Ordered Types in the AQUA Data Model[*]

Bharathi Subramanian
Brown University
Providence, RI, USA

Stanley B. Zdonik
Brown University
Providence, RI, USA

Theodore W. Leung
Brown University
Providence, RI, USA

Scott L. Vandenberg[†]
University of Massachusetts
Amherst, MA, USA

To appear in Proc. 4th Intl. Workshop on Database Programming Languages, 1993

Abstract

We present a query algebra that supports ordering among the data elements. Order is defined as a relationship between various data elements of an instance. This relationship can be a total or partial order among the elements or among equivalence classes where each equivalence class consists of one or more elements. In terms of data structures, ordered types can be viewed as graphs, trees, or lists.

Lately there has been a lot of interest in bulk types like lists, trees, and graphs that are not supported by traditional data models and query algebras. This interest is fueled by the fact that much of the data in the scientific domain is inherently ordered. Therefore, scientific applications that involve genome sequences, satellite data, scientific data, etc. require database support for ordered data structures like lists, trees, and graphs. In this paper, we discuss an extension to the AQUA query algebra to handle ordered types and their operators. We show how these operators can fit into a framework for query optimization.

1 Introduction

There are many applications [2,3,8] in which ordered types are required. Scientific applications have a need to store ordered types such as time-series data and genome sequences, and textual databases store information that is structured as a tree. These applications store huge volumes of data and must locate information from these structures very efficiently.

Query languages and algebras support declarative retrieval from a database. They are based on a set of high-level operations over collections of objects. These operations hide the looping structure that would be present in an algorithm that executes them. By and large, these operations have been confined

[*]Partial support for this work was provided by the Advanced Research Projects Agency under contract N00014-91-J-4052 ARPA order 8225, and contract DAAB-07-91-C-Q518 under subcontract F41100.

[†]Current address: Computer Science Dept., Siena College, Loudonville, NY 12211

to manipulations of sets. While there has been some recent work on extending query languages to other bulk types like sequences [5,10], additional research is needed.

This paper presents an extension to the AQUA (A QUery Algebra) query algebra [9] to include ordered types like graphs, lists, and trees. N-dimensional arrays are a topic for future work. We begin by defining algebraic operations over graphs. Graphs are used as the fundamental building block out of which the operations for the other types are derived. Sequences and trees are viewed as specialized graphs. Duplicates are introduced into these graph structures through a notion of a *cell* type.

We could argue that graphs and trees could be viewed as nested list structures, but the onus of maintaining the structure is placed on the user. For example in a tree structure, the user has to prevent two nodes from pointing to the same "child" list. Also, viewing trees and graphs as types in their own right allows us to utilize their specialized properties for query optimization and gives us more flexibility in defining operations over them. This distinction might also help in other related areas like specialized storage structures to speed access and specialized index structures for querying.

This paper focuses on an algebraic approach to queries over ordered types. An order is represented by a graph. Such an order can be restricted further to produce a tree (i.e., a partial order) or a list (i.e., a total order).

A goal of this work has been to define the operators on all the bulk types so that they are consistent with each other. The operations on a more specific version of a bulk type must follow from the operations on a more general version. For example, *Select* for a tree must be the same as *Select* for a graph when the tree is viewed as a graph. A graph with an empty edge set is essentially a set. Thus, this kind of degenerate graph must behave in all ways like a set. Identities that apply to sets must apply to graphs with no edges as well.

In several cases, an operation on an ordered type will not return an object of the starting type. We do not view this as a violation of the closure property. In our view, closure should require that an operation will return an object of a type within the model. For example, a **select** over a tree is not guaranteed to produce another tree; however, it will always produce a list of trees which is a perfectly good type in the model. Such a result can be composed with other operators for that type.

This paper first briefly introduces the AQUA data model. Next, it examines some related work, and then describes our approach to ordered types. This is followed by a discussion of the specific operators that we support for graphs, trees, and lists. We close with a few examples of how these operators are used and some suggestions for future research.

2 AQUA Model

The AQUA query algebra [9] is based on an object-oriented data model. All objects have identity, and these identities allow us to distinguish between objects using identity-based equalities.

Equality is essential to the definition of operators like union, intersection and other comparison-based operators. The default equality is identity, similar to that for unordered bulk types like sets and multisets. In the case of sets and multisets, other notions of equality are handled by providing the special operators **group** (which creates equivalence classes based on a given equality) and **choose** (which picks a member of each class nondeterministically). For ordered types, the notion of other equalities is simulated by using these equality-specific set operators on the node and the edge sets.

2.1 Constructing Types

One of the primary goals of the algebra and the model has been to support a large number of bulk types in a uniform manner. A type constructor is a metatype which defines a family of types. The *Set* type constructor defines the family of types that includes *Set[Int], Set[Department]*, etc. New types are created by instantiating the metatype *Set[T : Type]* with a specific type, like *Int* or *Department*.

AQUA provides the following type constructors: *sets, multisets, tuples, unions, functions, cells, lists, graphs,* and *trees.* The algebra also supports the *abs* constructor that allows creation of new abstract types. The *operators* of the AQUA algebra are a subset of the methods on the AQUA type constructors and include operations like **select**, **join**, and **union**.

2.2 Duplicates

All the ordered types are defined as a set of nodes N, and a set of edges E. However, as in the case of multisets, there are cases when there is a need to allow duplicates. Duplicate nodes could be handled in a manner similar to multisets but that would not allow for distinction between edges of two identical nodes. Thus, we introduce the concept of a cell. A cell can be thought of as a *wrapper* around an object that allows us to distinguish between two nodes containing the same object. With cells, we could have the same object represented as two different nodes, as the identity of the cells provides the uniqueness.

3 Related Work

Much of the previous work with ordering deals with order as in sequences or arrays. Beeri and Kornatzky [1] discuss trees in their paper. However, there is no known work with directed acyclic graphs or graphs, in the domain of database applications.

Beeri and Kornatzky propose an object-oriented query processing paradigm where the objects are built of primitive objects, an explicit object identity type constructor, and bulk type constructors. Then operations and optimizations are presented, which apply to any bulk type constructor definable in their paradigm. In this approach, lists, arrays, and trees can all be defined, and a subset of the useful operations on such structures is described in the paper. These operations include a “pump” function, which is similar to AQUA’s **fold** operation. Since the operations described in [1] are intended to be applicable to any bulk type, not just to lists and trees, they are too general for our purposes – we wish to distinguish between ordered and unordered types, and provide a richer set of operations. Furthermore, many of the operations listed in [1] are not described precisely, and their existence is assumed. Here we remove that

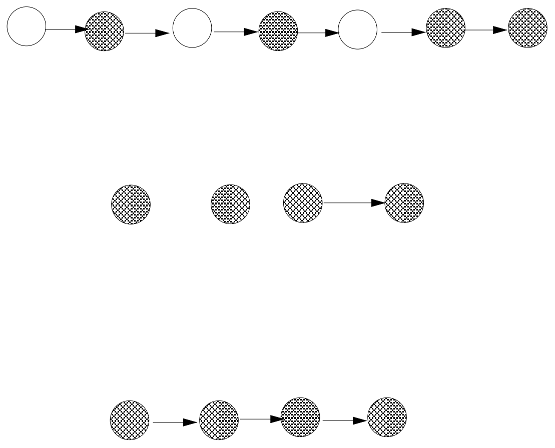

Figure 1: Select for Lists

assumption. Graphs are not discussed in [1], and it is not clear how (or if) they would fit into the paradigm presented there.

