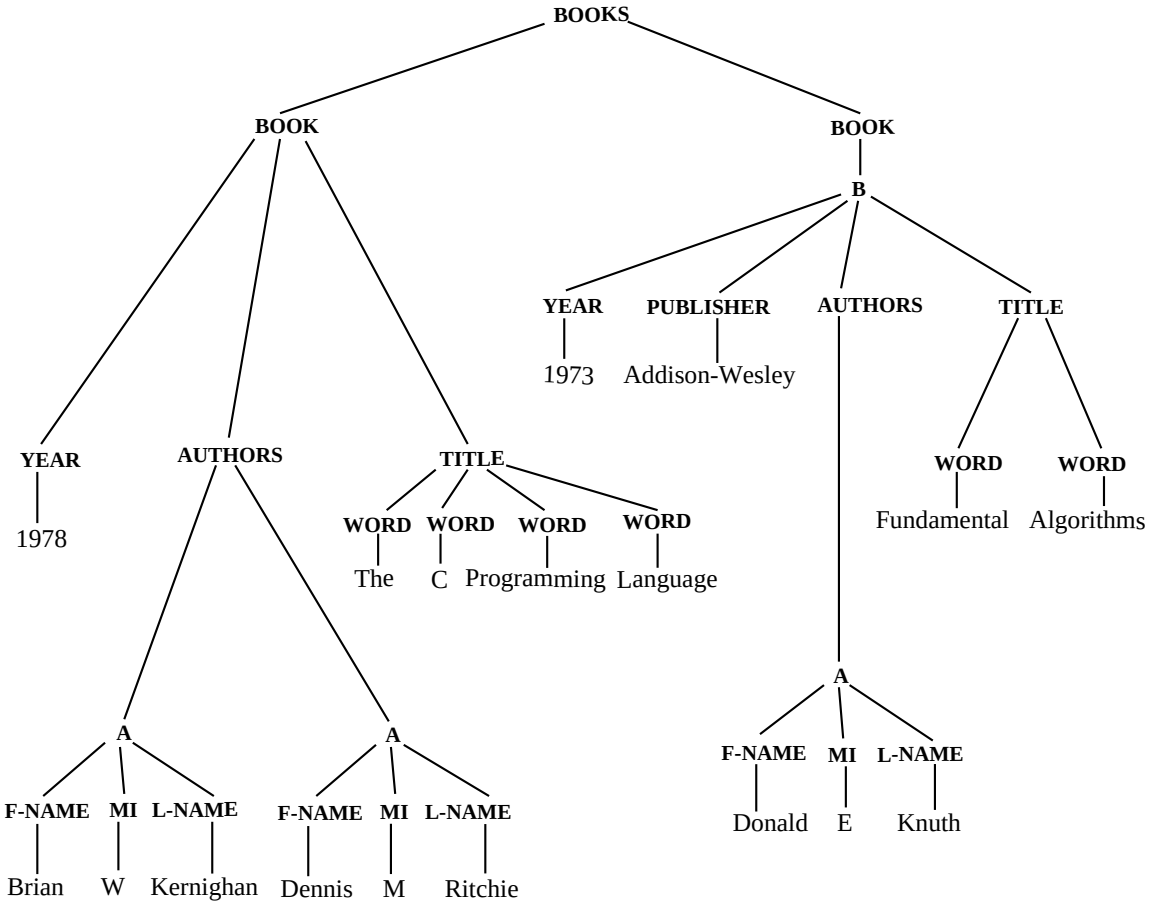


Figure 11: The result of an *Insert* operation.

Definition 4.7 $Insert(A, \beta)(\langle t_1, \mathcal{G}_1\rangle, \langle t_2, \mathcal{G}_2\rangle) = \langle t', \mathcal{G}'\rangle$, where t' and $\mathcal{G}'$ are defined as follows:

Case 1: β corresponds to an aggregate production p where $head(p) = A$.

$$
\begin{aligned}
t' &= t_1, \text{ if } (t_1 = []) \vee (t_2 = []) \vee (has\text{–}leaf(t_1)) \vee (subtrees(t_1) = ()) \\
&= [root(t_1), (t'_{11}, ..., t'_{1k}, t_2)], \text{ if } (root(t_1) = A) \wedge (\beta = (*, -)) \\
&= [root(t_1), (t_2, t'_{11}, ..., t'_{1k})], \text{ if } (root(t_1) = A) \wedge (\beta = (-, *)) \\
&= [root(t_1), (t'_{11}, ..., t'_{1m-1}, t_2, t'_{1m}, ..., t'_{1k})], \text{ if } (root(t_1) = A) \\
&\quad \wedge (\beta = (*, \beta_2)) \wedge (match(t_{1m}, \beta_2) = true) \\
&= [root(t_1), (t'_{11}, ..., t'_{1m}, t_2, t'_{1m+1}, ..., t'_{1k})], \text{ if } (root(t_1) = A) \\
&\quad \wedge (((\beta = (\beta_1, *)) \wedge (match(t_{1m}, \beta_1) = true)) \\
&\quad\quad \vee ((\beta = (\beta_1, \beta_2)) \wedge (match(t_{1m}, \beta_1) = true) \wedge (match(t_{1m+1}, \beta_2) = true))) \\
&= [root(t_1), (t'_{11}, ..., t'_{1k})], \text{ otherwise,}
\end{aligned}
$$

where, $subtrees(t_1) = (t_{11}, ..., t_{1k})$, and

$$
\begin{aligned}
t'_{1i} &= struct(Insert(A, \beta)(\langle t_{1i}, \mathcal{G}_{1i} \rangle, \langle t_2, \mathcal{G}_2 \rangle)), \text{ if } (reachable(A, root(t_{1i}), P)) \\
&= t_{1i}, \text{ otherwise.}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S_1, P_1 \cup P_2 \cup P_3]\!]), \text{ where} \\
&\quad P_1 \text{ and } P_2 \text{ are the sets of productions of } \mathcal{G}_1 \text{ and } \mathcal{G}_2, \text{ respectively, and} \\
&\quad\quad P_3 = \{p_1 \mid (head(p_1) = A) \wedge (tail(p_1) = B_1...B_m S_2 B_{m+1}...B_k) \wedge \\
&\quad\quad\quad ((\exists p \in P_1) \wedge (head(p) = A) \wedge (tail(p) = B_1...B_m B_{m+1}...B_k) \wedge \\
&\quad\quad\quad\quad (\beta \text{ corresponds to } p))\}, \\
&\quad\quad\quad \text{if } ((\beta_1 \neq -) \wedge (\beta_1 \neq *) \wedge (root(\beta_1) = B_m,\ m > 0))^7 \\
&\quad\quad\quad \vee ((\beta_2 \neq -) \wedge (\beta_2 \neq *) \wedge (root(\beta_2) = B_{m+1},\ m \geq 0)) \\
&\quad\quad = \{p_1 \mid (head(p_1) = A) \wedge (tail(p_1) = S_2 B_1...B_m) \\
&\quad\quad\quad \wedge ((\exists p \in P_1) \wedge (head(p) = A) \wedge (tail(p) = B_1...B_m))\}, \text{ if } (\beta = (-, *)) \\
&\quad\quad = \{p_1 \mid (head(p_1) = A) \wedge (tail(p_1) = B_1...B_m S_2) \\
&\quad\quad\quad \wedge ((\exists p \in P_1) \wedge (head(p) = A) \wedge (tail(p) = B_1...B_m))\}, \text{ if } (\beta = (*, -))
\end{aligned}
$$

[7] For a structure pattern $\beta_1 = [B_1, l_1]$, $root(\beta_1) = B_1$.

Case 2: β corresponds to a list production p, where $head(p) = A$.

$$
\begin{aligned}
t' &= t_1, \text{ if } (t_1 = []) \vee (has\text{–}leaf(t_1)) \vee (t_2 = []) \vee \\
&\quad ((subtrees(t_1) = ()) \wedge ((root(t_1) \neq A) \vee ((\beta \neq (*, -)) \wedge (\beta \neq (-, *))))) \\
&= [root(t_1), (t_2, t'_{11}, ..., t'_{1k})], \text{ if } (root(t_1) = A) \wedge (\beta = (-, *)) \\
&= [root(t_1), (t'_{11}, ..., t'_{1k}, t_2)], \text{ if } (root(t_1) = A) \wedge (\beta = (*, -)) \\
&= [root(t_1), concat(l'_1, ..., l'_k)], \text{ if } (root(t_1) = A) \wedge \\
&\qquad (\forall t_{1i} \in subtrees(t_1)(((\beta_1 \neq *) \wedge (root(t_{1i}) = root(\beta_1))) \\
&\qquad\qquad \vee ((\beta_2 \neq *) \wedge (root(t_{1i}) = root(\beta_2))))) \\
&\quad \text{where, } l'_i = (t_2, t'_{1i}), \text{ if } (\beta = (*, \beta_2)) \wedge (match(t_{1i}, \beta_2) = true) \\
&\qquad = (t'_{1i}, t_2), \text{ if } (\beta = (\beta_1, \beta_2)) \wedge (match(t_{1i}, \beta_1) = true) \\
&\qquad\quad \wedge (match(t_{1(i+1)}, \beta_2) = true), \ (1 \leq i < k) \\
&\qquad = (t'_{1i}), \text{ if } (\beta = (\beta_1, \beta_2)) \text{ for } i = k \\
&\qquad = (t'_{1i}, t_2), \text{ if } (\beta = (\beta_1, *)) \wedge (match(t_{1i}, \beta_1) = true) \\
&\qquad = (t'_{1i}), \text{ otherwise} \\
&= [root(t_1), (t'_{11}, ..., t'_{1k})], \text{ otherwise,}
\end{aligned}
$$

where $subtrees(t_1) = (t_{11}, ..., t_{1k})$, and,

$$
\begin{aligned}
t'_{1i} &= struct(Insert(A, \beta)(\langle t_{1i}, \mathcal{G}_{1i} \rangle, \langle t_2, \mathcal{G}_2 \rangle)), \text{ if } (reachable(A, root(t_{1i}), P)) \\
&= t_{1i}, \text{ otherwise.}
\end{aligned}
$$

$$
\mathcal{G}' = normalize([\![S_1, P_1 \cup P_2]\!]). \qquad \square
$$

4.3.4 Delete

The *Delete* operator finds trees that have a given variable as the root, and deletes from each such tree, subtrees, determined by a 'delete pattern'. Subtrees can be deleted either from trees that correspond to an aggregate production or to a list production. Deleting subtrees from aggregate trees is similar to the projection operator in the relational algebra.

Definition 4.8 Let $\mathcal{G} = [\![S, P]\!]$ be a scheme and let $A \in var(\mathcal{G})$ be such that for some production $p \in P$, $head(p) = A$ and $tail(p) = B^*$ or $tail(p) = B_1...B_k$. Then β is a delete pattern over A if for some $p \in P$, $head(p) = A$, and

1. $tail(p) = B_1...B_k$, $k > 1$, and

   (a) $\beta = (\equiv \ \beta_1)$ or $\beta = (\not\equiv \ \beta_1)$, where β_1 is a structure pattern over some $B_i \in tail(p)$,

or

(b) $\beta = (\notin (C_1, ..., C_m))$, where (i) for some $C_i \in (C_1, ..., C_m)$, $C_i \in tail(p)$, and (ii) there is a variable $B \in tail(p)$ such that $B \notin (C_1, ..., C_m)$, and (iii) for each $C_i \in (C_1, ..., C_m)$, there is some aggregate production q such that $C_i \in tail(q)$ and $head(q) = A$ and there is a variable $D \in tail(q)$ such that $D \notin (C_1, ..., C_m)$, or

2. $tail(p) = B^*$, $\beta = (\beta_1, \beta_2, \beta_3)$, $\beta_2 = (\beta_{21}, ..., \beta_{2m})$, $m \geq 1$, where each β_{2i} is either a '$-$' or a structure pattern over B, at least one of the $\beta'_{2i}s$ is not a $-$ and β_1 and β_3 are either '$*$' or structure patterns over B.

If β and p satisfy one of the above two conditions we say that β corresponds to p. □

Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let A be a variable in $var(\mathcal{G})$ and β a delete pattern over A. Then $Delete(A, \beta)\langle t, \mathcal{G} \rangle$ removes trees within t as described below.

1. β corresponds to an aggregate production.

(a) If $\beta = (\equiv \ \beta_1)$, then $Delete(A, \beta)\langle t, \mathcal{G} \rangle$ removes, from each tree in t that has A as the root, the subtree that matches β_1. If $\beta = (\not\equiv \ \beta_1)$, the subtree whose root is the same as $root(\beta_1)$ and that does not match β_1 is deleted. The tree from which a subtree is deleted must have at least two subtrees.

Example 4.12 *Remove the* `VOL` *attribute from all books in* d_1.

$Delete$`(BOOK, (≡ [VOL,*]))`d_1. ■

(b) If $\beta = (\notin (C_1, ..., C_m))$, then from each tree that has A as the root and at least one of the C_i's as a child, all subtrees whose root nodes are not in the list $(C_1, ..., C_m)$ are deleted.

Example 4.13 *Delete all of the attributes except* `AUTHORS` *and* `TITLE` *from the* `BOOK` *trees in* d_1.

$Delete$`(BOOK, (∉ (AUTHORS, TITLE)))`d_1. ■

2. β corresponds to a list production.

If $\beta = (\beta_1, \beta_2, \beta_3)$, then from each tree in t that has A as the root, $Delete(A, \beta)\langle t, \mathcal{G}\rangle$ removes lists of subtrees where each list matches β_2 and whose left sibling matches β_1, (if $\beta_1 \neq *$), and whose right sibling matches β_3, (if $\beta_3 \neq *$).

Example 4.14 *Delete the word* `'Fundamental'` *in the title* `'Fundamental Algorithms'`, *in* d_1.

$Delete$`(TITLE, (*,([WORD,Fundamental]),[WORD, Algorithms]))`d_1. ■

Definition 4.9 $Delete(A, \beta)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$, where, t' and $\mathcal{G}'$ are defined as follows:

Case 1: β corresponds to an aggregate production. β is of the form $(\theta\ \beta_1)$ where $\theta \in \{\equiv, \not\equiv\}$ or β is of the form $(\notin (C_1, ..., C_m))$.

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), l], \text{ if } (root(t) = A) \wedge (size(subtrees(t)) \geq 2) \\
&\qquad \wedge (\forall t_i, t_j \in subtrees(t)\ (t_i \neq t_j)) \text{ where,} \\
&\quad l = (t'_1, ...t'_{m-1}, t'_{m+1}, ..., t'_k), \\
&\qquad \text{if } ((\beta = (\equiv \beta_1)) \wedge (match(t_m, \beta_1) = true)) \\
&\qquad\ \vee ((\beta = (\not\equiv \beta_1)) \wedge (root(t_m) = root(\beta_m)) \wedge (match(t_m, \beta_1) = false)) \\
&\qquad = concat(l_1, ..., l_k), \text{ if } (\beta = (\notin (C_1, ..., C_m)) \\
&\qquad\qquad \wedge (\exists t_i \in subtrees(t) \mid (root(t_i) \in (C_1, ..., C_m))) \\
&\qquad\quad \text{where, } l_i = (), \text{ if } (root(t_i) \notin (C_1, ..., C_m)) \\
&\qquad\qquad\quad = (t'_i), \text{ otherwise,} \\
&\qquad = (t'_1, ..., t'_k), \text{ otherwise,} \\
&= [root(t), (t'_1, ..., t'_k)], \text{ otherwise} \\
\text{where, } t'_i &= struct(Delete(A, \beta)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}
$$

Case 2: β corresponds to a list production p where $head(p) = A$ and $\beta = (\beta_1, \beta_2, \beta_3)$. Let $length(\beta_2) = m$.

$$
\begin{aligned}
t' &= t, \text{if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p) = false), \\
&= [root(t), l], \text{ otherwise, where} \\
&\quad l = append(t'_1, l_1) \text{ if } (\beta_1 \neq *) \wedge \\
&\qquad (match(t_1, \beta_1) = true) \wedge (matchlist((t_2, ..., t_{m+1}), \beta_2) = true) \\
&\qquad \wedge ((match(t_{m+2}, \beta_3) = true) \vee (\beta_3 = *)) \\
&\quad = l_1 \text{ if } (\beta_1 = *) \wedge (matchlist((t_1, ..., t_m), \beta_2) = true) \wedge \\
&\qquad ((match(t_{m+1}, \beta_3) = true) \vee (\beta_3 = *)) \\
&\quad = append(t'_1, l_2), \text{ otherwise,} \\
\text{where, } l_1 &= subtrees(struct(Delete(A, \beta)\langle [root(t), (t_s, ..., t_k)], \mathcal{G}\rangle)), \\
&\quad \text{where, } s = m + 2, \text{if } \beta_1 \neq *, \ s = m + 1, \text{ if } \beta_1 = *, \text{ and} \\
&\quad l_2 = subtrees(struct(Delete(A, \beta)\langle [root(t), (t_2, ..., t_k)], \mathcal{G}\rangle)), \\
\text{where, } t'_i &= struct(Delete(A, \beta)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&\quad = t_i, \text{ otherwise}
\end{aligned}
$$