MDM [10] talks about a query algebra to support lists in an object-oriented data model. Operators from a discrete, linear-time temporal logic provide the basis for the algebra. The salient feature of the algebra is the extension of the predicate language to allow position-dependent queries, which adds a lot more flexibility to the kind of queries that can be posed to the database. Union and difference are similar to the corresponding operators in EXTRA/EXCESS [12]. However, the MDM algebra does not provide for operations on trees or graphs.

The NST algebra [6] is specifically designed for structured office documents and is an extension of relational algebra. The data model is based on nested sequences of tuples. It tries to maintain the order of the input lists whenever possible, with a higher preference for the order of the first input list. For example, in union, elements are concatenated and duplicates from the second list are eliminated. As a result, most of the operators are not commutative. Duplicates are allowed in lists, however union and intersection eliminate duplicates from the result set. A tree-like structure in a document (paragraphs under sections) is handled by treating it as a nested sequence of sequences.

Rs-operations [5] are sequence operations that are based on pattern matching. Along with these operations, sequence logic (SL), which is a first-order logic, is also introduced. Ginsburg and Wang define a set of powerful operations based on regular expressions, which act as a kind of template for the operation. However, the paper does not mention how these operations can be extended to trees. Also, the authors do not specify how these operations fit into a query optimization scheme.

The EXTRA/EXCESS system [12] contains an array type constructor; arrays can be fixed- or variable-length and can contain entities of any EXTRA type. The elements of an array are accessed using their array indices, but there is no ability to traverse from one element to another in these arrays. Operators are provided for extraction of elements and subarrays, for creating and concatenating arrays, and for applying a function to all elements of an array. The system provides no support for lists, trees, or graphs, and multidimensional arrays are constructed as arrays of arrays. Thus the arrays of EXTRA/EXCESS bear more resemblance to AQUA's N-dimensional arrays (which will be discussed in a future paper) than to the structures described in this paper.

4 Ordered Types

In this section, we describe *ordered* bulk types and the operations on them. Order in our setting is a mechanism to specify a "precedes" or "follows" relationship between pairs of elements. In its most general form, this kind of relationship can be represented as a graph, where elements are represented as nodes of the graph and edges between elements represent explicit precedence relationships. "Precede" is an antisymmetric relation, i.e. if a precedes b and b precedes a, then a and b are equivalent. As a parallel, strongly-connected components in graphs (a strongly connected component is one in which all nodes are reachable from each other) could be viewed as an equivalence class. In a sense, all nodes in the strongly-connected component are reachable from the same set of nodes in the graph and are equivalent for reachability queries.

Graphs form the basis for our definition of ordered types. Lists and trees are specialized forms of graphs. Besides the obvious restriction that the underlying structure be a tree (or a list), we impose an additional constraint of transitivity. In other words, if there is a directed path $a_0, a_1, \cdots, a_n$ in the tree (list), we assume that there is an implicit edge between a_0 and a_n. Note that these implicit edges are not actually present in the tree structure. To see why this transitivity assumption is *natural* when viewing lists, consider selecting birds from a list of animals A = [cat crow mouse sparrow dog robin parrot] (Figure 1). Treating list A as a graph would give us a set of graphs instead of a list [1]. Most of us would expect the select to return [crow sparrow robin parrot].

However, note that the edges between crow and sparrow & sparrow and robin did not exist in the original list though they are there in the *expected* result. The implicit assumption here is that the relationship between the elements is transitive. A similar example can be used to show that transitivity is a natural assumption for trees as well. Note that the *transitivity* property ensures that the resultant type is the same as the input type (Lists $\mapsto$ Lists and Trees $\mapsto$ Trees or a set of Trees). As a result of this property, the behavior of the operators for graphs is slightly different than that for lists and trees.

Sets in the AQUA model are at the other end of the spectrum; they can be viewed as the most *unordered* form of graphs, as sets can be represented as graphs with empty edge sets. We explore this connection in greater detail in section 5.4.

[1] Assuming we ignore the typing issues for the moment.

4.1 Graphs

Graphs are defined as a set V of nodes, and a set E of directed edges between the nodes. An edge is defined as a pair of nodes and the direction of the edge is from the first node to the second.

The type constructor for graphs is defined as *Graph[T]*, which constructs a graph type consisting of objects of type T as the nodes. Edges in the graph are tuples consisting of a pairs of nodes of type T. As mentioned earlier, this definition does not allow duplicate objects as nodes. Duplicates are handled by using a graph of type *Graph[Cell[T]]*. Since this is used later, while defining type conversion operators on trees and lists, we use the term "cell-graphs" to refer to such graphs. Graphs, like sets and multisets, do not participate in sub-typing.

4.2 Trees and Lists

Trees and lists are also defined as a set V of nodes, and a set (or a list in the case of ordered trees) E of directed edges between the nodes. However, the underlying structure of a tree (list) instance must be a tree (or a list). Assuming edges are directed away from the root, this implies that a tree must have one node with no incoming edges and all other nodes must have a single parent (or incoming edge). For a list, there must be one node with no incoming edge and a node with no outgoing edge (except for the empty list). All other nodes in a list must have one incoming edge and one outgoing edge.

The AQUA model supports two kinds of trees, *ordered-* and *unordered-trees*. Ordered-trees are trees where there is an order between the children of a node and unordered-trees assume that there is no explicit order between the children. Ordered-trees are defined as a set V of nodes and a list E of directed edges (as opposed to a set of directed edges for unordered-trees). The relative ordering among the edges from the parent node to the child node in the list E determines the order of the children nodes.

A tree (or list) of type T consists of nodes of type *Cell[T]* and the edges between these nodes. The node typing is different from that of graphs, where the nodes are of type T. We do this since it allows us to handle duplicates in a consistent manner. Duplicates in trees and lists present a problem when dealing with operators that could possibly map two or more nodes of the original tree onto the same object. In such a scenario, preserving all the associated edges might violate the tree (or list) structure. As a result, we adopt the "cell" structure to avoid duplicate nodes.

The type constructor for trees is defined as *Tree[T]*, which is a tree type consisting of nodes of the same type *Cell[T]* and edges between these nodes. The type constructor for lists is similar, *List[T]* is a list type consisting of nodes of type *Cell[T]* and their associated edges. Lists and trees do not participate in subtyping.

5 Operators

In this section, we describe in detail the various operations on graphs, trees and lists. The functionality of most operators is similar across all the ordered types. The syntax of the operations is similar to that used in AQUA[9], and is based on lambda calculus.

Predicates are functions with *boolean* return type, and are composed using AQUA's built-in operators and its term language (which is based on lambda calculus). Predicates are passed as parameters to operators like **select**.

Cells have two operators: **Cell**(a) creates a cell containing a. **Cell_content**(c) returns the object contained in cell c.

5.1 Graphs

In this subsection we describe the operators on graphs. Table 1 details all the operator definitions. We now describe some of the notation used in the table. The input graphs are $G = (V_G, E_G)$ and $H = (V_H, E_H)$ and the output graph is $R = (V_R, E_R)$. Individual nodes are denoted by lowercase letters, with the graph name as a subscript (for example, u_G, v_G, x_G). Predicates are indicated by p and f represents a function.

The primary query operators are **select**, **apply**, **union**, and **intersect**. Both **union** and **intersect** use the default equality for unioning (or intersecting) the node and edge sets. **Im_ancestor** and **im_descendant** are the traversal operators. The algebra also defines other *support* operators like **nodes** and **edges** along with *update* operators like **add_node**, **add_edge**, and **delete_edge**. The algebra also has *conversion* operators to convert from a graph (of the appropriate structure) to a tree or a list.

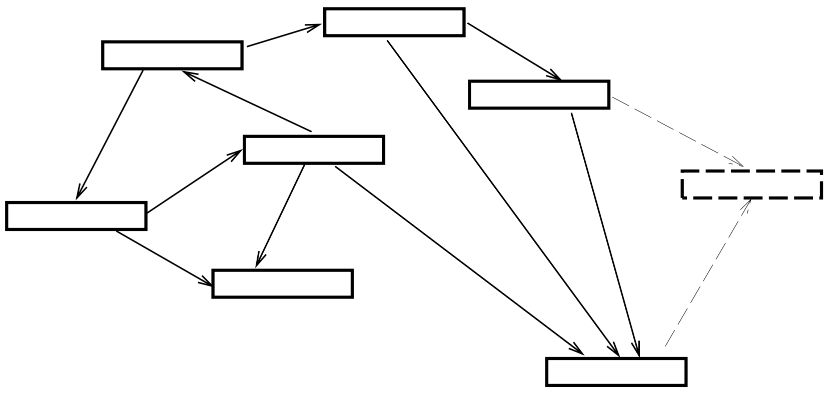

Figure 2: Flight graph for *Airline*1

As an example, consider an airline company, Airline1. They have a large number of airports out of which they operate, with flights connecting them (figure 2). In our graph, the nodes represent the airports and the edges represent the flights between the airports. The nodes are assumed to have more details about the airports, besides their city names.

```
type $Airport = abs(Tuple[city : String, ...]$
    $city(Airport) \rightarrow String;$
                    ⋮
```

Definitions		
$\mathbf{select}(p)(G)$	$=$	$(V_R = \{v_R \mid p(v_G)\},\ E_R = \{(u_R, v_R) \mid ((u_R, v_R) \in E_G) \wedge p(u_R) \wedge p(v_R)\})$
$\mathbf{apply}(f)(G)$	$=$	$(V_R = \{f(v_G)\},\ E_R = \{(f(u_G), f(v_G)) \mid (u_G, v_G) \in E_G\})$
$\mathbf{union}(G, H)$	$=$	$(V_R = \mathbf{union}(V_G, V_H),\ E_R = \mathbf{union}(E_G, E_H))$
$\mathbf{intersect}(G, H)$	$=$	$(V_R = \mathbf{intersect}(V_G, V_H),\ E_R = \mathbf{intersect}(E_G, E_H))$
$\mathbf{im\_ancestor}(x_G)(G)$	$=$	$\{v_G \mid (v_G, x_G) \in E_G\}$
$\mathbf{im\_descendant}(x_G)(G)$	$=$	$\{v_G \mid (x_G, v_G) \in E_G\}$
$\mathbf{sources}(G)$	$=$	$\{v_G \mid (x_G, v_G) \notin E_G\}$
$\mathbf{sinks}(G)$	$=$	$\{v_G \mid (v_G, x_G) \notin E_G\}$
$\mathbf{nodes}(G)$	$=$	V_G
$\mathbf{edges}(G)$	$=$	E_G
$\mathbf{graph}(x)$	$=$	$(V_R = \{x\},\ E_R = \emptyset)$
$\mathbf{t\_closure}(G)$	$=$	$(V_G,\ E_R = \{(u_G, v_G) \mid \mathit{connected}(u_G, v_G)\}$ where $((u_G, v_G) \in E_G) \vee (u_G = v_G) \Rightarrow \mathit{connected}(u_G, v_G)$ and $((u_G, m_G), (n_G, v_G) \in E_G) \wedge (\mathit{connected}(m_G, n_G))$ $\Rightarrow \mathit{connected}(u_G, v_G))$
$\mathbf{append}(u_G, v_H)(G, H)$	$=$	$(V_R = \mathbf{union}(V_G, V_H),\ E_R = \mathbf{union}(\{(u_G, v_H)\}, \mathbf{union}(E_G, E_H)))$
$\mathbf{add\_node}(x)(G)$	$=$	$(V_R = \mathbf{union}(V_G, \{x\}),\ E_G)$
$\mathbf{delete\_node}(x_G)(G)$	$=$	$(V_R = \mathbf{diff}(V_G, \{x_G\}),\ E_R = \mathbf{diff}(E_G, \mathbf{union}(\{(u_G, x_G)\}, \{(x_G, v_G)\})))$
$\mathbf{add\_edges}(S)(G)$	$=$	$(V_G,\ E_R = \mathbf{union}(E_G, \{(u_G, v_G) \mid ((u_G, v_G) \in S) \wedge (u_G, v_G \in V_G)\})$
$\mathbf{delete\_edges}(S)(G)$	$=$	$(V_G,\ E_R = \mathbf{diff}(E_G, S))$
$\mathbf{replace\_node}(x_G, y)(G)$	$=$	$\mathbf{add\_edges}(A)(\mathbf{add\_node}(y)(\mathbf{delete\_node}(x_G)(G)))$ where $A = \mathbf{union}(\{(y, v_G) \mid (x_G, v_G) \in E_G\}, \{(u_G, y) \mid (u_G, x_G) \in E_G\})$

Table 1: Graph Operators

)

type $Airline = Graph[Airport]$

Our query is to find all the places that have a direct flight to Boston, either by $Airline1$ or $Airline2$. The basic query is to get all the **im_ancestors** of the node *Boston* in the combined airline map, $TwoAirlines$. The combined airline map is obtained by unioning the maps of $Airline1$ and $Airline2$.