We now give the definition for $\mathcal{G}'$ (for both cases).

$$
\begin{aligned}
&\mathcal{G}' = normalize([\![S, P']\!]), \text{ where,} \\
&P' = P, \text{ if } (\beta \text{ corresponds to } p) \wedge (head(p) = A) \wedge (tail(p) = B^*), \\
&\quad = P \cup P_1 - P_2, \text{ otherwise, where,} \\
&\qquad P_1 = \{p_1 \mid (head(p_1) = A) \wedge (\exists p \in P) \wedge (head(p) = A) \\
&\qquad\quad \wedge (\beta \text{ corresponds to } p) \\
&\qquad\quad \wedge ((((\beta = (\equiv \beta_1)) \vee (\beta = (\not\equiv \beta_1))) \wedge (tail(p) = B_1 ... B_m ... B_k) \\
&\qquad\qquad \wedge (root(\beta_1) = B_m) \wedge (tail(p_1) = B_1 ... B_{m-1} B_{m+1} ... B_k)) \\
&\qquad\quad \vee ((\beta = (\notin (C_1, ..., C_m))) \wedge (tail(p_1) \sqsubseteq tail(p)) \wedge \\
&\qquad\qquad (\forall C \in tail(p)\ ((C \in (C_1, ..., C_m) \Rightarrow (C \in tail(p_1))) \\
&\qquad\qquad\qquad \wedge (C \notin (C_1, ..., C_m) \Rightarrow (C \notin tail(p_1)))))))\}, \\
&\qquad P_2 = \{\ \}, \text{ if } (\beta = (\equiv \beta_1)) \vee (\beta = (\not\equiv \beta_1)), \\
&\qquad\quad = \{p \mid (p \in P) \wedge (head(p) = A) \wedge (\beta \text{ corresponds to } p)\}, \\
&\qquad\qquad \text{if } (\beta = (\notin (C_1, ..., C_m))).
\end{aligned}
$$

□

4.3.5 Substitute

The *Substitute* operator,[8] allows trees in an instance to be replaced by another tree. Although this can, in most cases, be done in terms of *Find*s, *Insert*s and *Delete*s, it is useful to have an operator that allows changes to be made to an instance in a single step.

Let $\langle t_1, \mathcal{G}_1 \rangle$ and $\langle t_2, \mathcal{G}_2 \rangle$ be two instances. Let β be a structure pattern over some $B \in var(\mathcal{G}_1)$ and let $root(t_2) = B$. Let α be either $*$ or a variable $A \in var(\mathcal{G}_1)$ such that for some $p \in P_1$, $head(p) = A$ and $B \in tail(p)$. Then, $Substitute(\alpha, \beta)(\langle t_1, \mathcal{G}_1 \rangle, \langle t_2, \mathcal{G}_2 \rangle)$ looks for trees in t_1 that match with β and replaces them with t_2. Further, if α is a variable A, then each tree being replaced must be a subtree of a tree with A as the root.

Example 4.15 *Replace all occurrences of* `'Data Base'` *in the* `TITLE` *trees in the instance* d_3 (Figure 9(a)) *with* `'Database'`.

Substitute(`TITLE,[WORD,([B,*],[B,*])]`)
((*Find*(`WORD, ([WORD,Data], [WORD,Base]), B`)d_3),
⟨`[WORD,Database],⟦WORD, {WORD ⟶ String}⟧`⟩).

In this example, since `WORD` subtrees occur only in `TITLE` trees, we could have expressed the same query by replacing the variable `TITLE` (the first argument of *Substitute*) by a $*$. Also, in this example a list of subtrees is being replaced by a single `WORD` tree. If the query involved replacing a list of subtrees with another list of subtrees the expression would be slightly more complex involving the *Replace* operator. ■

Example 4.16 *Change the book-title* `'Fundamental Algorithms'`, *in the instance* d_1, *to the string* `'The Art of Computer Programming'`.

Let $\mathcal{G}_2$ represent the scheme `⟦TITLE, {TITLE ⟶ String}⟧`

Substitute(`BOOK, [TITLE,([WORD,Fundamental],[WORD,Algorithms])]`)(d_1,
⟨`[TITLE,'The Art Of Computer Programming'`[9]`], `$\mathcal{G}_2$⟩). ■

[8] not to be confused with the function *substitute*

[9] The quotes are given only to indicate to the reader that the value of the `TITLE` node is a string.

Definition 4.10 $Substitute(\alpha, \beta)(\langle t_1, \mathcal{G}_1\rangle, \langle t_2, \mathcal{G}_2\rangle) = \langle t', \mathcal{G}'\rangle$, where

$$
\begin{aligned}
t' &= [], \text{ if } (t_1 = []) \wedge (t_2 = []) \\
&= t_1, \text{ if } (t_2 = []) \wedge (((\alpha = *) \wedge (match(t_1, \beta) = false)) \vee (\alpha \neq *)) \\
&= t_1, \text{ if } (t_1 = []) \wedge (((S_1 \neq S_2) \wedge (\alpha = *)) \vee (\alpha \neq *)) \\
&= t_2, \text{ if } (\alpha = *) \wedge (match(t_1, \beta) = true) \\
&= t_1, \text{ if } (((match(t_1, \beta) = false) \wedge (\alpha = *)) \vee (\alpha \neq *)) \\
&\qquad \wedge ((has\text{-}leaf(t_1)) \vee (subtrees(t_1) = ())) \\
&= [root(t_1), (t''_{11}, ..., t''_{1k})], \text{ otherwise,} \\
&\quad \text{where, } t''_{1i} = t'_{1i}, \text{ if } (\alpha = *) \vee ((\alpha \neq *) \wedge \\
&\qquad ((match(t_{1i}, \beta) = false) \vee (root(t_1) \neq \alpha))) \\
&\qquad = t_2, \text{ otherwise} \\
&\qquad\quad \text{where, } t'_{1i} = struct(Substitute(\alpha, \beta)(\langle t_{1i}, \mathcal{G}_{1i}\rangle, \langle t_2, \mathcal{G}_2\rangle)), \\
&\qquad\qquad \text{if } ((\alpha \neq *) \wedge (reachable(\alpha, root(t_{1i}), P))) \\
&\qquad\qquad \vee ((\alpha = *) \wedge (reachable(B, root(t_{1i}), P))) \\
&\qquad\quad = t_{1i}, \text{ otherwise}
\end{aligned}
$$

$\mathcal{G}' = normalize([\![S_1, P_1 \cup P_2]\!])$ □

4.3.6 Rename

This operator renames non-leaf nodes in an instance. There are several different cases for this operator. In the most general case, all occurrences of a given node are renamed. In the other cases the renaming of nodes is restricted to nodes that occur in trees that correspond to a given production. While the *Rename* operator is useful by itself, i.e., for the main purpose of changing the names of nodes, *Rename* often serves a more useful function of restricting the application of operators to trees that correspond to only some productions. For example, suppose that an instance has the productions $A \longrightarrow BC$ and $A \longrightarrow BCD$ and that we want to retrieve all the B trees from trees that correspond to the first production. We can use the *Rename* operator to change the B nodes of trees that correspond to the second production to, say E, perform the *Retrieve* operation and then rename the E nodes back to B.

Rename has three parameters and has different actions corresponding to the types of the parameters as described below. Let $\langle t, \mathcal{G}\rangle$ be a list-structure instance and let X, Y and p be the three parameters. Then $Rename(X, Y, p)\langle t, \mathcal{G}\rangle$ renames nodes in t as follows:

1. $X = A$ and $Y = B$ where A and B are variables, $A \in var(\mathcal{G})$, and

   (a) $p = *$, and there are no productions in P that contain both A and B in the right hand side. *Rename* replaces every A node in t by B; or

   (b) $p = C \longrightarrow A^*$ and $p \in P$. In each tree that corresponds to p, the children of the tree are renamed to B; or

   (c) $p = C \longrightarrow A_1 ... A_k$, $p \in P$, $A \in tail(p)$, and $B \notin tail(p)$. In each tree that corresponds to p, the child with label A is renamed to B; or

   (d) $p = C \longrightarrow *$, for some $q \in P$, $head(q) = C$ and $A \in tail(q)$, and both A and B do not occur in the tail of any production whose head is C. In every tree that has C as the root, any child nodes that have A as the label are renamed to B. This case is a generalization of the previous two cases.

Example 4.17 *Rename the variable* `A` *to* `AUTHOR` *in the instance* d_1.

$Rename$`(A,AUTHOR,*)`d_1. ■

The above example corresponds to Case 1(a), the most general case of *Rename* where all occurrences of a variable are renamed. The remaining *Rename* cases allow renaming to be restricted to nodes of trees corresponding to specific productions. Cases 1(b) and 1(c) are useful when the same variable occurs in the tail of several productions and the variable is to be renamed only in trees corresponding to one of these productions. Case 1(d) is useful when the renaming is to be restricted to trees that have a specific root node. The following examples illustrate this.

Example 4.18 *Rename the* `TITLE` *nodes corresponding to all the* `VOL` *subtrees in* d_2 *to* `VOL-TITLE`.

$Rename$`(TITLE, VOL-TITLE, VOL` $\longrightarrow$ `TITLE NO.)`d_2 ■

Example 4.19 *Replace all occurrences of* `'Data Base'` *in the titles of the* `BOOK` *trees in* d_3 *with* `'Database'`.

This query is a variation of the query in Example 4.15, where the replacement is to take place only in those `TITLE` trees that are subtrees of `BOOK` trees. We can rename the `TITLE`

nodes of the VOL trees to VOL-TITLE, as in the previous example, and then rename the WORD nodes in VOL-TITLE trees to V-WORD as follows.

Rename(WORD, V-WORD, VOL-TITLE $\longrightarrow$ WORD*).

We could then apply the expression in Example 4.15 and then Rename VOL-TITLE and V-WORD nodes back to their original names. ■

The next two *Rename* cases are useful when trees corresponding to a list production are to be renamed to correspond to an aggregate production and vice-versa.

2. $X = B$, $Y = (B_1, ..., B_m)$, the B_i's are distinct, and $p \in P$ where $p = A \longrightarrow B^*$. *Rename* looks for trees that correspond to p and that have exactly m children. The children of each such tree are renamed according to Y.

3. $X = (B_1, ..., B_m)$, $Y = B$, and $p \in P$ where $p = A \longrightarrow B_1...B_m$. *Rename* finds trees that correspond to p and in each tree renames all of its children to B.

Example 4.20 *Rename the first and second* A *nodes in trees, in* d_1*, that correspond to the production* AUTHOR $\longrightarrow$ A*, *and that have exactly two* A *subtrees to* AUTHOR1 *and* AUTHOR2*, respectively.*

Rename(A,(AUTHOR1, AUTHOR2),AUTHORS $\longrightarrow$ A*)d_1.

Trees that have less than or more than two A subtrees are unchanged by the above operation. Figure 12 shows the result of this operation. ■

Definition 4.11 $Rename(X, Y, p)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$, where t' and $\mathcal{G}'$ are defined as follows.

Case 1: $X = A$ and $Y = B$.

Case 1(a): $p = *$.

$$\begin{aligned} t' &= t, \text{ if } (t = []) \vee ((root(t) \neq A) \wedge ((has\text{–}leaf(t)) \vee (subtrees(t) = ()))) \\ &= [B, subtrees(t)], \text{ if } (root(t) = A) \wedge (has\text{–}leaf(t)) \\ &= [root(t), (t'_1, ..., t'_k)], \text{ if } (root(t) \neq A) \\ &= [B, (t'_1, ..., t'_k)], \text{ otherwise,} \end{aligned}$$

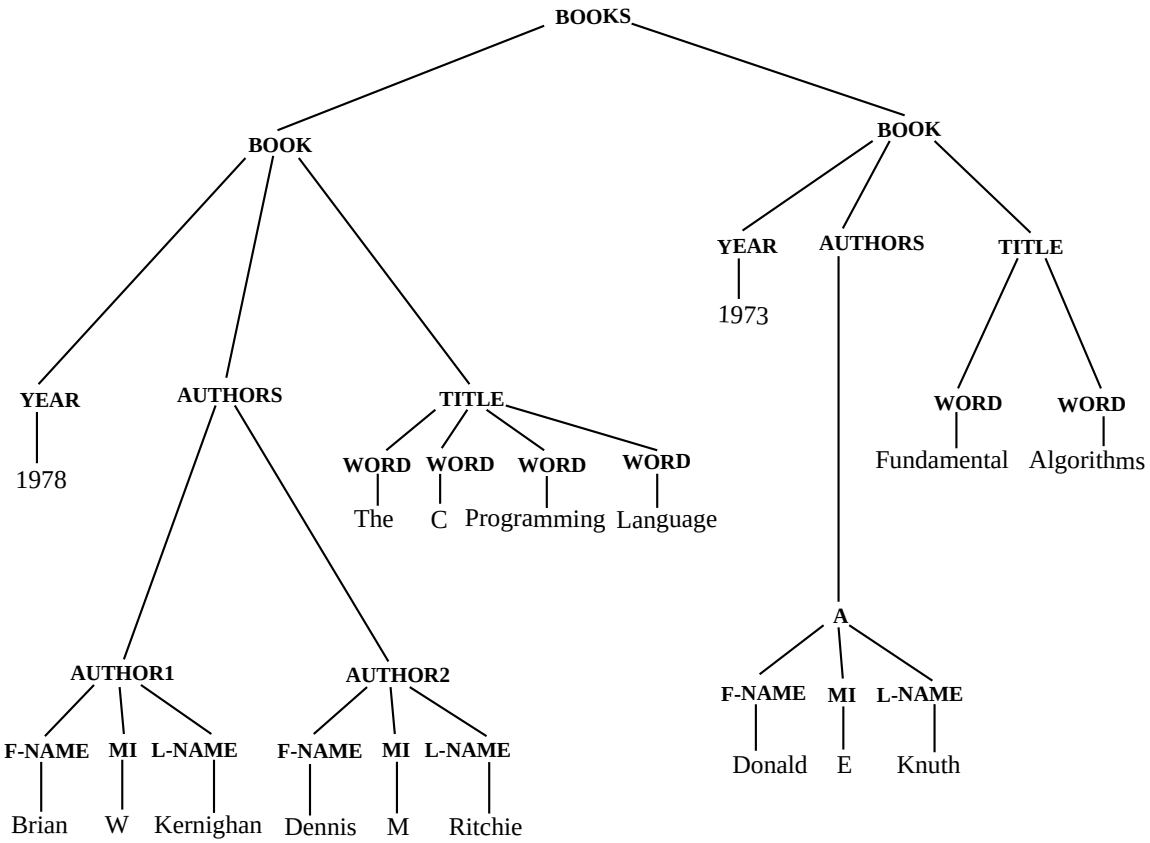


Figure 12: The result of a *Rename* operation.

$$\text{where, } t'_i = struct(Rename(A,B,p)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P))$$
$$= t_i, \text{ otherwise.}$$

$$\mathcal{G}' = normalize([\![S', P']\!]),$$
$$\text{where, } P' = \{q \mid (\exists p_1 \in P) \wedge (q = substitute(A, B, p_1))\}$$
$$S' = S, \text{ if } S \neq A$$
$$= B, \text{ if } S = A$$

Cases 1(b), 1(c) and 1(d): $p = C \longrightarrow A^*$, $p = C \longrightarrow A_1 ... A_k$, or $p = C \longrightarrow *$.

$$t' = t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ())$$
$$= [root(t), (t'_1, ..., t'_k)], \text{ if } (root(t) \neq C) \vee ((p \neq C \longrightarrow *) \wedge (satisfies(t,p) = false)),$$
$$= [root(t), (t''_1, ..., t''_k)], \text{ otherwise,}$$
$$\text{where, } t''_i = [B, subtrees(t'_i)], \text{ if } (root(t_i) = A)$$
$$= t'_i, \text{ otherwise}$$

$$\begin{aligned}
\text{where, } t'_i &= struct(Rename(A,B,p)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(C, root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}$$

$\mathcal{G}' = normalize([\![S, P']\!])$, where, $P' = P - P_1 \cup P_2 \cup P_3$, where,

$$\begin{aligned}
P_1 &= \{p_1 \mid (p_1 \in P) \wedge (((p \neq C \longrightarrow *) \wedge (p_1 = p)) \\
&\qquad \vee((p = C \longrightarrow *) \wedge (head(p_1) = C) \wedge (A \in tail(p_1))))\} \\
P_2 &= \{p_1 \mid (\exists q \in P) \wedge (((p \neq C \longrightarrow *) \wedge (p = q)) \\
&\qquad \vee((p = C \longrightarrow *) \wedge (head(q) = C))) \wedge (head(p_1) = C) \\
&\qquad \wedge(tail(p_1) = tail(substitute(A, B, q)))\} \\
P_3 &= \{p_1 \mid (\exists q \in P) \wedge (head(q) = A) \wedge (head(p_1) = B) \\
&\qquad \wedge(tail(p_1) = tail(q) \\
&\qquad\qquad \text{if } ((A \neq C) \vee ((A = C) \wedge (((p = C \longrightarrow *) \wedge (C \notin tail(q))) \\
&\qquad\qquad\qquad \vee((p \neq C \longrightarrow *) \wedge (q \neq p))))) \\
&\qquad\quad = tail(substitute(A, B, q)), \text{ otherwise.)}\ \}
\end{aligned}$$

<u>Case 2</u>: $X = B$, $Y = (B_1, ..., B_m)$ and $p = A \longrightarrow B^*$.

$$\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p) = false) \vee (k \neq m) \\
&= [root(t), (s'_1, ..., s'_m)], \text{ otherwise, where, } s'_i = [B_i, subtrees(t'_i)] \\
\text{where, } t'_i &= struct(Rename(X, Y, p)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}' &= normalize([\![S, P \cup \{A \longrightarrow B_1...B_m\} \cup P' \cup P'']\!]), \\
&\quad \text{where, } P' = \{p_1 \mid (\exists q \in P) \wedge (tail(p_1) = tail(q)) \\
&\qquad\qquad \wedge(head(p_1) \in (B_1, ..., B_m)) \wedge (head(q) = B)\} \\
&\quad \text{and } P'' = \{p_1 \mid (head(p_1) \in (B_1, ..., B_m)) \\
&\qquad\qquad \wedge(tail(p_1) = (B_1, ..., B_m))\}, \text{ if } A = B \\
&\qquad\quad = \{\ \}, \text{ otherwise.}
\end{aligned}$$

<u>Case 3</u>: $X = (B_1, ..., B_m)$, $Y = B$ and $p = A \longrightarrow B_1...B_m$.

$$\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p) = false), \\
&= [root(t), (s'_1, ..., s'_m)], \text{ otherwise, where, } s'_i = [B, subtrees(t'_i)]
\end{aligned}$$

$$\text{where, } t'_i = struct(Rename(X, Y, p)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P))$$
$$= t_i, \text{ otherwise}$$
$$\mathcal{G}' = normalize([\![S, P \cup \{A \longrightarrow B^*\} - \{p\} \cup P_1]\!])$$
$$\text{where, } P_1 = \{p_1 \mid (\exists q \in P) \wedge (head(q) \in (B_1, ..., B_m)) \wedge (head(p_1) = B)$$
$$\wedge (tail(p_1) = tail(q), \text{ if } (q \neq p)$$
$$= B^*, \text{ otherwise})\}.$$

□

4.3.7 Replace

The *Replace* operator is used for changing the structure of instances by adding or deleting interior nodes. However, the *Replace* operator, like *Rename* is more often used in conjunction with other operators than by itself. For example, in queries that involve the *Find* operation, *Replace* is used in getting rid of the interior nodes introduced by the *Find* operation. Also, like the *Rename* operator, *Replace* can be used to restrict the application of operators to trees that correspond to specific productions. The operators *Rename* and *Replace* change the structure of instances without affecting the 'information content' of the instances. They can hence be considered *restructuring* operators. Other types of restructuring operators were considered, in [11] and [1], in the context of the Format model.

Replace has two parameters and has different actions depending upon the types of these parameters. The first parameter p_1 is a production and the second parameter p_2 is either a production or is of the form $C \longrightarrow *$. If $\langle t, \mathcal{G}\rangle$ is an instance and p_1 and p_2 are of one of the different forms given below, then $Replace(p_1, p_2)\langle t, \mathcal{G}\rangle$ will look for trees in t that correspond to p_1 and change them as described below.

1. This *Replace* case is used to introduce interior nodes in an instance. Both p_1 and p_2 are productions, $p_1 \in P$, $head(p_2) \notin tail(p_1)$, and

   (a) $p_1 = A \longrightarrow B^*$, and $p_2 = C \longrightarrow B$ or $p_2 = C \longrightarrow B^*$.

   i. If $p_2 = C \longrightarrow B$, then for each tree s in t that corresponds to p_1, each of its subtrees s_i is replaced by a tree whose root is C and whose (only) subtree is s_i.

   ii. If $p_2 = C \longrightarrow B^*$, then for each tree s that corresponds to p_1, its list of subtrees is replaced by a single tree whose root node is C and whose subtrees are the subtrees of s.

(b) $p_1 = A \longrightarrow \beta$, where β is a constant or a type, and $p_2 = C \longrightarrow \beta$. The subtree of each tree s that corresponds to p_1 is replaced by a tree whose root node is C and whose only subtree is the subtree of s.

(c) $p_1 = A \longrightarrow C_1...C_k$, $p_2 = C_l \longrightarrow C_m...C_n$, $(C_m, ..., C_n) \subseteq (C_1, ..., C_k)$, and $C_l \notin tail(p_1)$. For each tree s that corresponds to p_1, the part of its list of subtrees corresponding to nodes $C_m, ..., C_n$ is replaced by a tree whose root node is C_l and whose subtrees are the subtrees being replaced.

Example 4.21 *Change the* MI *attribute of* A *trees, in* d_1*, to have two attributes, so that the first attribute is the initial and the second a* '.'.

$$Insert(\texttt{M,([MI,*],*)})(Replace(\texttt{A} \longrightarrow \texttt{F-NAME MI L-NAME, M} \longrightarrow \texttt{MI})d_1,$$
$$\langle \texttt{[DOT, .],[[DOT, \{DOT} \longrightarrow \texttt{.\}]]}\rangle).$$

This example corresponds to Case 1(c). The MI part of A trees are changed so that each MI tree is replaced by a tree labeled M, that has two subtrees, labeled MI and DOT, where the first subtree is the original MI tree and the second has a '.' as its value. Figure 13 shows the result of this query. ■

2. This replace case, removes interior nodes and is particularly useful in getting rid of interior nodes introduced by other operators such as *Find*. $p_1 \in P$, and p_2 is either a production in P or of the form $C \longrightarrow *$ where for some $p \in P$, $head(p) = C$. Also, $head(p_2) \in tail(p_1)$, and p_1 and p_2 must be of one of the following forms.

   (a) $p_1, p_2 \in P$ and

   i. $p_1 = A \longrightarrow C^*$ and $p_2 = C \longrightarrow B^*$, or

   ii. $p_1 = A \longrightarrow C^*$ and $p_2 = C \longrightarrow B$, or

   iii. $p_1 = A \longrightarrow C$ and $p_2 = C \longrightarrow B^*$, or

   iv. $p_1 = A \longrightarrow C$, $p_2 = C \longrightarrow b$, where b is a constant or a type.

   Each tree s, in t, that corresponds to p_1 and whose subtrees all correspond to p_2, is replaced by a tree with root node A and whose subtrees are (for cases i and ii, the concatenation of) the subtrees of the subtrees of s.

   For cases i and ii if $C = B$ then for each tree s in t that satisfies p_1, those subtrees that correspond to p_2 are replaced by their lists of subtrees. In other words, it is not necessary that all the subtrees of s correspond to p_2.

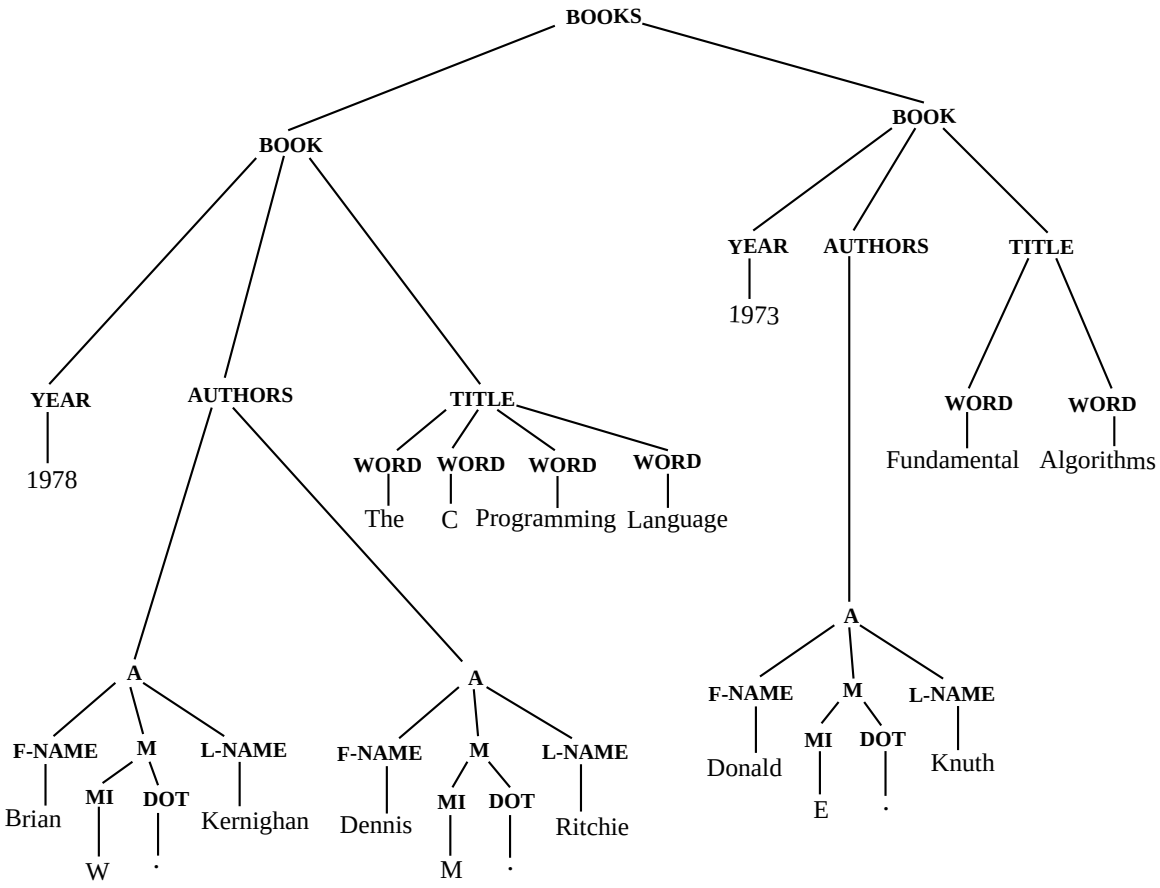


Figure 13: The result of a query involving the application of a *Replace* operation.

(b) $p_1, p_2 \in P$, $p_1 = A \longrightarrow C_1 ... C_m$, $p_2 = C_l \longrightarrow D_1 ... D_n$, $C_l \in tail(p_1)$, and $(\forall D_i \in tail(p_2), C_j \in tail(p_1), (j \neq l) \Rightarrow (D_i \neq C_j))$. For each tree s that corresponds to p_1, if the subtree of s with root node C_l corresponds to p_2, then that subtree is replaced by its list of subtrees.

(c) p_1 is a production in P of the form $A \longrightarrow C$, and $p_2 = C \longrightarrow *$. This case is a generalization of Cases 2(a) and (b) when p_1 is of the form $A \longrightarrow C$. The subtree of each tree that corresponds to p_1 is replaced by its (the subtree's) subtrees. This case is useful for getting rid of the extra interior nodes introduced by the *Find* command. Although this can be done in terms of the previous *Replace* cases, it is convenient to be able to get rid of the interior nodes with a single operation when there are several productions that have C as the head.

Example 4.22 *Change the* YEAR *attribute of the book* 'The C Programming Language' *to* 1974.

Replace(BOOK $\longrightarrow$ B, B $\longrightarrow$ *)
(*Substitute*(B,[YEAR,*])
(*Find*(BOOK, [TITLE (= (The C Programming Language))], B)d_1,
$\langle$[YEAR,1974],$[\![$YEAR, {YEAR $\longrightarrow$ Integer}$]\!]\rangle$)).

The purpose of the *Replace* operation in this example was to eliminate the B nodes that were introduced as a result of the *Find* operation. Since, there are two different productions, (i.e., B $\longrightarrow$ YEAR AUTHORS TITLE and B $\longrightarrow$ YEAR AUTHORS VOL TITLE), that B trees could correspond to, the more general form of *Replace*, i.e., Case 2(c) was used. ■

Definition 4.12 $Replace(p_1, p_2)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$, where, t' and $\mathcal{G}'$ are defined as follows:

Case 1: $head(p_2) \notin tail(p_1)$

Case 1(a) and 1(b):

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee ((has\text{-}leaf(t)) \vee (subtrees(t) = ())) \wedge (satisfies(t, p_1) = false) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false), \\
&= [root(t), ([C, subtrees(t)])], \text{ if } (satisfies(t, p_1)) \wedge (has\text{-}leaf(t)) \\
&= [root(t), (s'_1, ..., s'_k)], \text{ if } (satisfies(t, p_1)) \wedge (p_2 = C \longrightarrow B), \\
&\quad \text{where } s'_i = [C, (t'_i)] \\
&= [root(t), ([C, (t'_1, ..., t'_k)])], \text{ otherwise,} \\
\text{where, } t'_i &= struct(Replace(p_1, p_2)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(head(p_1), root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S, P - \{p_1\} \cup \{p_2\} \cup P']\!]), \text{ where,} \\
P' &= \{A \longrightarrow C^*\}, \text{ if } p_2 = C \longrightarrow B, \\
&= \{A \longrightarrow C\}, \text{ otherwise.}
\end{aligned}
$$

Case 1(c):

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false), \\
&= [root(t), (t''_1, ..., t''_{k-n+m})], \text{ otherwise, where} \\
&\quad t''_i = t'_i, \text{ if } i < m \\
&\quad\quad = t'_{i+n-m}, \text{ if } i > m \\
&\quad\quad = [C_l, (t'_m, ..., t'_n)], \text{ otherwise,} \\
&\text{where, } t'_i = struct(Replace(p_1, p_2)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(head(p_1), root(t_i), P)) \\
&\quad\quad = t_i, \text{ otherwise.}
\end{aligned}
$$

$$
\mathcal{G}' = normalize([\![S, P - \{p_1\} \cup \{p_2, A \longrightarrow C_1 ... C_{m-1} C_l C_{n+1} ... C_k\}]\!])
$$

Case 2: $head(p_2) \in tail(p_1)$

Case 2(a):

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false) \vee \\
&\quad ((satisfies(t, p_1)) \wedge (head(p_2) \notin tail(p_2)) \\
&\quad\quad \wedge (\exists t_i \in subtrees(t) \mid (satisfies(t_i, p_2) = false))) \\
&= [root(t), subtrees(t_1)], \text{ if } (tail(p_2) \in \mathcal{T}) \vee (tail(p_2) \in ,) \\
&= [root(t), concat(l_1, ..., l_k)], \text{ otherwise, where,} \\
&\quad l_i = (t'_i), \text{ if } (satisfies(t_i, p_2) = false) \\
&\quad\quad = subtrees(t'_i), \text{ otherwise.} \\
&\text{where, } t'_i = struct(Replace(p_1, p_2)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(head(p_1), root(t_i), P)) \\
&\quad\quad = t_i, \text{ otherwise.}
\end{aligned}
$$

Case 2(b):

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false) \vee (satisfies(t_l, p_2) = false) \\
&= [root(t), (s'_1, ..., s'_{m+n-1})], \text{ otherwise, where} \\
&\quad s'_i = t'_i, \text{ if } (i < l) \\
&\quad\quad = t'_{l,i-l+1}, \text{ if } (i \geq l) \wedge (i \leq l + n - 1) \\
&\quad\quad = t'_{i-n+1}, \text{ if } (i > l + n - 1) \\
&\text{where, } t'_i = struct(Replace(p_1, p_2)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(head(p_1), root(t_i), P)) \\
&\quad\quad = t_i, \text{ otherwise.}
\end{aligned}
$$

$\mathcal{G}'$ for both Cases 2(a) and 2(b) is determined as follows:

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S, P - P_1 - P_2 \cup P_3]\!]), \text{ where,} \\
&\quad P_1 = \{p_1\}, \text{ if } (\forall q \in P\ ((head(q) = head(p_2)) \Rightarrow (q = p_2))) \\
&\quad\quad = \{\}, \text{ otherwise,} \\
&\quad P_2 = \{p_2\}, \text{ if } (p_1 \neq A \longrightarrow C^*) \wedge (head(p_2) \neq S) \\
&\quad\quad\quad \wedge (\forall q \in P (head(p_2) \in tail(q)) \Rightarrow (q = p_1)), \\
&\quad\quad = \{\}, \text{ otherwise,} \\
&\quad P_3 = \{A \longrightarrow B^*\}, \text{ for Case 2a(ii),} \\
&\quad\quad = \{A \longrightarrow C_1 ... C_{l-1} D_1 ... D_n C_{l+1} ... C_m\}, \text{ for Case 2(b),} \\
&\quad\quad = \{A \longrightarrow tail(p_2)\}, \text{ for all remaining cases.}
\end{aligned}
$$

Case 2(c):

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false) \\
&= [root(t), subtrees(t'_1)], \text{ otherwise,} \\
&\text{where, } t'_1 = struct(Replace(p_1, p_2)\langle t_1, \mathcal{G}_1\rangle), \text{ if } (reachable(head(p_1), root(t_1), P)) \\
&\quad\quad = t_1, \text{ otherwise.}
\end{aligned}
$$

$\mathcal{G}' = normalize([\![S, P']\!])$, where,

$$
\begin{aligned}
P' &= P - \{p_1\} \cup \{q \mid (\exists p \in P) \wedge (head(p) = head(p_2)) \\
&\quad\quad \wedge (head(q) = head(p_1)) \wedge (tail(q) = tail(p))\}.
\end{aligned}
$$

□

4.3.8 Reorder

This operator looks for trees corresponding to a given aggregate production and rearranges the subtrees in a different order to correspond to a new aggregate production.