TwoAirlines = **union**($Airline1$, $Airline2$)

$$\text{DirectToBoston} = \textbf{im\_ancestor}(\textbf{choose}(\textbf{nodes}(\textbf{select}(\lambda(n)\, n.city = Boston)(TwoAirlines))))(TwoAirlines)$$

5.2 Trees

In this section, we briefly describe the operators on trees. Most of the tree operators have been derived from the corresponding graph operators, so in the following paragraphs we shall highlight the differences between the corresponding graph and tree operators. We discuss the operators for ordered-trees below; operators for unordered-trees follow logically from the ordered-tree operators. Therefore, for the sake of simplicity we use the short-form "tree" to refer to ordered-trees. Also, in all our examples we assume that all edges are directed away from the root of the tree.

The basic tree operators in AQUA are: **select** (p) (T), **apply** (f) (T), **sub_select** (r) (T), **PT** (T), **all_desc** (r) (T), **all_anc** (r) (T), **sources** (T), **sinks** (T), **nodes** (T), **edges** (T), **e_edges** (T), **im_ancestor** (x)(T), **im_descendant** (x) (T), **tree** (x), and **find_path** (x) (T). The update operators are: **append** (*from_node*, *sibling*) (T_1, T_2), **add_node** (x, *parent*, *sibling*) (T), **delete_node** (x) (T), **replace_node** (x, y) (T), and a number of *conversion* operators to go from a tree to a graph or a list.

Note that some functions are defined only on graphs: **union**, **intersect**, **t_closure**, **add_edges** and **delete_edges**. This is mainly because the result of these operations will not be a tree (the resultant structure will be a graph). However, the functionality of the operators can be obtained by converting the input trees to graphs and applying the corresponding graph operators. For example, **union** on two trees with common nodes might produce a graph due to unioning the edge sets of the common nodes from the two input trees. This operation however, can be performed by "converting" trees to graphs.

The main differences between the other tree operators and their corresponding graph operators are:

- The children of a node in a tree are ordered. As a result, the operators return a list of nodes or sub-trees instead of a set. For example, **sources**(T), **sinks**(T), **im_ancestor**(x)(T), and **im_descendant**(x)(T) are similar to the corresponding graph operators, except that the result is a list of nodes. So, **source** returns a singleton list consisting of the root of the tree, **sinks** returns a list consisting of all the leaves of the tree, **im_ancestor** returns a list containing the parent of the given node

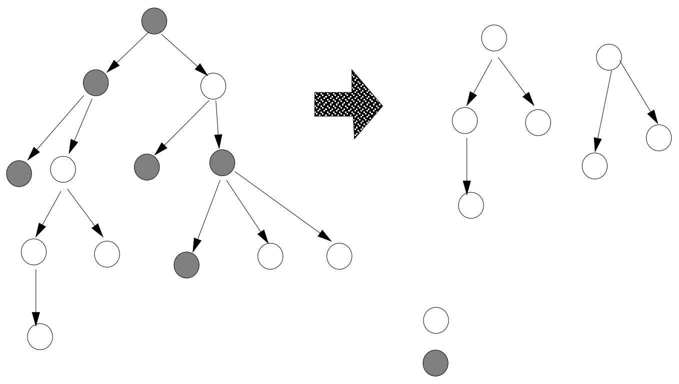

Figure 3: Select on a tree

in the tree and **im_descendant** returns a list of all the children of the given node, in order.

- The edges of a tree are transitive. Therefore, **edges** operator returns a list containing all the edges of the tree – this includes the edges that were explicitly added to the tree and the edges that were "created" due to the transitive property of the edges. The **e_edges** operator, in contrast, returns a list containing only the edges that were explicitly added to the tree. For example, **edges** on the tree rooted at 5 (figure 3) would return $[\{5,8\},\{5,9\},\{8,13\},\{5,13\}]$ and **e_edges** would return $[\{5,8\},\{5,9\},\{8,13\}]$. This property also influences any operator that "deletes" nodes (**select** and **delete_node**). Deletion of a node causes an implicit edge, i.e. an edge created due to transitivity, to become an explicit edge which can be loosely thought of as the edges used to draw the tree. So, if we just look at explicit edges, a deletion causes addition of new edges (from the list of all edges) between the the parent and the children nodes of the deleted node (Figure 4).

  Select$(p)(T)$ selects a list of sub-trees of tree T, based on the nodes that satisfy predicate p. All the edges between selected nodes from the input tree are present in the resultant graph. Any new edges "created" due to the transitivity relationship between the nodes are also added in the resultant tree (Figure 3). The ordering in the resultant list is based on the *relative* ordering of the roots of each tree in the list. For example, in figure 3, the tree with node 5 as the root comes "before" the tree rooted at node 3, in spite of the difference in levels. The ordering is mainly based on position – if sub-tree A is to the left of sub-tree B (assuming ordering is from left to right), then A is followed by B in the resultant list. It is a kind of depth-first ordering. The sub-trees in the list are ordered based on the relative order in which the respective roots of the sub-trees are visited in a depth-first traversal.

- For certain operators, there are certain constraints on their behavior, as the result has to be a tree. For example, **add_node** adds a node and an edge connecting the node to the tree (at the specified point).

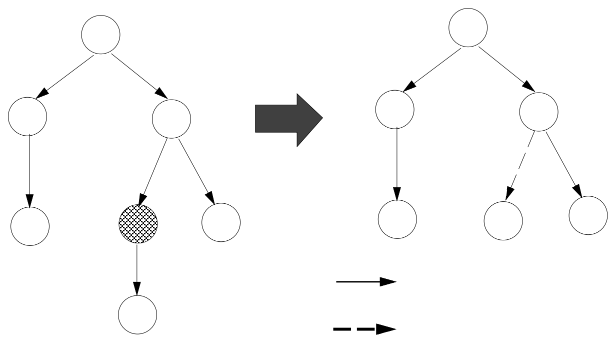

Figure 4: Deletion in a tree

In the case of **append**, the *to_node* is always the root of the second tree, hence it is not a parameter to the operator. **Append** (*from_node*, *sibling*) (T_1, T_2) appends tree T_1 to tree T_2, as a child of the *from_-node*, after the sub-tree rooted at *sibling* (which is also a child of the *from_node*). If *sibling* is not specified, the tree T_2 is added as the first child of the *from_node*.

- Trees of type T are composed of cells that contain objects of type T. As a result, most function applications deal with the contained-object instead of the cell. So, as in the case of **apply** on a graph, **apply** on trees transforms the "contained-object" based on the parameter function f.

  Apply$(f)(T)$ applies the function f to the "content" of each cell (node) of the tree to transform the existing object into a new object. The edge-set remains the same. This ensures that the basic structure of the tree is not modified. The resultant tree is built of new cells that contain the transformed objects (Figure 5).

Other operators that are specific to trees are: **Tree**(x) creates a tree that consists of node x. **Find_path**$(x)(T)$ returns a list of nodes encountered on the path from the root of the tree T to the node x. The last node of the resultant list is x, and the first node is the root of the tree. **Sub_select**$(r)(T)$ returns a set of all sub-trees of tree T that match the pattern r. The **PT**(T) operator (powertree) takes as input a tree T and returns a set of all sub-trees of T. This operator is somewhat similar in spirit to the power-set operator for sets and is used primarily for defining other more specific operators. For example, **sub_select** can be expressed as:

$$\mathbf{sub\_select}(r)(T) = \mathbf{set\_select}(\lambda(x)\ x \in \mathcal{L}(r))(\mathbf{PT}(T))$$

PT generates a set of all subtrees of T and **set_select** (**select** over a set) selects those subtrees that are in the tree language defined by the tree regular expression r ($\mathcal{L}(r)$). **All_desc**$(r)(T)$ and **all_anc**$(r)(T)$ are specialized cases of the **PT** operator that extract all maximal subtrees of T that start (**all_desc**) or end (**all_anc**) with the pattern r. Specification of the match pattern r is discussed in detail in subsection 6.2.

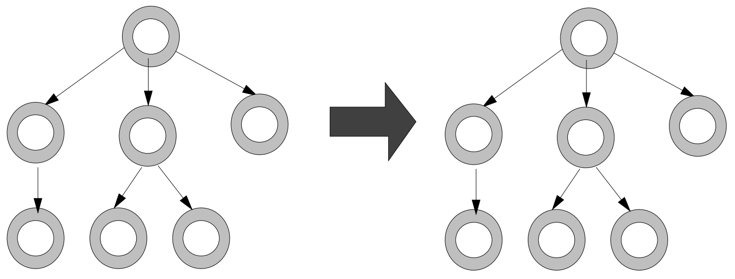

Figure 5: **Apply**$(\lambda(x)\,A)(T)$

Replace_node and **nodes** are similar to the corresponding operators in graphs. The only difference is due to typing of the input and the output. For example, **nodes** on a tree returns a set containing all the nodes of the tree, similar to **nodes** for graphs. However, the tree-nodes are cells unlike graph-nodes which are objects.

5.3 Lists

In this section, we discuss operations on lists. These operators are almost identical to the corresponding operators on trees, except for the input and output types which are lists instead of trees, and the absence of the "sibling" parameter. Any kind of add operation in trees requires a *sibling* parameter, that specifies the node after which the new node/sub-tree must be added. This is needed for trees as the children of a node are ordered. In the case of a list however, since there is only one child for every node, this parameter is unnecessary.

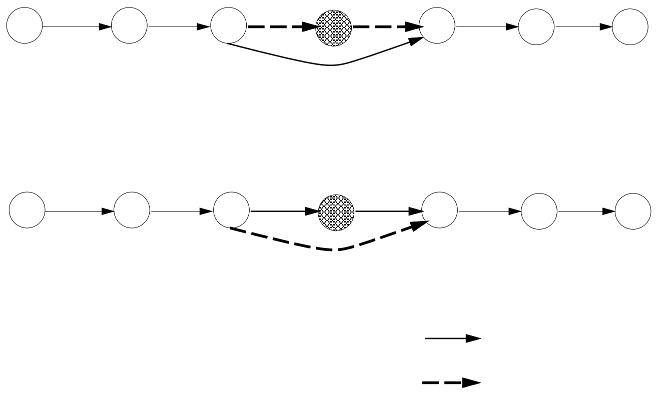

Figure 6: Add_node and Delete_node on a list

Select (p) (L), **apply** (f) (L), **sub_select** (r) (L), **PL** (L) (similar to **PT**), **all_suffix** (r) (L) (similar to **all_desc**), **all_prefix** (r) (L) (similar to **all_anc**), **nodes** (L), **edges** (L), **e_edges** (L), **add_node** (x, y) (L), **replace_node** (x, y) (L), **delete_node** (x) (L), and the *conversion* operators are similar to the corresponding unordered tree operators. List predicates are discussed in greater detail in subsection 6.1. **Add_node** in a tree does not involve deletion of any existing edges, however, in the case of lists adding a node in the middle of the list might result in deletion of an edge and addition of up to two edges (Figure 6). **Source**(L), **sink**(L), **im_ancestor** (x) (L), **im_descendant**$(x)(L)$ are also similar to their equivalent graph operators but they return a single node for lists instead of a list of nodes as in trees. **Append**(L_1, L_2) is similar to **append** in trees and graphs, but the *from_node* is always the last node of list L_1 and the *to_node* is the first node of list L_2. This results in a concatenation of the two input lists.

There are list operators that do not have corresponding tree operators. **List**(x) creates a list with a single element x. **Sort**$(f)(L)$ sorts list L based on the comparator function f. f is a transitive function that given any two elements (a, b) from the list L, returns *less-than* if a should appear before b in the result list, *greater-than* if a should appear after b in the result list and *equal-to* if a and b are "equal" based on the function f.

5.4 Sets as graphs

The AQUA model supports various bulk types, among them sets and graphs. Sets can be viewed as graphs with an empty edge set. With this view, all the operators for graphs neatly transform into the corresponding operators for sets. For example, **union** and **intersection** on graphs with empty edge sets are similar to these operations on sets. Similarly, **apply** and **select** behave the same way for graphs with an empty edge set, as with sets. This makes the addition of ordered types into the model seamless and consistent with the other bulk types.

6 Predicates

Ordered bulk types, unlike sets and multisets, have the notion of "position" of the constituent objects. This opens up possibilities of having a more powerful predicate language. The remainder of this section describes a richer set of predicate formers for lists and trees.

This investigation was motivated by the observation that query optimization is facilitated by identities that allow us to break a predicate into pieces, some of which can be evaluated cheaply. These pieces must be composable to produce the original query. We give examples of this type of decomposition in the context of our pattern-based predicate languages.

6.1 List Predicates

In this subsection we discuss order-based predicates for lists. These predicates are based on regular expressions (for describing match patterns) and lambda calculus. However, we do envisage a more user-friendly interface that would translate user-defined queries to algebra queries based on regular expressions.