Example 4.23 *Rearrange the attributes of* A *nodes in* d_1 *so that* L-NAME *is the first attribute and* F-NAME *is the second.*

$$Reorder(\texttt{A} \longrightarrow \texttt{F-NAME MI L-NAME, A} \longrightarrow \texttt{L-NAME F-NAME MI})$$
$$(Reorder(\texttt{A} \longrightarrow \texttt{F-NAME L-NAME, A} \longrightarrow \texttt{L-NAME F-NAME})d_1).$$ ■

Definition 4.13 Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let p_1 and p_2 be productions such that $p_1 \in P$, $p_1 = A \longrightarrow B_1 ... B_m$, and $p_2 = A \longrightarrow C_1 ... C_m$, where each $C_i = B_j$ for some j. Then, $Reorder(p_1, p_2)\langle t, \mathcal{G} \rangle = \langle t', \mathcal{G}' \rangle$, where,

$$\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, p_1) = false), \\
&= [root(t), (s'_1, ..., s'_m)], \text{ otherwise, where} \\
&\quad s'_i = t'_j, \text{ where } j \text{ is such that } C_i = B_j, \\
\text{and } t'_i &= struct(Reorder(p_1, p_2)\langle t_i, \mathcal{G}_i \rangle), \text{ if } (reachable(head(p_1), root(t_i), P)) \\
&= t_i, \text{ otherwise.}
\end{aligned}$$

$$\mathcal{G}' = normalize([\![S, P - \{p_1\} \cup \{p_2\}]\!])$$ □

4.3.9 Group

The *Group* operator groups subtrees of trees corresponding to a given list production, such that the subtrees in a group agree on certain attributes. This, of course, implies that the subtrees of the trees corresponding to the list production must correspond to aggregate productions. This is very similar to the 'nest' operator provided in query languages for the nested relational model.

Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let $A \longrightarrow B^*$ be a production in $\mathcal{G}$ and let $D \in tail(p)$, $(D \neq A$ and $D \neq B)$, for some aggregate production p such that $head(p) = B$. Also, suppose that for any production q in $\mathcal{G}$, if $head(q) = B$, then q is an aggregate production. Let E be a variable not in $var(\mathcal{G})$. Then $Group(A \longrightarrow B^*, D, E)\langle t, \mathcal{G} \rangle$ does the following. For

each tree in t that satisfies the production $A \longrightarrow B^*$, its subtrees are formed into groups that agree on the values of all of their attributes except D. Each group is represented by a single tree with root node B, and with the same list of children as those of the trees in the group, and with the subtrees determined as follows. The subtrees that correspond to non-D attributes are the same as those of the trees in that group. The subtree corresponding to D has as its list of subtrees the D subtrees of all the trees in the group, but with the root nodes labeled E instead of D.

Example 4.24 *Group all volumes of each book in d_2 so that the* VOL *attribute of each book is replaced by a* VOLUMES *attribute that has a subtree labeled* VOL *for each volume in the group. The attributes of each* VOL *subtree must include the attributes of the original* VOL *subtree and the* YEAR *attribute of the* BOOK *subtree.*

Group(BOOKS $\longrightarrow$ BOOK*, VOLUMES, VOL)
(*Replace*(VOLUMES $\longrightarrow$ YEAR VOL, VOL $\longrightarrow$ NO.)
(*Replace*(VOLUMES $\longrightarrow$ YEAR VOL, VOL $\longrightarrow$ TITLE NO.)
(*Replace*(BOOK $\longrightarrow$ AUTHORS YEAR VOL TITLE, VOLUMES $\longrightarrow$ YEAR VOL)
(*Reorder*(BOOK $\longrightarrow$ YEAR AUTHORS VOL TITLE,
BOOK $\longrightarrow$ AUTHORS YEAR VOL TITLE)d_2)))).

Figure 14 shows the result of the above query. ■

Definition 4.14 $Group(A \longrightarrow B^*, D, E)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$, where,

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, A \longrightarrow B^*) = false), \\
&= [root(t), append(t''_1, l_2)], \text{ otherwise, where,} \\
&\qquad t''_1 = t'_1, \text{ if } (D \notin children(t_1)) \\
&\qquad\quad = [root(t_1), (t''_{11}, ..., t''_{1m})], \text{ otherwise, where, } (t_{11}, ..., t_{1m}) = subtrees(t_1), \\
&\qquad\qquad \text{and } t''_{1j} = t'_{1j}, \text{ if } root(t_{1j}) \neq D \\
&\qquad\qquad\quad = [D, concat(l'_1, ..., l'_k)], \text{ otherwise, where,} \\
&\qquad\qquad\qquad l'_i = (), \text{ if } (children(t_i) \neq children(t_1)) \\
&\qquad\qquad\qquad\qquad \vee (\exists C \in children(t_i) \mid (C \neq D) \wedge (t_i[C] \neq t_1[C])) \\
&\qquad\qquad\qquad\quad = ([E, subtrees(t_i[D]')]), \text{ otherwise,} \\
&\text{and } l_2 = subtrees(struct(Group(A \longrightarrow B^*, D, E)\langle [root(t), l_3], \mathcal{G}\rangle)), \text{ where,}
\end{aligned}
$$

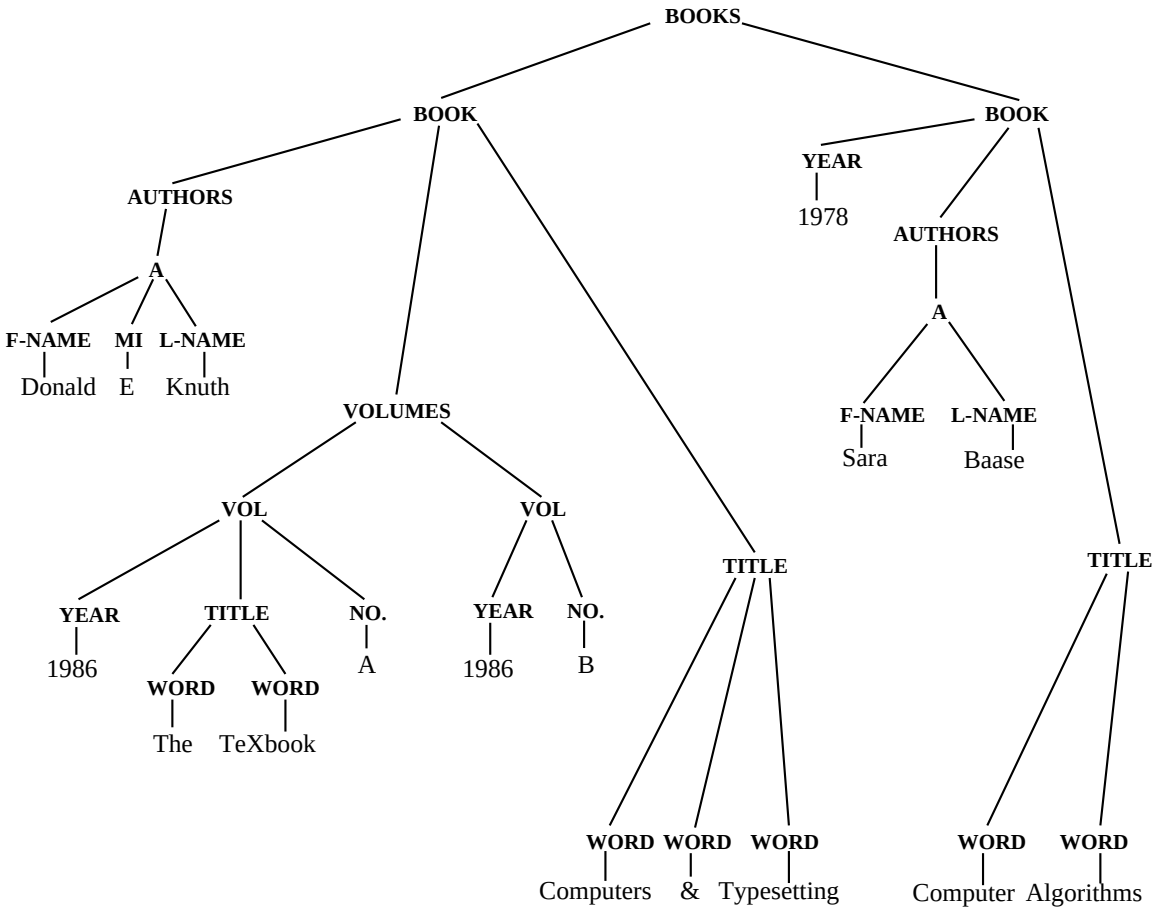


Figure 14: The result of restructuring an instance using the *Group* operator.

$$
\begin{aligned}
l_3 &= (t_2, ..., t_k), \text{ if } (D \notin children(t_1)), \\
&= subtrees(t) - \{t_i \in subtrees(t) \mid (children(t_i) = children(t_1)) \\
&\qquad \wedge (\forall C \in children(t_i)\ ((C = D) \vee (t_i[C] = t_1[C])))\}^{10}, \text{ otherwise.}
\end{aligned}
$$

$$
\begin{aligned}
\text{where, } t_i' &= struct(Group(A \longrightarrow B^*, D, E)\langle t_i, \mathcal{G}_i \rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&= t_i, \text{ otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}' &= [\![S, P \cup \{D \longrightarrow E^*\} - P_1 \cup P_2]\!], \text{ where} \\
P_1 &= \{p \in P \mid (head(p) = D)\}, \text{ if } (D \neq S) \wedge (B \neq S) \\
&\qquad \wedge (\forall q \in P(B \in tail(q)) \Rightarrow (q = A \longrightarrow B^*)) \\
&\qquad \wedge (\forall q \in P(D \in tail(q)) \Rightarrow (head(q) = B)), \\
&= \{\}, \text{ otherwise,} \\
P_2 &= \{p \mid (\exists q \in P) \wedge (head(q) = D) \wedge (head(p) = E) \wedge (tail(p) = tail(q))\}. \ \square
\end{aligned}
$$

[10] The result of this difference operation between a list and a set is the list obtained by removing from the first argument, (i.e., the list), those elements that also appear in the second (i.e., the set).

4.3.10 UnGroup

UnGroup does the opposite of the *Group* operator by splitting the group represented by each subtree of a tree corresponding to a given list production into several trees.

Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let $A \longrightarrow B^*$ be a production in $\mathcal{G}$, and let $D \in tail(p)$, $(D \neq A)$, for some aggregate production p such that $head(p) = B$. Let there be exactly one production, say q, in P such that $head(q) = D$ and let $tail(q) = E^*$ for some $E \in var(\mathcal{G})$. Also, suppose that for any production q in $\mathcal{G}$, if $head(q) = B$, then q is an aggregate production. Then, $UnGroup(A \longrightarrow B^*, D \longrightarrow E^*)\langle t, \mathcal{G} \rangle$ looks for trees corresponding to $A \longrightarrow B^*$. For each such tree s, it breaks up each subtree s_i into several trees $(s_{i_1}, ..., s_{i_n})$ so that all the s_{ij}'s have the same list of children as s_i. The subtrees of each s_{ij} corresponding to the non-D attributes are the same as those of s_i. The subtree corresponding to D has as its subtrees, the subtrees of the jth subtree of the D node of s_i.

Performing $UnGroup(A \longrightarrow B^*, D \longrightarrow E^*)$ on an instance, immediately after a $Group(A \longrightarrow B^*, D, E)$, will give back for each tree corresponding to $A \longrightarrow B^*$, all of its original subtrees, but not necessarily in the original order.

Example 4.25 *Restructure the instance d_2 by grouping together all the books authored by each author.*

Rename(B,BOOKS,*)(*Group*(BIBLIO $\longrightarrow$ ITEM*,B,BOOK)
(*Rename*(BOOK,ITEM,*)(*Rename*(BOOKS,BIBLIO,*)
(*Replace*(BOOK $\longrightarrow$ AUT YEAR VOL TITLE, B $\longrightarrow$ YEAR VOL TITLE)
(*Replace*(BOOK $\longrightarrow$ AUT YEAR TITLE, B $\longrightarrow$ YEAR TITLE)
(*Reorder*(BOOK $\longrightarrow$ YEAR AUT VOL TITLE, BOOK $\longrightarrow$ AUT YEAR VOL TITLE)
(*Reorder*(BOOK $\longrightarrow$ YEAR AUT TITLE, BOOK $\longrightarrow$ AUT YEAR TITLE)
(*Rename*(AUTHORS,AUT,*)(*UnGroup*(BOOKS $\longrightarrow$ BOOK*, AUTHORS $\longrightarrow$ A*)d_2))))))))).

Figure 15 shows the result of the above query. ■

Definition 4.15 $UnGroup(A \longrightarrow B^*, D \longrightarrow E^*)\langle t, \mathcal{G} \rangle = \langle t', \mathcal{G}' \rangle$ where,

$$\begin{aligned} t' &= t \text{ if } (t = []) \vee (has\text{-}leaf(t)) \vee (subtrees(t) = ()) \\ &= [root(t), (t'_1, ..., t'_k)] \text{ if } (satisfies(t, A \longrightarrow B^*) = false), \end{aligned}$$

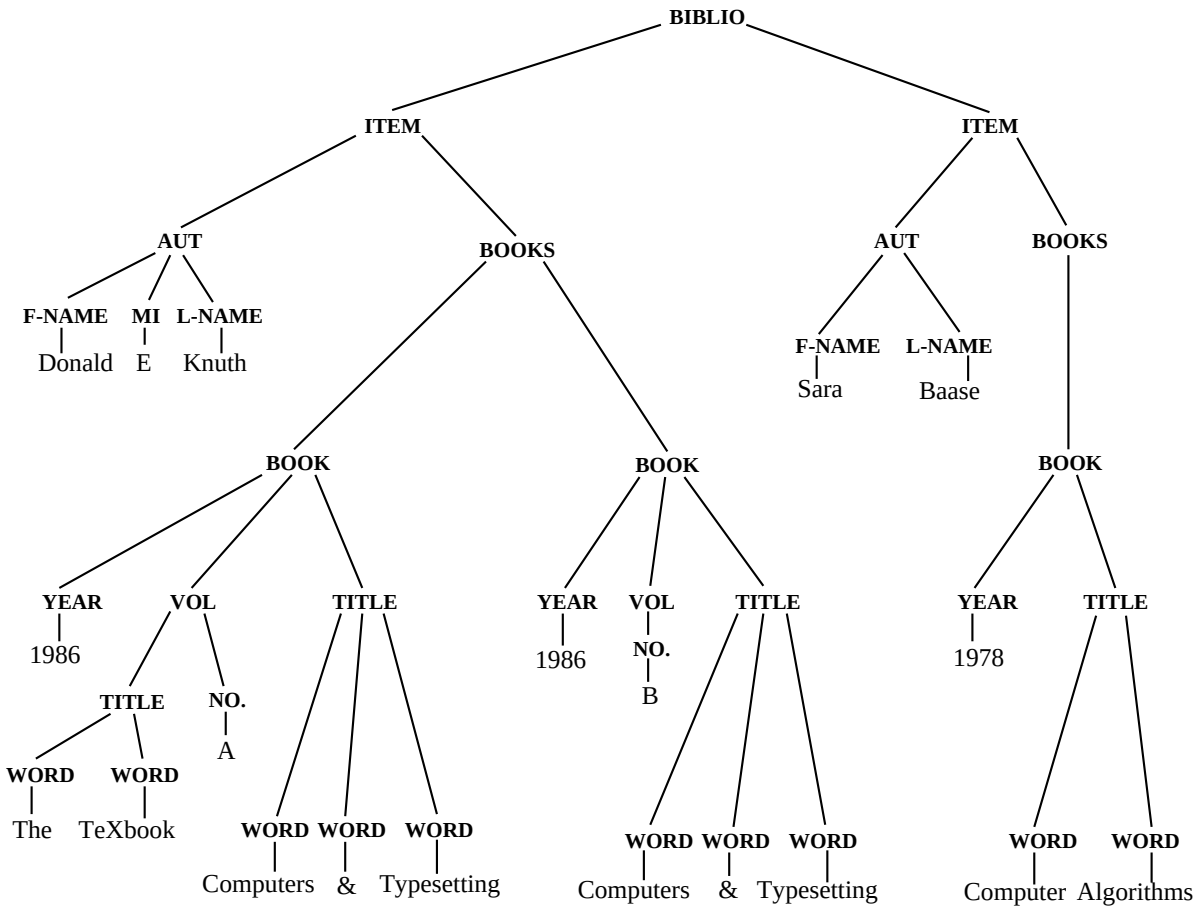


Figure 15: An example of restructuring with the *Group* and *UnGroup* operators.