We use the standard notations for specifying regular expressions – "(" and ")" are used for specifying precedence, $|$ for disjunction (union), * for Kleene star, $\circ$ for concatenation, $\widehat{}$ to mark the beginning of the list, and \$ to mark the end of the list. We use ab as a shorthand for $a \circ b$, which stands for list a concatenated with list b. As an example, consider the regular expression $R = (ab)^* \mid a^+$, defined over strings of characters. R defines a language that contains strings formed by either repeating the pattern ab zero or more times or by repeating a one or more times. Instances of strings in this language are $\varnothing$, aaa, ab, $abab$. $\varnothing$ denotes the null string and $+$ is similar to the Kleene star and the language defined by x^* is the language defined by $x^+ \mid \varnothing$, for any regular expression x. We can also specify wild cards (or *don't cares*) by using the symbol ?, which acts as a placeholder for one symbol. $?^*$ matches zero or more symbols; so $?^*a$ matches all strings ending in a. We also use the terms string and sequence to signify a list composed of immutable characters.

To illustrate the use of these predicates, consider a sample query that finds all sub-sequences of a sequence that match a particular pattern $a?t?^*tg$. Such a query could potentially find use in a genome sequence database, where we are searching for a particular protein sequence or a gene. The above query would translate to **sub_select**$(a?t?^*tg)(L)$ in the query algebra. **Sub_select** selects all substrings of the list L that match the input pattern. So, if $L = [acctcggagtccccacttg]$, then **sub_select**$(a?t?^*tg)(L)$ would return set $\{[agtccccacttg], [acttg]\}$, containing the two sub-sequences that match the regular expression $a?t?^*tg$.

This query can also be expressed in terms of other operators like **PL**, **all_suffix**, and **all_prefix** (subsection 5.3). This provides the query optimizer with numerous options for rewriting the query, depending on the cost-effectiveness of the resultant query. The **PL** operator returns a set of all the possible sub-lists of the input list. **All_suffix** and **all_prefix** are specialized forms of the **PL** operator. They return maximal substrings (i.e. the portion of the list from a given point till one of the ends) starting with the pattern or ending with the pattern respectively, for each occurrence of the pattern. These two operators are very useful for establishing the position(s) of the pattern in the list as they are always anchored at one of the end-points. For example,

$$\begin{aligned}\mathbf{PL}([abc]) &= \{[a],[b],[c],[ab],[bc],[abc]\}\\ \mathbf{all\_suffix}([e?g])([abcdefghidefgh]) &= \{[efghidefgh],[efgh]\}\\ \mathbf{all\_prefix}([ef])([abcdefghidefgh]) &= \{[abcdef],[abcdefghidef]\}\end{aligned}$$

In the next few paragraphs we illustrate some possible query transformations using the **sub_select**$(a?t?^*tg)(L)$ query as an example.

One possible way of expressing the same query in terms of **PL** is:

$$\mathbf{set\_select}(\lambda(l)\, l \in \mathcal{L}(a?t?^*tg))(\mathbf{PL}(L))$$

PL(L) returns all the possible substrings of L. We then use the **select** operator over sets (aliased to **set_select** to avoid any ambiguity) and the list predicate to pick the sub-sequences that match the pattern.

Now suppose we already have an index on all the positions of the symbol a in the sequence L. We could then rewrite the query to take advantage of this information in the following manner:

$$\textbf{apply}(\lambda(l)\,\textbf{all\_prefix}(\hat{}a?t?^*tg)(l))(\textbf{all\_suffix}(a)(L))$$

All_suffix takes advantage of the index on the input list L and can therefore be computed very quickly. Also, the result reduces the positions we need to check for a match. So, to obtain the final result we need to check if the lists (in the set of lists obtained by the **all_suffix** operation) start with (denoted by $\hat{}$) the pattern $\hat{}a?t?^*tg$, using the **all_prefix** operator. **All_prefix** extracts the sub-sequences that are in the language defined by the regular expression $\hat{}a?t?^*tg$. Similarly, if we had an index on tg, we could rewrite the query as:

$$\textbf{apply}(\lambda(l)\,\textbf{all\_suffix}(a?t?^*tg\$)(l))(\textbf{all\_prefix}(tg)(L))$$

In a similar manner, we can rewrite the query to take advantage of indices on both a and tg.

$$\begin{aligned}&\textbf{set\_select}(\lambda(s)\,s \in \mathcal{L}(\hat{}a?t?^*tg\$))\\&\qquad(\textbf{apply}(\lambda(l)\,\textbf{all\_suffix}(a)(l))(\textbf{all\_prefix}(tg)(L)))\end{aligned}$$

A slightly more complex strategy can be used if we only have an index for occurrences of t in L. Assume that the list L is split into two lists L_1 and L_2 such that $L_1L_2 = L$ and L_2 starts with t (using the index). So, for each such split we have to check if the query below is non-empty and in such a case, return the matching sublist.

$$\textbf{set\_select}(\lambda(l)\,l \in \mathcal{L}(?^*a?))(L_1) \wedge \textbf{set\_select}(\lambda(l)\,l \in \mathcal{L}(t?^*tg?^*))(L_2)$$

Another interesting case is querying over a set of lists S to check if a particular pattern $a?t?^*tg$ exists in any of the lists. This can be expressed as:

$$\textbf{set\_select}(\lambda(l)\,l \in \mathcal{L}(?^*a?t?^*tg?^*))(S)$$

As in the earlier examples, we can use any indices for query rewrites. If we have an index into the set S indicating the lists that contain the symbol t (or any sub-string of the match pattern), we could rewrite the above query as:

$$\textbf{set\_select}(\lambda(l)\,l \in \mathcal{L}(?^*a?t?^*tg?^*))\;(\textbf{set\_select}(\lambda(s)\,s \in \mathcal{L}(?^*t?^*))(S))$$

The first **set_select** uses the index and as a result reduces the input size for the second **set_select**. We could also use multiple indices in the same way:

$$\begin{aligned}&\textbf{set\_select}(\lambda(l)\,l \in \mathcal{L}(?^*a?t?^*tg?^*))\\&\qquad(\textbf{set\_select}(\lambda(s)\,s \in \mathcal{L}(?^*t?^*))(S) \cap\\&\qquad\;\textbf{set\_select}(\lambda(s)\,s \in \mathcal{L}(?^*g?^*))(S))\end{aligned}$$

6.2 Tree Predicates

Recall that the standard **select** operator is defined to return a set of nodes based on the properties of the contents of those nodes. The **sub_select** operator returns all subtrees of a tree that satisfy a certain property. In other words,

sub_select takes connectivity and structure into account while **select** does not.

Consider the query "retrieve all the portions of this family tree in which somebody named a is an ancestor of somebody named b". In this case we are searching for any subtree which matches the predicate "somebody named a is an ancestor of somebody named b". In the case of lists, similar conditions can be stated using regular expressions. To extend the standard regular expression notation to trees, we build on the results of [4,11]. The basic notation is the same as that of regular expressions: * for Kleene closure, | for union (disjunction), and $a \circ b$ for "a concatenated with b" (we use ab as a shorthand for $a \circ b$). The only fundamental difference is in the meaning of the concatenation operator. In a regular expression, which always represents a string (i.e., a total ordering), ab simply means that b follows immediately after a. However, a node in a tree may have more than one successor (child).

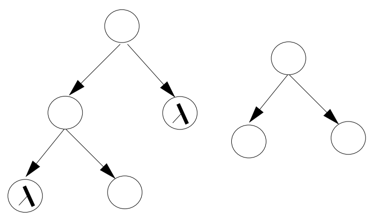

Figure 7: Multiple concatenation points

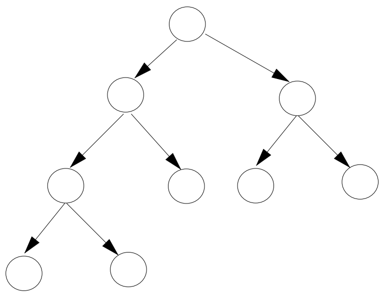

Figure 8: Result of figure 7

To introduce our notation, we must remember that there is a clear distinction between terms that represent trees and terms that represent patterns, just as there is a difference between a regular expression and an actual string. For example, consider the string "$abacdeb$" and the regular expression "b^*de". The regular expression matches the substring "$bacde$" of the first string. As a simple example of our notation, consider the tree (not pattern) represented by "$a\,(b\;c)$". This is a tree with "a" at the root, "b" at the left child, and "c" at the right child. In our notation for trees, a node is followed by "(" then by its children, then by ")". This corresponds to a preorder listing of the nodes. We do not consider unordered trees in this section, although most of the ideas apply there as well.

As another example, consider the tree term $T =$ "$a\,(b\,(d\;e)\;c\,(f\;g))$", representing a full binary tree with three levels. The notation for tree *patterns* extends the simple tree notation in a manner similar to the extension made to strings by regular expressions. In what follows, it should be clear from the context whether a tree or a pattern is being described by a particular term. As a simple example, consider the *pattern* represented by the term "$a\,(b\;c)$". It matches a *subtree* of T which is represented by the term "$a(bc)$". Note that just as in the matching of substrings to regular expressions, we are not interested in what *follows* the matching subtree in T. We are only interested in finding the matching subtree, just as in the above regular expression example we noted that the matching substring is "$bacde$", not "$bacdeb$". The **sub_select** operator is defined to return the matching subtrees, not what follows them (see examples below).

To represent concatenation in a pattern, we use a special symbol to indicate the *concatenation points* – the points in the expression where the second term is to be appended to the first. We first illustrate this graphically and then describe the algebraic notation. In figure 7 we have two trees. The special symbol λ indicates a concatenation point and must appear at the leaves. The concatenation of the left and right trees in figure 7 gives the result in figure 8. Note that λ appears twice in the left tree. The meaning of the concatenation point is that all occurrences of the concatenation point are to be replaced with the tree on the right, giving the result in figure 8.

The union operation on tree patterns is no different from its regular expression counterpart. The Kleene operator * is based on the concatenation of one pattern onto itself, any number of times. Thus it also needs to make use of the notion of concatenation points. As an example, consider a tree with three nodes, a at the root, b at the left child, and λ at the right child, and call it T. Then some of the elements of T^* are shown in figure 9.

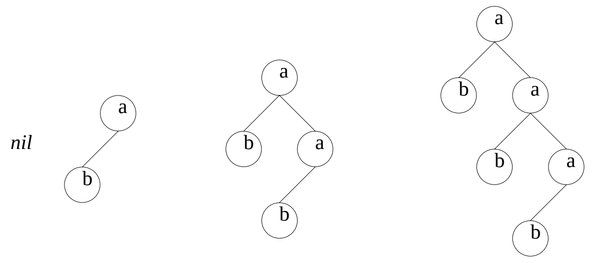

Figure 9: Part of a Kleene closure

We now describe the syntax that will enable us to express these patterns inside an algebraic query. The basic idea is that any concatenation, including one engendered by *, must be given one or more instances of λ as concatenation points. As an example, the concatenation of the trees of figure 7 is described as follows in our syntax, where square brackets are used for grouping:

$$[[a\,(b\,(\lambda\;\;f\;)\;\lambda)]\;[c\,(d\;\;e\;)]]$$

Only a node together with all of its children may be the subject of concatenation or *. The trees in figure 9 are a subset of

$$[a\ (b\ \lambda\ )]^*$$

Let us examine a more complicated example of concatenation. Consider the following pattern:

$$[[a\ (\lambda\ \ \lambda\ )]\ [\ [c|d]\ (e\ f)]]$$

Formally, the result of a concatenation is defined as the set of all trees formed by replacing every λ in every tree matching the first pattern with a tree matching the second pattern. Not every λ need be replaced by the same tree from the second set, but every λ must be replaced by one of them. Figure 10 shows two of the four trees that satisfy the previous pattern.

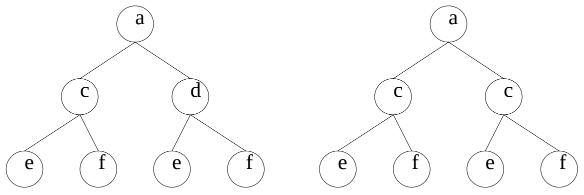

Figure 10: Two concatenations of $[a(\lambda\ \lambda)]\ [[c\mid d](e\ f)]$

The symbol ? indicates that the contents of a node can be anything. Thus a pattern with any value at the root and the values b and c as the left and right children, and no other nodes, would be expressed as

$$[?\ (b\ \ c\ )]$$

This ? symbol, however, stands only for any possible *contents* of a node, not for any possible subtree. In other words, ? represents a tree with one node, whose contents are unknown. An important special case of tree patterns are those in which we are only interested in the structure of the tree [7], not in the contents of the nodes. The ? symbol makes such patterns easy to express in our notation. To represent the most general tree pattern, which will match any tree at all, we need an analog to the "?*" of regular expressions. We define the symbol $T?$ to stand for any binary tree as follows:

$$T? = [?(\lambda\ \lambda)]^*$$

Now we present some examples of queries that select from some tree T all subtrees matching a given pattern. Recall that **sub_select** returns exactly the subtree(s) matching the pattern, and does not return descendants of those subtrees. As a realistic application, consider a relational query optimizer which represents queries as trees. All operators are unary or binary, and to simplify the presentation we will ignore additional parameters. Given some query tree Q, the following AQUA algebra expression returns all subtrees of Q representing a join whose left input is also a join:

$$\textbf{sub\_select}(join\ (join\ ?))(Q)$$

We do not specify the children of the inputs to the first join because this query is intended to return only the portion of the tree with this structure, not any of its children.

Now consider a query to retrieve all subtrees of Q representing a join whose left input contains a selection somewhere in it:

$$\textbf{sub\_select}(join\ ([[?\ [(\lambda\ ?)\ |\ (?\ \lambda)]]^*\ select]\ ?))(Q)$$

The disjunction ensures that any subtree containing a selection will match the pattern. Intuitively, it ensures that any number of left and right "turns" leading to a selection will qualify.