$$
\begin{aligned}
&= [root(t), concat(l_1, l_2)], \text{ otherwise, where,}\\
&\quad l_1 = (t_1'), \text{ if } D \notin children(t_1)\\
&\quad\quad = (s_1, ..., s_n), \text{ otherwise, where, } n = size(subtrees(t_1[D])),\\
&\quad\quad\quad \text{and, } s_i = [root(t_1), (t_{i1}'', ..., t_{im}'')], \text{ where,}\\
&\quad\quad\quad\quad subtrees(t_1) = (t_{11}, ..., t_{1m}),\\
&\quad\quad\quad\quad \text{and, } t_{ij}'' = t_{1j}', \text{ if } root(t_{1j}) \neq D,\\
&\quad\quad\quad\quad\quad = [D, subtrees(t_{1j_i}')], \text{ otherwise,}\\
&\quad\quad\quad\quad\quad \text{where, } subtrees(t_{1j}) = (t_{1j_1}, ..., t_{1j_n})\\
&\text{and } l_2 = subtrees(struct(UnGroup(A \longrightarrow B^*, D \longrightarrow E^*) l_2'))\\
&\quad\quad \text{where, } l_2' = \langle [root(t), (t_2, ..., t_k)], \mathcal{G} \rangle.\\
t_i' &= struct(UnGroup(A \longrightarrow B^*, D \longrightarrow E^*)\langle t_i, \mathcal{G}_i \rangle), \text{ if } (reachable(A, root(t_i), P))\\
&= t_i, \text{ otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S, P - P_1 \cup P_2]\!]), \text{ where,} \\
P_1 &= \{D \longrightarrow E^* \}, \text{ if } (B \neq S) \wedge (D \neq S) \\
&\qquad \wedge (\forall q \in P(B \in tail(q)) \Rightarrow (q = A \longrightarrow B^*)) \\
&\qquad \wedge (\forall q \in P(D \in tail(q)) \Rightarrow (B = head(q))) \\
&= \{\ \}, \text{ otherwise,} \\
P_2 &= \{q \mid (\exists p \in P) \wedge (head(p) = E) \wedge (head(q) = D) \\
&\qquad \wedge (tail(q) = tail(p))\}.
\end{aligned}
$$

□

4.3.11 Number

This operator finds trees corresponding to a given list production and numbers the subtrees. The left most subtree is numbered '1', the next one is numbered '2' and so on. This operator is useful, in combination with the *Sort* operator, in preserving the order of the elements in a list of subtrees, while performing operations, like *Group* and *UnGroup*, that can change their order. Section 5.7 contains a few examples that illustrate this. *Number* is also useful as a counting operator as shown in Example 4.26.

Let $\langle t, \mathcal{G} \rangle$ be a list-structure instance. Let $A \longrightarrow B^*$ be a production in P and suppose that there are no list or leaf productions in P with B as the head. Let D be a variable, not in $var(\mathcal{G})$. Then, for each tree corresponding to $A \longrightarrow B^*$, $Number(A \longrightarrow B^*, D)$ numbers its subtrees by adding to each subtree, a tree with root node D and leaf node i, where the number i corresponds to the ith subtree.

Example 4.26 *Find the number of books in* d_1.

Replace(ANSWER $\longrightarrow$ B, B $\longrightarrow$ *)(*Rename*(B,(B),ANSWER $\longrightarrow$ B*)
(*Retrieve*(B,ANSWER)(*Find*(D, (ϕ, >), B)
(*Insert*(A,(-,*))((*Retrieve*(D,A)(*Number*(BOOKS $\longrightarrow$ BOOK*,D)d_1)),
⟨[D,0],[\![D,{D $\longrightarrow$ Integer}]\!]⟩))))).

The resulting structure will have ANSWER as the root node and an integer representing the number of books as the leaf. ■

Definition 4.16 $Number(A \longrightarrow B^*, D)\langle t, \mathcal{G}\rangle = \langle t', \mathcal{G}'\rangle$, where

$$
\begin{aligned}
t' &= t, \text{ if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
&= [root(t), (t''_1, ..., t''_k)], \text{ if } (satisfies(t, A \longrightarrow B^*)), \text{ where,} \\
&\quad t''_i = [root(t_i), (t'_{i1}, ..., t'_{im}, [D, i])], \text{ where, } (t_{i1}, ..., t_{im}) = subtrees(t_i), \\
&= [root(t), (t'_1, ..., t'_k)], \text{ otherwise,} \\
\text{where, } t'_i &= struct(Number(A \longrightarrow B^*, D)\langle t_i, \mathcal{G}_i\rangle), \text{ if } (reachable(A, root(t_i), P)) \\
&= t_i, \text{ otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}' &= normalize([\![S, P - P_1 \cup P_2]\!]), \text{ where} \\
P_1 &= \{q \in P \mid (head(q) = B)\}, \text{ if } (B \neq S) \\
&\qquad \wedge(\forall p \in P(B \in tail(p)) \Rightarrow (p = A \longrightarrow B^*)), \\
&= \{\}, \text{ otherwise,} \\
P_2 &= \{D \longrightarrow Integer\} \cup \{p \mid (\exists q \in P) \wedge (head(q) = head(p) = B) \\
&\qquad \wedge(tail(p) = tail(q)D)\}
\end{aligned}
$$

□

4.3.12 Sort

This operator sorts the subtrees of trees corresponding to a given list production. The subtrees must all correspond to aggregate productions and must all have a common attribute on which the sorting is to be done. We first define a *sort specification*.

Definition 4.17 Let $\mathcal{G} = [\![S, P]\!]$ be a list-structure scheme and B be a variable in $var(\mathcal{G})$. Then, L is a sort specification on B, if (i) L is C, or $C(L')$, where C is a variable in $var(\mathcal{G})$ and L' is a sort specification on C, and (ii) for some aggregate production p in P, $head(p) = B$ and $C \in tail(p)$, and (iii) for every production q in P if $head(q) = B$ then $C \in tail(q)$ and q is an aggregate production and (iv) if L is C, there can only be leaf productions with C as the head and if there are two or more leaf productions with C as the head, then their tails must all be constants of the same domain. □

A sort specification is essentially an attribute or a path leading to an attribute on which the sorting is to be done.

Example 4.27 *Sort the books in* d_1 *by the* `YEAR` *attribute.*

This query can be expressed as follows:

$Sort$(BOOKS $\longrightarrow$ BOOK*,YEAR)d_1.

Let us suppose that the book instances had an attribute called DATE which is the date when the book was published, and DATE was an aggregate of two attributes MM and YY. If we wanted to sort the books according to the date published we could do the following.

$Sort$(BOOKS $\longrightarrow$ BOOK*,DATE(YY))($Sort$(BOOK $\longrightarrow$ BOOK*,DATE(MM))d_1). ■

Definition 4.18 Let $\langle t, \mathcal{G} \rangle$ be an instance and let $A \longrightarrow B^*$ be a production in P and L a sort specification on B. Then,

$Sort(A \longrightarrow B^*, L)\langle t, \mathcal{G} \rangle = \langle t', \mathcal{G} \rangle$, where

$$
\begin{array}{rcl}
t' & = & t, \text{ if } (t = []) \vee (has\text{–}leaf(t)) \vee (subtrees(t) = ()) \\
 & = & [root(t), (t'_1, ..., t'_k)], \text{ if } (satisfies(t, A \longrightarrow B^*) = false) \\
 & = & [root(t), append(t'_q, l)], \text{ otherwise, where,} \\
 & & \quad q = smallest(subtrees(t), L), \text{ and} \\
 & & \quad l = subtrees(struct(Sort(A \longrightarrow B^*, L)\langle [root(t), (s_1, ..., s_{k-1})], \mathcal{G} \rangle)), \\
 & & \quad \text{where, } s_i = t_i, \text{ if } i < q \\
 & & \qquad\qquad = t_{i+1}, \text{ otherwise,} \\
\text{where, } t'_i & = & struct(Sort(A \longrightarrow B^*, L)\langle t_i, \mathcal{G}_i \rangle), \text{ if } (reachable(A, root(t_i), P)) \\
 & = & t_i, \text{ otherwise.}
\end{array}
$$

Let $\mathcal{G} = [\![S, P]\!]$ be a list-structure scheme and let $(t_1, ..., t_k)$ be a list where each t_i is a non-empty structure over $\mathcal{G}'$, where $\mathcal{G}' = normalize([\![B, P]\!])$, for some $B \in var(\mathcal{G})$. Let L be a sort specification on B. Then,

$$
\begin{array}{rcl}
smallest((t_1, ..., t_k), L) & = & q \mid (\forall b_j\ (b_q \leq b_j)) \wedge (\forall b_p\ ((p < q) \Rightarrow (b_p > b_q))), \text{ if } L = C, \\
 & & \quad \text{where, each } b_i = subtrees(t_i[C]), \\
 & = & smallest((c_1, ..., c_k), L'), \text{ if } L = C(L'), \\
 & & \quad \text{where, each } c_i = t_i[C]
\end{array}
$$ □

4.3.13 Apply

All of the operators defined so far perform global actions on instances. The operators examine the instance at every level, checking if the tree being examined corresponds to a production or to some other pattern, and then perform some action on it. In some cases it might be convenient to have the operators perform locally. For instance, consider Figure 16 which contains information about a PARTS and SUBPARTS database. Suppose that we want to retrieve, for each part, the part numbers (PNO's) of (recursively) all of its subparts and add this list as an extra attribute to each PART tree. The result we expect is shown in Figure 17. However, performing a *Retrieve*(PNO,B) will give us all of the PNO's in the instance and not the desired result. What is needed is an operator which can apply any other operator (or a composition of several operators) to specific trees in the structure of an instance, and which can, for each such tree, store the result of the operation within the same tree.

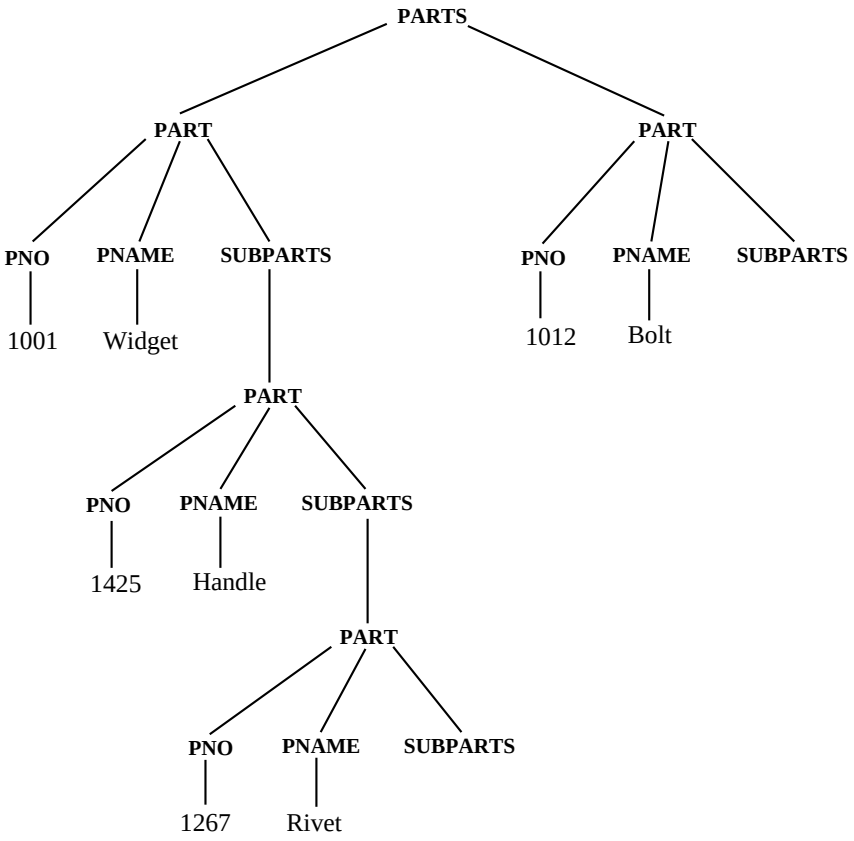


Figure 16: The PARTS list-structure.

The operators defined so far all have one or more parameters, (e.g., *Sort* has two parameters – a production and a sort specification), and all except *Insert* and *Substitute* take

one instance as input. We will refer to unary operators (operators that take only one instance as input) with values substituted for their parameters as *parameterized unary operators*. For example, $Find$(`BOOK, [YEAR (= (1990))], B`) is a parameterized unary operator over the book-collection scheme. In the case of $Insert(Substitute)$, we define the function $Insert\text{–}f(Substitute\text{–}f)$ that has the same parameters as $Insert(Substitute)$, but that takes only one instance as input and returns a function that takes a single instance as input.

$(Insert\text{–}f(A,\beta)\langle t_2,\mathcal{G}_2\rangle)\langle t_1,\mathcal{G}_1\rangle = Insert(A,\beta)(\langle t_1,\mathcal{G}_1\rangle,\langle t_2,\mathcal{G}_2\rangle)$.

$Insert\text{–}f(A,\beta)\langle t_2,\mathcal{G}_2\rangle$ returns a function which when applied to $\langle t_1,\mathcal{G}_1\rangle$ returns the same result as the one that would have been obtained by applying $Insert(A,\beta)$ to $(\langle t_1,\mathcal{G}_1\rangle,\langle t_2,\mathcal{G}_2\rangle)$.

Similarly, $(Substitute\text{–}f(\alpha,\beta)\langle t_2,\mathcal{G}_2\rangle)\langle t_1,\mathcal{G}_1\rangle = Substitute(\alpha,\beta)(\langle t_1,\mathcal{G}_1\rangle,\langle t_2,\mathcal{G}_2\rangle)$.

Definition 4.19 A *parameterized operator* δ over a scheme $\mathcal{G}$, is either a parameterized unary operator over $\mathcal{G}$ or is of the form $Insert\text{–}f(A,\beta)\langle t_2,\mathcal{G}_2\rangle$ or of the form $Substitute\text{–}f(\alpha,\beta)\langle t_2,\mathcal{G}_2\rangle$ such that for any instance $\langle t,\mathcal{G}\rangle$, $\delta\langle t,\mathcal{G}\rangle$ is a valid operation. □

Note that for any parameterized operator δ, if $\delta\langle t,\mathcal{G}\rangle=\langle t',\mathcal{G}'\rangle$, then $\mathcal{G}'$ is independent of t. So, we can write $\delta(\mathcal{G}) = \mathcal{G}'$.

Definition 4.20 A *parameterized expression*, f over a scheme $\mathcal{G}$ is a list of parameterized operators $(f_1,...,f_n)$, $n \geq 1$, such that (i) f_n is a parameterized operator over $\mathcal{G}$ and (ii) each f_{i-1}, $1 < i \leq n$, is a parameterized operator over $f_i(f_{i+1}(...(f_n(\mathcal{G}))...))$. □

Definition 4.21 Let $\langle t,\mathcal{G}\rangle$ be an instance and let $p = A \longrightarrow B_1...B_k$ be a production in $\mathcal{G}$ and let B be a variable not in $tail(p)$. Let $f = (f_1,...,f_n)$ be a parameterized expression over $\mathcal{G}'$ where $\mathcal{G}'=normalize([\![A,P]\!])$. Let $eval(f,\langle t,\mathcal{G}\rangle)$ denote the expression $(f_1(f_2(...(f_n\langle t,\mathcal{G}\rangle)...)))$. Then,

$Apply(A \longrightarrow B_1...B_k, B, f)\langle t,\mathcal{G}\rangle = \langle t',\mathcal{G}'\rangle$, where t' and $\mathcal{G}'$ are as follows:

$$
\begin{aligned}
t' &= t, \text{ if } (t=[]) \vee (has\text{–}leaf(t)) \vee (subtrees(t)=()) \\
&\qquad \vee((satisfies(t,p)=true) \wedge (struct(eval(f,\langle t,\mathcal{G}\rangle))=[])) \\
&= [root(t),(t'_1,...,t'_k)], \text{ if } (satisfies(t,p)=false) \\
&= [root(t),(t'_1,...,t'_k,[B,(struct(eval(f,\langle t,\mathcal{G}\rangle)))])], \text{ otherwise,} \\
\text{where, } t'_i &= struct(Apply(A \longrightarrow B_1...B_k,B,f)\langle t_i,\mathcal{G}_i\rangle), \text{ if } (reachable(A,root(t_i),P)), \\
&= t_i, \text{ otherwise.}
\end{aligned}
$$

$$\begin{aligned} \mathcal{G}' &= normalize([\![S, P \cup P_f \cup P_1]\!]), \\ \text{where, } [\![S_f, P_f]\!] &= f_1(...(f_n(normalize([\![A, P]\!])))...), \\ \text{and } P_1 &= \{B \longrightarrow S_f, A \longrightarrow B_1...B_mB\}. \end{aligned}$$ □

Example 4.28 *The query corresponding to the* PARTS-SUBPARTS *example can be expressed as follows.*

Delete(SPNO,(*,([PNO,([D,*])]),*))(*Find*(PNO, (ϕ, <), D)
(*Replace*(SPNO $\longrightarrow$ S,S $\longrightarrow$ *)
(*Apply*(PART $\longrightarrow$ PNO PNAME SUBPARTS,SPNO,(*Retrieve*(PNO,S)))d_2)))

The last two operations (*Find* and *Delete*) are necessary since the *Apply* operation will retrieve, for each PART, not only the PNO's of the SUBPARTS but also that of the PART itself. ■

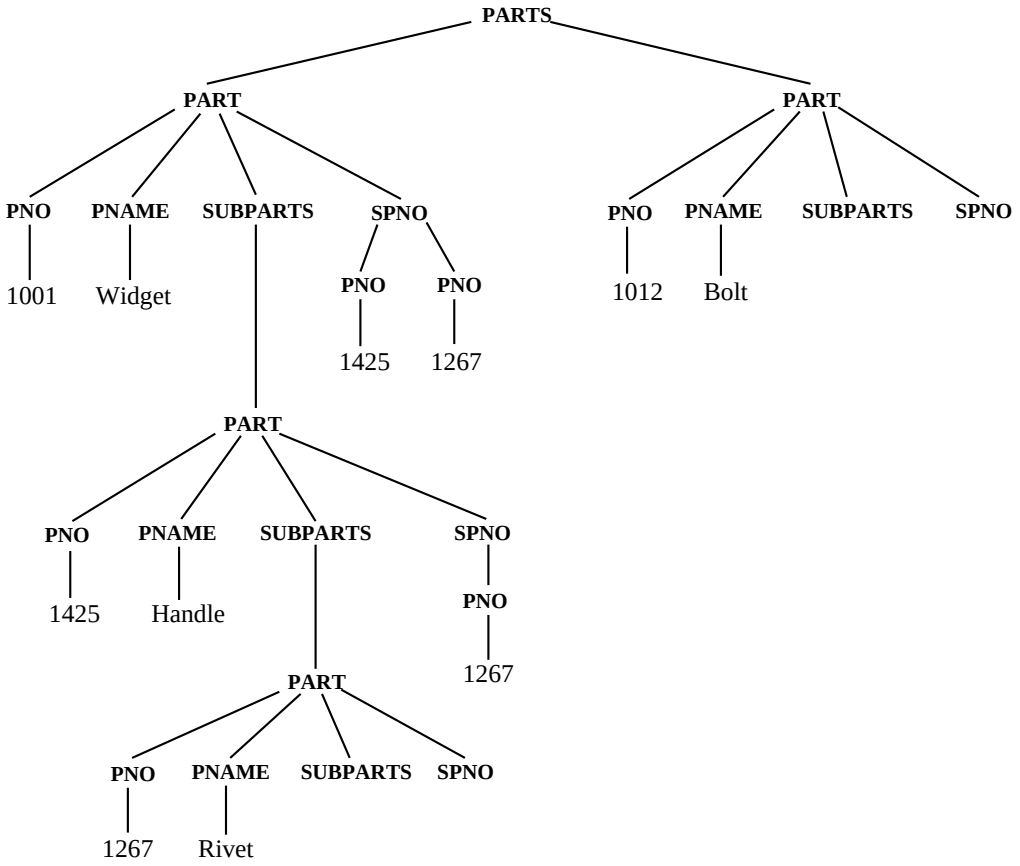


Figure 17: An example of the *Apply* operation.

4.3.14 Change-Scheme

Change–Scheme is a value preserving operator which is used to modify the schema of an instance by adding new productions and deleting extraneous ones (extraneous productions are those that are not satisfied by any tree in an instance). Each of the operators that we have described so far returns, as the result of the operation, an instance whose scheme depends only on the scheme(s) of the input instance(s) and not on the value(s) (i.e., the structure). Because of this, queries tend to enlarge the schemas (increase the number of productions) of the results. Since the value of the result of an operation is not examined to determine the productions that should be in the scheme of the result, schemas of the results of queries tend to "accumulate" productions. It would, hence, be useful to have an operator that can delete productions from the schema of an instance without making changes to its value.

It is also necessary to be able to add new productions to a schema of an instance. For example, let us suppose that in an instance $\langle t, \mathcal{G} \rangle$, the scheme $\mathcal{G}$ has the production `A` $\longrightarrow$ `Integer` and every tree in t which corresponds to this production has either a '`0`' or a '`1`' as its leaf. Now let us suppose that we want to change $\mathcal{G}$ by replacing the production `A` $\longrightarrow$ `Integer` by `A` $\longrightarrow$ `0` and `A` $\longrightarrow$ `1`. We will need an operator that allows productions to be explicitly added since none of the other operators can give us the desired result. Note, however, that if we wanted to perform the opposite, i.e., replace `A` $\longrightarrow$ `0` and `A` $\longrightarrow$ `1` with `A` $\longrightarrow$ `Integer`, or if we wanted to add aggregate or list productions, we could do so with the *Substitute* operator (see Section 5.6). However, *Change–Scheme* provides a more natural way of modifying schemas.

If $\langle t, \mathcal{G} \rangle$ is an instance and P_1 and P_2 are sets of productions such that $P_1 \subseteq P$ and $P_2 \cap P = \phi$, then the set of productions P is replaced by $P - P_1 \cup P_2$ provided every tree in t corresponds to some production in $P - P_1 \cup P_2$.

We first define the function *required–in* which takes a structure t, a set of productions P, and a production p, as arguments and returns true or false depending on whether or not there is some tree in t which is satisfied by p and is not satisfied by any production in P (other than p).

$\diamond$ Let t be a structure, P a set of productions, and p a production. Then,

$$
\begin{aligned}
\mathit{required\text{–}in}(t, P, p) &= \mathit{true}, \text{ if } (\mathit{satisfies}(t, p) = \mathit{true}) \wedge \\
&\qquad (\forall q \in P\ ((q \neq p) \Rightarrow (\mathit{satisfies}(t, q) = \mathit{false}))) \\
&= \mathit{true}, \text{ if } (t \neq []) \wedge (\mathit{has\text{–}leaf}(t) = \mathit{false}) \wedge \\
&\qquad (\exists t_i \in \mathit{subtrees}(t) \mid \\
&\qquad\qquad (\mathit{required\text{–}in}(t_i, P, p) = \mathit{true})) \\
&= \mathit{false}, \text{ otherwise.}
\end{aligned}
$$

◇

Definition 4.22 Let $\langle t, \mathcal{G} \rangle$ be an instance and let P_1 and P_2 be sets of productions as described earlier. Then,

$\mathit{Change\text{–}Scheme}(P_1, P_2)\langle t, \mathcal{G} \rangle = \langle t, \mathcal{G}' \rangle$, where

$$
\begin{aligned}
\mathcal{G}' &= \mathcal{G}, \text{ if } (\exists p \in P_1) \mid (\mathit{required\text{–}in}(t, P - P_1 \cup P_2, p) = \mathit{true}), \\
&= \mathit{normalize}([\![S, P - P_1 \cup P_2]\!]), \text{ otherwise.}
\end{aligned}
$$

□

Example 4.29 *Remove the extraneous production* `A ⟶ F-NAME L-NAME` *from the instance* d_1.

$\mathit{Change\text{–}}$`Scheme({A ⟶ F-NAME L-NAME},{})`d_1

■

Example 4.30 *Replace the productions* `NO. ⟶ Integer` *and* `NO. ⟶ Char` *in the instance* d_2 *with the set* {`NO. ⟶ 1,NO. ⟶ 2,NO. ⟶ A,NO. ⟶ B`}.

$\mathit{Change\text{–}Scheme}(\mathtt{P_1}, \mathtt{P_2})d_2$

where $\mathtt{P_1}$ and $\mathtt{P_2}$ are the set of productions being deleted and added, respectively. ■

5 A Discussion on the List-Structure Algebra

The choice of the operators for the algebra was based on the criteria outlined in Section 4.1. We show, in this section, how the algebra meets these criteria. We list the five criteria again for easy reference.

(i) *The language should be well-suited for list-oriented applications.*

(ii) *The operators must be simple enough to understand and implement but powerful enough to enable queries to be expressed succinctly.*

(iii) *The number of basic operators required to express most reasonable queries must be kept to a minimum.*

(iv) *The language must satisfy the closure property. In other words, the type of the objects returned by the operators must be the same as the type of the operands.*

(v) *Except for operators that are designed specifically to change the scheme of an instance depending upon its value, all other operators, when applied to an instance, must return an instance whose scheme is independent of the value of the input instance.*

As mentioned in Section 1, the concept of ordering is crucial to any list-oriented application. It is, therefore, essential that this concept be built into not only the data model, but also the query language. Similarly, the concept of variable schema, which is important for many list-oriented applications, must also be incorporated in the query language. It is quite obvious from the choice of the operators and their semantics that the list-structure algebra satisfies these two requirements and hence also satisfies criterion (i).

Again, from the definitions themselves, one can see that criteria (iv) and (v) are satisfied. Now, according to criteria (ii) and (iii) the number of operators should be kept to a minimum while allowing most reasonable queries to be expressed succinctly. These two criteria are somewhat conflicting since the minimal set of operators needed to express a certain class of queries may not be sufficient to express most queries succinctly. In other words, while a small set of operators is more manageable than a larger set, a minimal set of operators may not be the ideal choice for a practical query system. The choice of the operators for the

list-structure algebra was based on achieving a reasonable balance between the two criteria, rather than on satisfying one or the other. The rest of this section shows how this balance is achieved.

All of the operators, except Reorder are basic operators. In some of the other operators, not all different cases are basic. For instance, the case when α in $Substitute(\alpha, \beta)$ is a variable can be expressed in terms of other operators, whereas the case when α is '$*$' cannot always be expressed using other operators. We show how the non-basic operator and the non-basic cases for the basic operators can be expressed in terms of other operators and then justify why these non-basic operators and cases are provided. We also show how operators like union, difference, intersection and join can be simulated in this algebra.

5.1 Delete

$Delete(A, \beta)\langle t, \mathcal{G}\rangle$, where $\beta = (\notin (C_1, ..., C_m))$ (Case 1(b) in Section 4.3.4) can be simulated in terms of the other $Delete$ cases as follows.

For each $p \in P$, such that $head(p) = A$ and one of the C_i's is in $tail(p)$ and some $D_i \in tail(p)$ and $D_i \notin (C_1, ..., C_m)$ perform the following:

1. $Replace(p, E \longrightarrow tail(p))$, where $E \notin var(\mathcal{G})$
2. For each $D_i \in tail(p)$ such that $D_i \notin (C_1, ..., C_m)$ perform a $Delete(E, (\equiv [D_i, *]))$
3. $Replace(A \longrightarrow E, E \longrightarrow *)$

The scheme of the result of the above simulation will contain extraneous productions since a $Delete(E, (\equiv [D_i, *]))$ will not delete the productions with E as the head and D_i in the tail. These can be removed by means of the $Change\text{–}Scheme$ operator. This $Replace$ case has been provided because if there are several attributes that are to be deleted (possibly from several different productions) then the number of $Delete$ operations can be quite large when expressed in terms of the other $Delete$ cases.

5.2 Substitute

Let $\alpha = A$ and β be a structure pattern over B. $Substitute(\alpha, \beta)(\langle t_1, \mathcal{G}_1 \rangle, \langle t_2, \mathcal{G}_2 \rangle)$ can be expressed in terms of the other operators as follows.

Let C, D and E be variables not in $\mathcal{G}_1$ or $\mathcal{G}_2$. For better readability, we list the operations to be performed in separate steps instead of writing a single expression formed by their composition. We make a few simplifying assumptions. We assume that (a) $A \neq B$, (b) for all productions $p, q \in P_1$, if $head(p) = head(q) = B$, $tail(p) = B_1 ... B_m$, and $tail(q) = Q^*$, then $\forall B_i \in tail(p)$, $B_i \neq Q$ and (c) for all $p \in P_1$ if $head(p) = B$ then $B \notin tail(p)$. The cases when these assumptions do not hold can also be simulated in terms of other operators, although a few additional steps will be required.

Let $c = (\equiv l)$, where l is such that $\beta = [B, l]$.

1. $r_1 = Find(B,\ c,\ C)\langle t_1, \mathcal{G}_1 \rangle$
2. $r_2 = Rename(B, D, A \longrightarrow *)r_1$
3. $r_3 = Rename(B, E, *)\langle t_2, \mathcal{G}_2 \rangle$
4. $r_4 = Insert(D,\ ([C, *], *))(r_2, r_3)$
5. $r_5 = Delete(D,\ (\notin (E)))r_4$
6. $r_6 = Rename(E, B, *)r_5$
7. $r_7 = Replace(D \longrightarrow B, B \longrightarrow *)r_6$
8. $r_8 = Rename(D, B, *)r_7$
9. $r_9 = Replace(B \longrightarrow C, C \longrightarrow *)r_8$

r_9 is equivalent to $Substitute(A, \beta)(\langle t_1, \mathcal{G}_1 \rangle, \langle t_2, \mathcal{G}_2 \rangle)$.

One can expect update operations, where one item is replaced by another, to be used very often. Since there are several steps involved in performing this in terms of the other operators in the algebra, it is better to provide an equivalent single step operation. $Substitute(*, \beta)$ cannot, in general, be expressed in terms of the other operators.

5.3 Rename

$Rename(A, B, C \longrightarrow *)$ can be expressed in terms of the other *Rename* cases.

(a) If $A \neq C$ and $B \neq C$, then for each p, such that $head(p) = C$ and $A \in tail(p)$, perform a $Rename(A, B, p)$.

(b) If $A \neq C$ and $B = C$, then perform the following steps:

1. For each p such that $head(p) = C$ and $A \in tail(p)$ perform a $Rename(A, D, p)$ where $D \notin var(\mathcal{G})$.
2. Rename the D nodes to B by performing a $Rename(D, B, *)$.

(c) If $A = C$, then for each p such that $head(p) = C$ and $A \in tail(p)$, perform a $Rename(A, D, p)$ where $D \notin var(\mathcal{G})$. Then for each such p, perform a $Rename(A, B, q)$, where $head(q) = D$ and $tail(q) = tail(p)$. Finally, perform a $Rename(D, B, *)$.

When there are several productions with C as the head, it is more convenient to use $Rename(A, B, C \longrightarrow *)$, which is the reason this *Rename* case has been provided.

5.4 Replace

The *Replace* operator is a basic operator since it cannot always be expressed in terms of other operators. However, of the various cases, (described in Section 4.3.7), the following cases can be simulated using other *Replace* cases and other operators of the algebra.

(a) $Replace(A \longrightarrow B, B \longrightarrow *)$ can obviously be expressed in terms of other *Replace* cases. For each production p in the scheme of the input, such that $head(p) = B$, we can perform a $Replace(A \longrightarrow B, p)$. (If $A = B$, then $Replace(A \longrightarrow A, A \longrightarrow A)$ should be performed first before any of the other *Replace* operations). However, if there are a large number of productions with B as the head, then it is more convenient to use the more general case, i.e., $Replace(A \longrightarrow B, B \longrightarrow *)$.