In these examples we have been assuming very simple node contents – immutable strings. However, the syntax easily accommodates arbitrarily complex node contents. Any algebraic expression which evaluates to something of the appropriate type can be used to specify the contents of a node inside a tree pattern. It would also be possible to define an extended version of **sub_select** which takes an additional parameter indicating an algebraic expression to be applied to each node. The result of this expression, rather than the actual node, could then be matched against the pattern.

The **sub_select** operator can clearly find all occurrences of any tree pattern inside any tree. However, there are other ways of expressing the same queries. We now use the **PT** and **all_desc** operators to define alternative ways of expressing some queries.

The **PT** (powertree) operator takes a tree T and returns all subtrees of T. The definition of "subtree" is analogous to the definition of "substring". **All_desc** (r) (T) retrieves all subtrees of T which *start with* the pattern r and include all descendants of that occurrence of r.

One motivation for the **all_desc** operator can be illustrated by the following example. Consider the tree T of figure 11 and the query:

$$\textbf{sub\_select}(e\ (a\ b))(T)$$

This query can be rewritten using **all_desc** as follows:

$$\textbf{collapse}(\textbf{apply}(\lambda(s)\ \textbf{sub\_select}(e\ (a\ b))(s))\ (\textbf{all\_desc}(e)(T)))$$

This version of the query first finds all subtrees of T whose root contains simply "e" and whose descendants go as far down T as possible. The query then finds all subtrees of each of these subtrees that have "a" and "b" as the children of "e".

This query might be "cheaper" when we have an index that will return all nodes containing "e". In that case, the **all_desc** operation makes direct use of the index to compute its result. The **sub_select** operations, in this case, will only be examining subtrees with the proper root node. Assuming the situation of figure 11, in which there may be thousands of nodes in the outlined region R, none of which contain "e", the processing time for the query is potentially orders of magnitude faster than in the initial version. Note that the rewriting used above does not always result in a more efficient execution of the query, even if an index is used. For example, if "e" occurred many times in the same large subtree, many copies of parts of that subtree would be returned, resulting in a potentially longer search than with the original **sub_select** query.

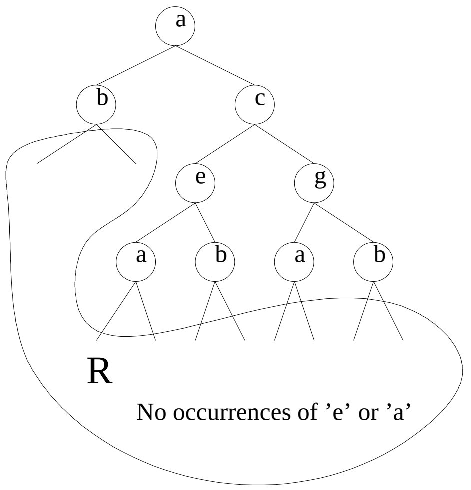

Figure 11: A large tree

Suppose now that we have available an index which provides fast access to all nodes containing "e" that also have "a" as their first child. A similar rewriting of the query can facilitate the use of such an index:

$$\textbf{collapse}(\textbf{apply}(\lambda(s)\ \textbf{sub\_select}(e\ (a\ b))(s))\ (\textbf{all\_desc}(e\ (a\ ?))(T)))$$

In this case, the index is even more restrictive, leaving even less work for the expensive (**sub_select**) portion of the query.

AQUA also provides an **all_anc** operator, which travels up the tree in the way that **all_desc** travels down the tree. In other words, **all_anc** (r) (T) will retrieve a set containing all occurrences of r in T alongwith their paths from the root to r. We omit a complete description of **all_anc** here due to space limitations.

7 Conclusions and Future Work

This paper has described the support for ordered bulk data types provided by the AQUA data model and algebra. The primary ordered bulk type is a graph, from which we derive trees and lists by imposing constraints on the edge set. Uniqueness of tree and list nodes is enforced using the *Cell* type constructor. Important aspects of AQUA's ordered bulk type support are the consistency of operators and semantics among the various ordered types and the close relationship between graphs and sets, resulting in a very uniform data model.

We have further described a simple predicate language for lists and trees that supports queries that depend on order. This formalism is based on regular expressions, but could be extended to more expressive pattern languages such as context-free grammars.

Current and future research includes investigation of additional operators on ordered bulk types (e.g. LFP, as described in [9]) and implementation techniques for indexing over ordered bulk types. We are presently looking at ways to extend our tree pattern language to work with DAGs. Indexable ordered types in AQUA (such as N-dimensional arrays) will be discussed in a future paper.

Acknowledgments

We would like to thank Catriel Beeri, Gail Mitchell, Arnold Rosenberg, and Sairam Subramanian for useful discussions.

References

[1] Catriel Beeri and Yoram Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages," *Proceedings of the International Conference on Database Theory* (1990), 72–83.

[2] James C. French, Anita K. Jones, and John L. Pfaltz, "Summary of the Final Report of the NSF Workshop on Scientific Database Mgmt.," *SIGMOD Record* 19 (1990), 32–40.

[3] Karen A. Frenkel, "The Human Genome Project and Informatics," *Communications of the ACM* 34 (1991), 41–51.

[4] Johann C. Freytag, "Tree Acceptors and Some of their Applications," *Journal of Computer and System Sciences* 4 (1970), 406–451.

[5] Seymour Ginsburg and Xiaoyang Wang, "Pattern Matching by Rs-Operations: Towards a Unified Approach to Querying Sequenced Data," *Proceedings of the 11th ACM Principles of Database Systems* (1992), 293–300.

[6] Ralf Harmut Güting, Roberto Zicari, and David M. Choy, "An Algebra for Structured Office Documents," *ACM Transactions on Office Information Systems* 7 (1989), 123–157.

[7] R. Karp, R. Miller, and A. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays," *Proc. 4th Annual ACM Symposium on Theory of Computing* (1972), 125–136.

[8] Eric S. Lander, Robert Langridge, and Damien M. Saccocio, "Mapping and Interpreting Biological Information," *Communications of the ACM* 34 (1991), 33–39.

[9] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik, "The AQUA Data Model and Algebra," *Proc. 4th Intl. Workshop on Database Programming Languages* (1993).

[10] Joel Richardson, "Supporting Lists in a Data Model (A Timely Approach)," *Proceedings of the 18th VLDB Conference* (1992).

[11] J. W. Thatcher and J. B. Wright, "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic," *Mathematical Systems Theory* 2 (1968), 57–81.

[12] Scott L. Vandenberg and David J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance," *Proceedings of the SIGMOD Intl. Conference on Management of Data* (1991), 158–167.