(b) $Replace(A \longrightarrow C_1...C_k, C_l \longrightarrow C_1...C_k)$, where $C_l \notin (C_1, ..., C_k)$, can be expressed by $Find(A,\ \phi,\ C_l)$ if there is only one production whose head is A. If there is more than one production whose head is A, then the corresponding expression is $Find(A,\ (\equiv ([C_1, *], [C_2, *], ..., [C_k, *])),\ C_l)$. This is Case 1(c) of *Replace*, described in Section 4.3.7, when $tail(p_1) = tail(p_2)$. The case when $tail(p_2)$ has fewer variables than $tail(p_1)$ can also be expressed in terms of other operators but involves several operations.

(c) $Replace(A \longrightarrow B^*, C \longrightarrow B^*)$ can be expressed as $Find(A,\ \forall[B\ \phi],\ C)$. However, this *Replace* case has been provided for the sake of consistency. Since, the purpose of the *Replace* operator is to either introduce or remove internal nodes from a structure, while that of *Find* is to search and mark trees that satisfy certain conditions, it is more natural to use *Replace* when the main purpose is to introduce nodes.

(d) $Replace(A \longrightarrow b, C \longrightarrow b)$ where b is a constant can be written as $Find(A,\ (\equiv b),\ C)$. We use the same reasoning provided in the previous case to justify why this *Replace* case is provided.

In all of the above cases, the structure of the result of applying the alternate expression will be the same as the one that would have been obtained by applying the corresponding *Replace* operator. However, the scheme will contain extraneous productions that can be removed using *Change–Scheme*.

5.5 Reorder

This is the only operator in the algebra for which it is always possible to find an equivalent expression in terms of the other operators. $Reorder(A \longrightarrow B_1...B_k, A \longrightarrow C_1...C_k)\langle t, \mathcal{G}\rangle$ can be simulated by the following steps. We assume that $A \notin (B_1, ..., B_k)$. The case when $A \in (B_1, ..., B_k)$ requires a few additional steps. We list the parameterized operators to be applied in sequence to $\langle t, \mathcal{G}\rangle$. Let B, C and D be variables not in $var(\mathcal{G})$.

1. For each B_i do the following:
   (a) $Find(B_i, \phi, C)$
   (b) For each p in $\mathcal{G}$ (the original scheme) such that $head(p) = B_i$, perform a $Change–Scheme(\{p\}, \{\})$.
   (c) $Insert(B_i,\ (*, [C, *]))\langle [D, j], [\![D, \{D \longrightarrow Integer\}]\!]\rangle$, where $B_i = C_j$
   (d) $Change–Scheme(\{B_i \longrightarrow C\}, \{\})$
2. $Rename((B_1, ..., B_k), B, A \longrightarrow B_1...B_k)$
3. $Sort(A \longrightarrow B^*, D)$
4. $Rename(B, (C_1, ..., C_k), A \longrightarrow B^*)$

5. For each C_i, perform the following:

   (a) $Delete(C_i,\ (\notin (C)))$

   (b) $Replace(C_i \longrightarrow C, C \longrightarrow *)$

At this stage the structure of the result will be the same as that of the result of applying the *Reorder* operation. However, the scheme will contain many extraneous productions. We expect this rearranging operation to be needed quite often in queries, and hence the decision to provide this as an operator of the language.

5.6 Change-Scheme

It is possible to add new productions to the scheme of an instance without changing its value (except in the case when the new production is of the form $A \longrightarrow b$, and the production $A \longrightarrow \tau$, where $b \in Dom(\tau)$, is present in the original set of productions). This can be done using the *Substitute* operator. For instance, let $\langle t_1, \mathcal{G}_1 \rangle$ be an instance and let us suppose that we want to add the production $A \longrightarrow BC$ to the scheme $\mathcal{G}_1$ (where $A, B, C \in var(\mathcal{G}_1)$). Then, if S_1, the start symbol of $\mathcal{G}_1$, is not equal to A,

$$Substitute(*, [A, *])(\langle t_1, \mathcal{G}_1 \rangle, \langle [], \mathcal{G}_2 \rangle) = \langle t_1, \mathcal{G}_1' \rangle,$$

where $\mathcal{G}_2$ is a list-structure scheme whose start symbol is A and whose set of productions P_2 is such that $P_2 - P_1 = A \longrightarrow BC$, and $\mathcal{G}_1' = [\![S_1, P_1 \cup \{A \longrightarrow BC\}]\!]$.

If $S_1 = A$, then

$$Replace(A \longrightarrow D, D \longrightarrow *)(Substitute(*, \beta)(Find(A, \phi, D)(\langle t_1, \mathcal{G}_1 \rangle), \langle [], \mathcal{G}_2 \rangle))$$

will give us the desired result, where $D \notin var(\mathcal{G}_1)$ and β is a structure pattern over[11] $A_{\mathcal{G}_1}$ such that if $\beta = [A, l]$ then $l \neq *$ and if $l = (\beta_1, ..., \beta_m)$ then at least one of the β_i's is not a '$*$' or a '$-$'. (The constraint on l is to ensure that there is no tree in $Find(A, \phi, D)\langle t_1, \mathcal{G}_1 \rangle$ with which β can match). However, as mentioned earlier, since one cannot always add leaf productions or delete extraneous productions using other operations, *Change–Scheme* is an essential operation.

[11] We use the subscript $\mathcal{G}_1$ for A to denote the fact that the structure pattern over A is defined in the context of $\mathcal{G}_1$.

5.7 Union, Difference and Intersection

We give examples to show how union, difference and intersection can be simulated in this algebra. We assume that these operations take two instances, $\langle t_1, \mathcal{G}_1 \rangle$ and $\langle t_2, \mathcal{G}_2 \rangle$ as input and form the union, etc., of subtrees of all trees in t_1 corresponding to a given list production and the subtrees of t_2. For instance, $Union(R \longrightarrow P^*)(r_1, r_2)$, (where r_1 and r_2 are the instances shown in Figure 18(a) and Figure 18(b)), will give the result shown in Figure 19.

(a) (b)

Figure 18: The instances r_1 and r_2.

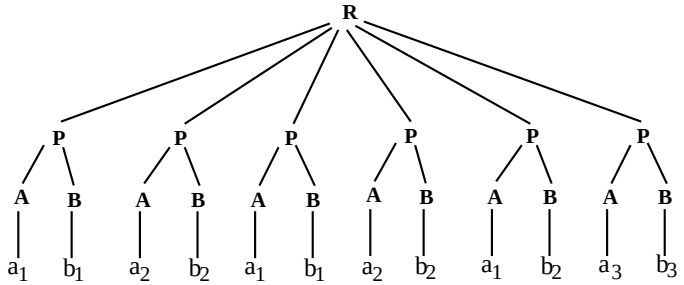


Figure 19: The Union of r_1 and r_2.

$Union(R \longrightarrow P^*)(r_1, r_2)$ can be expressed in the algebra by applying the following sequence of operations. Let S be a variable that is not in the scheme of r_1 or r_2.

$r_3 = Replace(R \longrightarrow P^*, S \longrightarrow P^*)r_1$
$r_4 = Insert(R, ([S, *], *))(r_3, r_2)$
$r_5 = Rename((S, R), S, R \longrightarrow SR)r_4$
$r_6 = Replace(R \longrightarrow S^*, S \longrightarrow P^*)r_5$

The instance r_6 contains the union of the subtrees of r_2 and the subtrees of all trees in r_1 that correspond to $R \longrightarrow P^*$. By union, we mean that the subtrees of the two trees being

unioned are simply concatenated. Duplicates are not removed as in the case of a set union. However, if for some reason we wanted the union to add to trees in t_1 only those subtrees in t_2 that are not already present in the trees that correspond to $R \longrightarrow P^*$, then we will have to perform some additional steps. Most of the steps required to perform this second type of union as well as difference and intersection are the same. So, we list the common steps involved in all of these operations and then specify the additional steps necessary for each operation. We assume that there is only one production p that has P as the head and we assume that p is an aggregate production with $A\ B$ as the tail. If there are several aggregate productions with P as the head, then some of the steps listed below, e.g. steps 9 and 15, will have to be repeated for each such production. If there is a list production q with P as the head, we can perform a $Replace(q, A \longrightarrow tail(q))$ so that all the trees that correspond to $R \longrightarrow P^*$, have subtrees that correspond only to aggregate productions.

Let C, D, E, F, S and Q be variables that are not in the schemes of r_1 or r_2.

1. $r_3 = Insert(P, (*, -))(r_1, \langle [C, 1], [\![C, \{C \longrightarrow Integer\}]\!] \rangle)$
2. $r_4 = Insert(P, (*, -))(r_2, \langle [C, 2], [\![C, \{C \longrightarrow Integer\}]\!] \rangle)$
3. $r_5 = Replace(R \longrightarrow P^*, S \longrightarrow P^*)r_3$
4. $r_6 = Insert(R, ([S, *], *))(r_5, r_4)$
5. $r_7 = Rename((S, R), S, R \longrightarrow SR)r_6$
6. $r_8 = Replace(R \longrightarrow S^*, S \longrightarrow P^*)r_7$
7. $r_9 = Change\text{-}Scheme(P_1, \{\})r_8$
   where, $P_1 = \{R \longrightarrow S^*, R \longrightarrow S, S \longrightarrow P^*, S \longrightarrow S, S \longrightarrow S^*, P \longrightarrow AB\}$
8. $r_{10} = Number(R \longrightarrow P^*, D)r_9$
9. $r_{11} = Replace(P \longrightarrow ABCD, E \longrightarrow CD)r_{10}$
10. $r_{12} = Group(R \longrightarrow P^*, E, F)r_{11}$

At this stage we perform different sequences of operations according to the operator being simulated.

Union

11. $r_{13} = Find(E,\ [F\ (\equiv ([C, 1], *))] \wedge [F\ (\equiv ([C, 2], *))], Q)r_{12}$
12. $r_{14} = Delete(Q,\ (*, ([F, ([C, 2], *)]), *))r_{13}$

Difference

11. $r_{13} = Find(E,\ [F\ (\equiv ([C,2],*))],\ Q)r_{12}$
12. $r_{14} = Delete(R,(*,([P,(*,[E,([Q,*])])]),*))r_{13}$

Intersection

11. $r_{13} = Find(E,\ [F\ (\equiv ([C,1],*))] \wedge [F\ (\equiv ([C,2],*))], Q)r_{12}$
12. $r_{14} = Delete(R,(*,([P,(*,[E,(*,[F,*],*)])]),*))r_{13}$

The following steps are again common to all the different cases.

13. $r_{15} = Replace(E \longrightarrow Q, Q \longrightarrow *)r_{14}$
14. $r_{16} = UnGroup(R \longrightarrow P^*, E \longrightarrow F^*)r_{15}$
15. $r_{17} = Replace(P \longrightarrow ABE, E \longrightarrow CD)r_{16}$
16. $r_{18} = Delete(P,(\equiv [C,*]))r_{17}$
17. $r_{19} = Sort(R \longrightarrow P^*, D)r_{18}$
18. $r_{20} = Delete(P,(\equiv [D,*]))r_{19}$

r_{20} contains the final result. Figure 20 shows the result of performing the operations corresponding to the union operation. The numbering and sorting steps can be omitted if it is not necessary that the order of the elements be preserved. For instance, in the case of union, the above sequence with steps 8, 9, 15, 17 and 18, omitted (and with all occurrences of the variable E replaced by C, and with corresponding changes to the structure, value and delete patterns), when applied to the instances in Figure 18 will give the result shown in Figure 21.

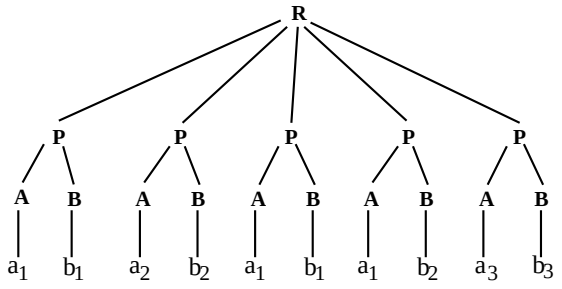


Figure 20: Union of r_1 and r_2 where no duplicates are added.

We have shown how operations like union, difference, can be simulated in the algebra. While it is quite cumbersome to express these operations in terms of the algebraic operators, these operations have not been provided in the algebra. This is because we do not expect them to

be used very often and since we know that they can be simulated in the algebra when really necessary.

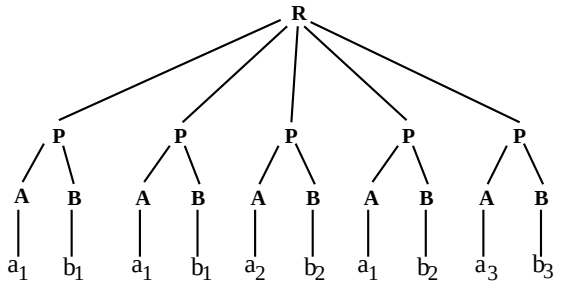


Figure 21: Union of r_1 and r_2 where no duplicates are added and the order of the elements is not preserved.

5.8 Cartesian-Product and Join

We show how 'joins' between two instances can be simulated in the algebra. We assume that the cartesian-product or join is performed between trees in the first instance that correspond to a given list production, and the structure of the second instance. We assume that the structure of the second instance also corresponds to a list production and that the subtrees of the two trees being joined all correspond to aggregate productions. Figure 23 shows the result of $Join(R \longrightarrow P^*)(s_1, s_2)$ where s_1 and s_2 are the instances shown in Figure 22(a) and Figure 22(b), respectively.

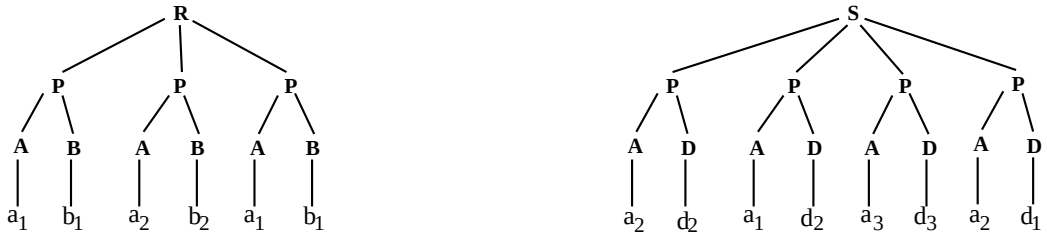


Figure 22: The instances s_1 and s_2.

This join can be simulated by the following steps. Let Q, E and F be variables that are not in the schemes of s_1 or s_2.

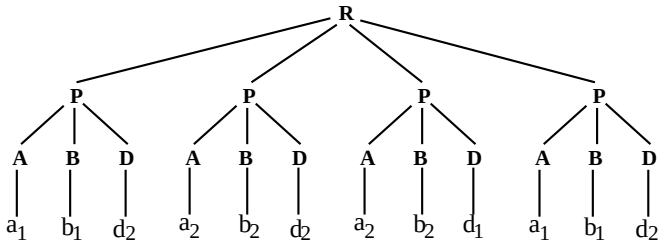


Figure 23: The result of $Join(R \longrightarrow P^*)(s_1, s_2)$.

$s_3 = Rename(P, Q, *)s_2$
$s_4 = Rename(A, E, *)s_3$
$s_5 = Insert(P, (*, -))(s_1, s_4)$
$s_6 = UnGroup(R \longrightarrow P^*, S \longrightarrow Q^*)s_5$
$s_7 = Replace(P \longrightarrow ABS, S \longrightarrow ED)s_6$

At this stage we have, essentially, the cartesian-product of the two instances s_1 and s_2. We can now delete the trees in s_7 that do not agree on the values of A and E and then delete the E subtree to get the join of s_1 and s_2.

$s_8 = Find(P, (A \neq E), F)s_7$
$s_9 = Delete(R, (*, ([P, ([F, *])]), *))s_8$
$s_{10} = Delete(P, (\equiv [E, *]))s_9$

s_{10} is equivalent to $Join(R \longrightarrow P^*)(s_1, s_2)$.

As mentioned earlier, goals (ii) and (iii) in Section 4.1 are conflicting and this is somewhat true in the case of join-like operators. Not providing these operators in the language seems to violate goal (ii) since simulating these operations in terms of the other operators involves a number of steps. On the other hand, it is not entirely clear how often one can expect such operations to be needed. Unlike the (flat) relational model, the hierarchical structure of the list-structure model allows complex objects to be represented within the same structure (to a limited extent). This in turn, reduces the number of join operations that are typically needed in a query. However, one can expect join operations to be used more often than operations like union and difference. Also, the frequency of use of the join operations is very likely to be dependent on the type of application being supported. In applications where join operations are needed very often, it would be useful to provide these as operators within the language.

6 Extending the List-Structure Model and Algebra

In this section we describe possible extensions and modifications to the list-structure model and the list-structure algebra. In Section 3.1 we outlined the kinds of data objects that should be representable in any model that supports list-oriented applications. The list-structure model supports all of those object types except (ordered and unordered) sets and bags. Sections 6.1 and 6.2 show how the model and the language can be extended to deal with sets. Section 6.3 shows how the model (and the language) can be extended to resolve ambiguity when a structure corresponds to two or more productions. In Section 6.4 we show how arithmetic operators can be incorporated in the language.

6.1 Ordered Sets

The list-structure model currently allows three types of objects, atomic objects (leaf nodes), aggregate objects and lists of objects of the same type. Lists (subtrees of trees that correspond to list productions) can have duplicate elements. However, in some cases, it may be known that certain lists will not have duplicates and it might be useful to be able to impose this as a constraint. For example, the list of authors of a book will not have duplicate elements. The model can be extended to allow this type of lists as a different list representation, as follows.

We will first need to distinguish between productions that refer to general lists (the ones we have seen so far) and productions that refer to ordered-sets (or lists that cannot have duplicate elements). We can do so by placing a distinguishing symbol on the arrow in the production. For example, $A \xrightarrow{list} B^*$ and $A \xrightarrow{o-set} B^*$ refer to list and ordered-set productions, respectively. The definition of a structure has to be modified as follows:

Let *type* be a function that takes a production as input and returns the type of the production – 'aggr' if it is an aggregate production, 'list', if it is a list production, 'o-set' if it is an ordered-set production and 'leaf' if it is a leaf production.

Definition 6.1 t is a finite structure over a scheme $\mathcal{G} = [\![S, P]\!]$, if

1. $t = []$, the empty structure over $\mathcal{G}$,
2. $t = [S, l]$, where S, the start symbol, is the root of the structure and l is a list of

subtrees, such that

(a) $l = (t_1, ..., t_k)$, $k \geq 1$, and for some $p \in P$, $head(p) = S$ and $tail(p) = B_1...B_k$, and each t_i is a non-empty structure over $\mathcal{G}_i$, where $\mathcal{G}_i = normalize([\![B_i, P]\!])$, or

(b) $l = (t_1, ..., t_m)$, $m \geq 0$, and for some $p \in P$, $head(p) = S$, $tail(p) = B^*$ and $type(p)$='list', and each t_i is a non-empty structure over $\mathcal{G}'$, where $\mathcal{G}' = normalize([\![B, P]\!])$, or

3. $t = [S, \{[t_1, ..., t_m]\}]$ $m \geq 0$, and for some $p \in P$, $head(p) = S$, $tail(p) = B^*$ and $type(p)$='o-set', and each t_i is a non-empty structure over $\mathcal{G}'$, where $\mathcal{G}' = normalize([\![B, P]\!])$, and $\forall t_i, t_j$ $(t_i \neq t_j)$, or

4. $t = [S, b]$, where S is the root of the structure and b is a constant, and for some $p \in P$, $head(p) = S$ and either $tail(p) = b$, or $tail(p) = \tau$, where τ is a type and $b \in Dom(\tau)$.

□

The definitions of the operators must also be modified. Rather than rewriting the definitions for all the operators, we show how the definitions can be modified by giving the modified definition for one of the operators. We first describe a new function, *rm–dpl* that will be used in the definition.

◇ The function *rm–dpl* takes a list l_1 and returns an ordered-set with any duplicate elements in l_1 removed. If there are several occurrences of an element in l_1, the first occurrence of that element (from left to right) is the one that is retained. ◇

We redefine the operator *Substitute* as shown below. The only difference between the definition given here and that in Section 4.3.5 is that when the subtrees of a tree corresponding to an ordered-set production are being replaced, the function *rm–dpl* is applied to the resulting list of subtrees. This is because, the pattern β can match with several subtrees in the list of subtrees which will result in duplicates after the substitution.

Let $\langle t_1, \mathcal{G}_1 \rangle$ and $\langle t_2, \mathcal{G}_2 \rangle$ be two list-structure instances. Let β be a structure pattern over some $B \in var(\mathcal{G}_1)$ and let $root(t_2) = B$. Let α be either $*$ or a variable $A \in var(\mathcal{G})$ such that for some $p \in P_1$, $head(p) = A$ and $B \in tail(p)$.

Definition 6.2 $Substitute(\alpha, \beta)(\langle t_1, \mathcal{G}_1 \rangle, \langle t_2, \mathcal{G}_2 \rangle) = \langle t', \mathcal{G}' \rangle$, where

$$
\begin{aligned}
t' &= [], \text{ if } (t_1 = []) \wedge (t_2 = []) \\
&= t_1, \text{ if } (t_2 = []) \wedge (((\alpha = *) \wedge (match(t_1, \beta) = false)) \vee (\alpha \neq *)) \\
&= t_1, \text{ if } (t_1 = []) \wedge (((S_1 \neq S_2) \wedge (\alpha = *)) \vee (\alpha \neq *)) \\
&= t_2, \text{ if } ((\alpha = *) \wedge (match(t_1, \beta))) \\
&= t_1, \text{ if } (((match(t_1, \beta) = false) \wedge (\alpha = *)) \vee (\alpha \neq *)) \\
&\qquad \wedge ((has\text{-}leaf(t_1)) \vee (subtrees(t_1) = ())) \\
&= [root(t_1), rm - dpl(t''_{11}, ..., t''_{1k})], \text{ if } (subtrees(t) = \{\![t_{11}, ..., t_{1k}]\!\}), \\
&= [root(t_1), (t''_{11}, ..., t''_{1k})], \text{ otherwise,} \\
&\quad \text{where, } t''_{1i} = t'_{1i}, \text{ if } (\alpha = *) \vee ((\alpha \neq *) \wedge \\
&\qquad ((match(t_{1i}, \beta) = false) \vee (root(t_1) \neq \alpha))) \\
&\qquad = t_2, \text{ otherwise} \\
&\qquad\quad \text{where, } t'_{1i} = struct(Substitute(\alpha, \beta)(\langle t_{1i}, \mathcal{G}_{1i} \rangle, \langle t_2, \mathcal{G}_2 \rangle)), \\
&\qquad\qquad \text{if } ((\alpha \neq *) \wedge (reachable(\alpha, root(t_{1i}), P))) \\
&\qquad\qquad \vee ((\alpha = *) \wedge (reachable(B, root(t_{1i}), P))) \\
&\qquad\qquad = t_{1i}, \text{ otherwise}
\end{aligned}
$$

$\mathcal{G}' = normalize([\![S_1, P_1 \cup P_2]\!])$ □

Most of the other operators can be extended along similar lines. Some, however, will require additional changes. Instance patterns, conditions, etc., will have to be redefined. Some of the function definitions, such as those for *concat* and *append* will have to be modified. Additional operators that make trees that correspond to a list production, correspond to an ordered-set production, and vice-versa could be provided.

6.2 Sets

While ordered-sets are useful in modeling lists that cannot have duplicates, in some cases, it might be necessary to be able to model general sets (in which the order of elements is immaterial). For instance, we might be interested in finding all the books whose authors are all members of a given set. The '$\subseteq$' comparison operator will not give us the correct answer if the set of authors is an ordered set. There are two possible ways of extending the model to deal with this type of situation.

One way is to introduce set comparison operators that take ordered-sets as arguments but treat them as general sets, by ignoring the order of the elements. For example,

$\{\![a_1, a_3]\!\} \subset \{\![a_1, a_2, a_3]\!\}$ is false, whereas $\{\![a_1, a_3]\!\} \overset{set}{\subset} \{\![a_1, a_2, a_3]\!\}$ is true,

where $\overset{set}{\subset}$ denotes the subset comparison, whereas $\subset$ denotes the sublist comparison. Similarly, $\overset{set}{\equiv}$ can be used to denote set equality between structures. In this method, the model does not need to be extended to allow sets and only the query language has to be modified. A disadvantage of using different operators for list and set comparisons is in comparing structures with many different levels of sub-structures. For instance, in the expression $\{\![a_2, a_1, \{\![a_4, a_5]\!\}]\!\} \overset{set}{\equiv} \{\![a_1, a_2, \{\![a_5, a_4]\!\}]\!\}$, the comparison between the two structures could treat only the outermost ordered-sets as sets or treat ordered-sets at all levels as sets. There are obvious limitations in both schemes since each type of comparison precludes the other.

A second way of dealing with set operations is to introduce general sets as a new data type within the model itself. This can be done along the lines outlined in the previous section for incorporating ordered sets in the model. Thus, $A \xrightarrow{set} B^*$ will represent a set production and a structure corresponding to this production will be of the form $[A, \{t_1, ..., t_k\}]$. This overcomes the problem of determining if entities are to be treated as sets or ordered-sets.

6.3 Resolving Ambiguity

Given a structure t over some scheme $\mathcal{G}$, it is possible to determine a unique production in $\mathcal{G}$ that t corresponds to, except in the following cases. When t has () as its list of subtrees and there are two or more list productions in $\mathcal{G}$ that have t's root as the head - for example, if $t = [A, ()]$ and $A \longrightarrow B^*$ and $A \longrightarrow C^*$ are both in $\mathcal{G}$ - then t corresponds to more than one production. Similarly, if $t = [A, (t_1)]$, where t_1 has B as its root and there are two productions, $A \longrightarrow B^*$ and $A \longrightarrow B$, in $\mathcal{G}$, then t corresponds to both productions.

Such ambiguities could sometimes lead to unexpected results. Suppose that t_1 is a structure that corresponds to $A \longrightarrow C^*$ and that it has a non-empty list of subtrees. Let us suppose that after a few deletions, t_1 is left with an empty list of subtrees. Now, let us suppose that we want to insert some structure t_2 as the left-most subtree in trees that correspond to $A \longrightarrow B^*$. t_2 will be inserted in all trees that correspond to $A \longrightarrow B^*$, including t_1. Thus t_1 which originally corresponded to $A \longrightarrow C^*$ will now correspond to $A \longrightarrow B^*$, even though

this was not intended.

There are several ways of overcoming this problem. One way would be to disallow two list productions from having the same head and to disallow two productions such as $A \longrightarrow B^*$ and $A \longrightarrow B$ in the scheme. A better way would be to include a reference to the productions within the structure. So, for instance, a structure could be represented by the triplet [A,n,l], where A is a variable, l, a list of subtrees and n, a reference (a number or a pointer) to the production p that t corresponds to.

6.4 Arithmetic Operators

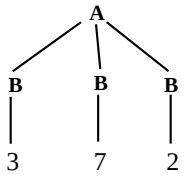
In theory, not all query languages have built-in arithmetic functions, while in practice most query languages provide aggregate functions like sum and average. Languages like SQL, and those proposed in [15] and [13], are examples of query languages augmented with aggregate functions.

In the list-structure algebra, the Apply function provides an easy means of integrating the results of aggregate or other functions in the structure. We give an example to illustrate this. Let *sum* be a function which takes a list production, say $A \longrightarrow B^*$ as a parameter and which when applied to a structure, searches (in a depth-first manner) for a tree that corresponds to the production. It then, returns the sum of the leaf nodes of the subtrees (assuming that B occurs as the head of only one production p, where $tail(p)$ is an integer constant or the type INTEGER). If no such tree is found it returns an empty structure []. So, $sum(A \longrightarrow B^*)$ when applied to the structure x_3 in Figure 24 will give the integer '12' as the result.

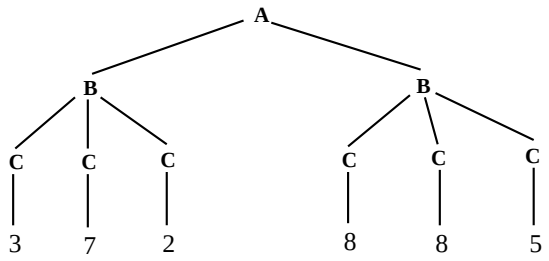
Now, consider the structure x_4 in Figure 24. Let us suppose that we want to find the sum of all the C subtrees of each of the B subtrees of A. The following expression will, for each B tree, restructure the B tree so that its group of C subtrees is an attribute, find the sum for its C subtrees, and insert the sum as a second attribute.

$$Apply(\texttt{B} \longrightarrow \texttt{D,E},sum(\texttt{D} \longrightarrow \texttt{C*}))(Replace(\texttt{B} \longrightarrow \texttt{C*},\texttt{D} \longrightarrow \texttt{C*})x_4)$$

The structure of the result of the above expression is shown in Figure 25.



The structure x_3



The structure x_4

Figure 24: Example list-structures.

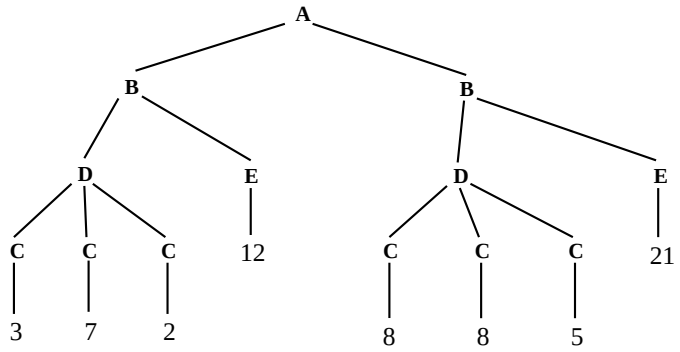


Figure 25: An example of an arithmetic operation using the *Apply* operator.

Alternatively, we can have *sum* return as a result, a structure with a specified root node, whose subtree is the integer denoting the sum. For example, (the structure part of) $sum(A \longrightarrow B^*, G)$ when applied to the structure x_3 in Figure 24 will be $[G, 12]$. This way, the closure property that was discussed in Section 4.1 is maintained.

Acknowledgements

I would like to thank Ed Robertson, Larry Saxton and Dirk Van Gucht for their ideas and suggestions which improved this paper substantially. I would also like to thank Orestes Appel and Chad Edge who worked on a partial implementation of the list-structure model and algebra.

References

[1] ABITEBOUL, S., AND HULL, R. Restructuring hierarchical database objects. *Theoretical Computer Science 62* (1988), 3–38.

[2] ANDRIES, M., GEMIS, M., PAREDAENS, J., THYSSENS, I., AND VAN DEN BUSSCHE, J. Concepts for graph-oriented object manipulation. Tech. Rep. 91-36, University of Antwerp (UIA), 1991. To appear in *Proceedings EDBT'92*, Lecture Notes in Computer Science, March 1992.

[3] BANCILHON, F., ET AL. The design and implementation of O_2, an object-oriented database system. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems* (Bad Münster am Stein-Ebernburg, Germany, September 1988), pp. 1–22.

[4] CHEN, P. P. The entity-relationship model–toward a unified view of data. *ACM Transactions on Database Systems 1*, 1 (March 1976), 9–36.

[5] CODD, E. F. A relational model for large shared data banks. *Communications ACM 6*, 13 (June 1970), 377–387.

[6] GONNET, G. H., AND TOMPA, F. W. Mind your grammar-a new approach to modelling text. In *Proceedings of the 13th VLDB* (Brighton, England, 1987), pp. 339–346.

[7] GÜTING, R. H., ZICARI, R., AND CHOY, D. M. An algebra for structured office documents. *ACM Transactions on Office Information Systems 7*, 4 (April 1989), 123–157.

[8] GYSSENS, M., PAREDAENS, J., AND VAN GUCHT, D. A grammar–based approach towards unifying hierarchical data models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, 1989), pp. 263–272.

[9] Hammer, M., and McLeod, D. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems 6*, 3 (September 1981), 351–386.

[10] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979, ch. Context-Free Grammars, pp. 77–106.

[11] Hull, R., and Yap, C. K. The Format model: A theory of database organization. *JACM 31*, 3 (1984), 518–537.

[12] Jaeschke, G., and Schek, H. J. Remarks on the algebra on non-first normal form relations. In *Proceedings of the first ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, 1982), pp. 124–138.

[13] Klug, A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM 29*, 3 (July 1982), 699–717.

[14] Macleod, I. A. A query language for retrieving information from hierarchic text structures. *The Computer Journal 34*, 3 (1991), 254–264.

[15] Ozsoyoglu, G., Ozsoyoglu, Z. M., and Matos, V. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems 12*, 4 (December 1987), 566–592.

[16] Pistor, P., and Traunmueller, R. A database language for sets, lists and tables. *Information Systems 11*, 4 (1986), 323–336.

[17] Rabitti, F. A model for multimedia documents. In *Office Automation*, D. Tsichritzis, Ed. Springer, New York, 1985, pp. 227–250.

[18] Shipman, D. W. The Functional Data Model and the data language DAPLEX. *ACM Transactions on Database Systems 6*, 1 (March 1981), 140–173.

[19] Thomas, S. J., and Fischer, P. C. *Nested Relational Structures.* JAIPress, 1986, pp. 269–307.