Supporting Lists in a Data Model (A Timely Approach)

Joel Richardson
IBM Almaden Research Center

Abstract

This paper considers the problem of adding *list* as a type constructor to an object-oriented data model. In particular, we are concerned with how lists in a database can be constructed and how they can be queried. We propose that operators from a discrete, linear-time temporal logic provide a natural basis for making assertions about the ordering of elements in a list. We then show how such assertions may be incorporated into a query algebra by extending the Melampus Data Model (MDM) with a list type constructor and by allowing temporal assertions as predicates on lists. The extended algebra allows the expression of a significantly larger class of queries than previously possible. Furthermore, temporal operators provide a basis for creating new lists that satisfy desired ordering properties. For example, sorting is shown to fall out as a special case. This paper also describes a new framework based on Boolean circuits for evaluating the truth of an assertion on a given list. This framework provides many opportunities for optimization and parallelism, and it lends insight to the meaning and complexity of a temporal formula.
Keywords: data models, complex objects, temporal logic, query languages.

1 Introduction

Recently, there has been a great deal of interest in defining data models that include, beside traditional sets (or relations), other "bulk" data types. (See, for example, [DBPL 89, DBPL 91].) Perhaps the most common such construct to be added is the ordered sequence. For example, sequences appear as data type constructors in Galileo [Albano 85], FAD [Bancilhon 87], NST [Güting 89], EXTRA/EXCESS [Carey 88, Vandenberg 91], O_2 [Bancilhon 89], and Algres [Ceri 90]. In part, this frequency reflects the intuition that a list is the "next most" complicated data structure after sets (or multisets) and thus a logical choice for researchers to tackle first. Even more, however, it is a reflection of the inherent need in many applications for ordering among data items. In a scientific database, for example, each experimental run may be recorded as a sequence of observations. A project planning database may keep a prioritized list of tasks. In a stock database, each stock's history may be represented as a sequence of price quotations. Even in a traditional business database, ordering is needed, e.g., to sort the output of a query.

The power of data manipulation languages to extract information has not kept pace with the ability of data definition languages to create complex structures. In the case of lists, the ability to extract sublists and the ability to select lists from a set are usually quite constrained. For example, both O_2 [Bancilhon 89] and EXTRA/EXCESS [Carey 88] allow a list to be the target of a select query; each element that satisfies the selection predicate is returned in an (order-preserving) result list. It is not possible, however, to retrieve the *first* element to satisfy the predicate, or all those following the last such element, etc. An O_2 or EXTRA/EXCESS database may also include a set of lists. However, there are few predicates that one may apply in selecting from such a set.

Data models have been similarly constrained in their support of list creation, with support being limited to low-level, physical operations and a few specially defined, high-level operations. For example, the user may be allowed to create a singleton list from a given object or to concatenate two lists. Sorting may be provided as a special-purpose operator.

In this paper, we describe a new approach to incorporating lists into a data model. The approach was inspired by work in the area of temporal logic and is

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

based on a simple observation: temporal formulae (for discrete, linear time) which usually serve to describe time-ordered sequences of "states," can equally well describe spatially-ordered sequences of objects.[1] For example, the temporal modality "always," which can be used to express dynamic integrity constraints such as "an employee's salary never decreases," can also express properties of a (static) sequence, such as "the list of employees is sorted by salary." By using temporal logic as the basis of a list query facility, it is possible to express easily a significantly larger class of interesting queries than in previous work. For example, suppose we have a scientific database in which set S contains the results of a large number of runs from a superconductivity experiment. Each run consists of a list of observations. We may select the runs in which superconductivity was (or was not) displayed, those in which the temperature monotonically decreased, the runs in which resistance dropped to zero before the temperature reached its minimum, etc. Furthermore, given an individual run (a single list), we can extract the observations preceding the first display of superconductivity, those following each decrease in temperature, etc.

Using temporal logic as a basis for querying lists offers many advantages. It has a well-developed formal basis. It is a concise and powerful declarative language, allowing the user to specify ordering properties without specifying how to evaluate them. In addition to being advantageous for the user, a declarative language also presents many opportunities for optimization. Furthermore, since the formalism of temporal logic is derived from intuitive concepts of ordering (e.g., "eventually" or "from now on"), the expression of such queries is (relatively) painless. Of course, temporal logic has limitations as well, and certain potentially interesting queries cannot be expressed. For example, it would not be possible (barring extensions) to select the elements in a list that occupy even-numbered positions [Wolper 83].

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In particular, we focus on other data models that have included lists and on other applications of temporal logic in a database context. After defining a set of temporal operators appropriate for finite lists in Section 3, Section 5 then incorporate *List* as a type constructor into the Melampus Data Model (MDM) [Richardson 91a, Richardson 91b]. First, we address several issues that arise when the formalism of temporal logic is incorporated into a data model. Then we extend the MDM query algebra by adding list as a fundamental type constructor, by adding operators to build lists, and by allowing temporal predicates to appear in queries involving lists. The usefulness of these extensions is demonstrated by examples. Section 6 then presents a novel framework for evaluating the truth of a temporal predicate against a list. This framework, based on the construction of Boolean circuits, reveals the inner regularity of the computation and exposes opportunities for parallelism and other optimizations. Section 7 introduces *rigid variables*, shows how they may be used to express more powerful queries, and shows how the evaluation framework can be extended to handle them. The paper concludes with a discussion of future work.

[1] We do not mean to imply spatial databases or GIS applications, although they, too, can use sequenced objects. Rather, "spatially-ordered" here means ordered within the space of database objects.

2 Related Work

2.1 NST

The NST (Nested Sequences of Tuples) data model was designed to model documents [Güting 89]. In a document, nested sequential order arises naturally, e.g., the order of paragraphs in a section, and sections in an article. One of the goals of NST is to provide algebraic operators that are convenient for manipulating structured documents, e.g., to produce an invoice from orders and inventory. Thus, NST features powerful sequence restructuring operators. However, as in EXTRA/EXCESS and O_2, the facilities for querying sequences are relatively weak. In particular, **exists** α (**forall** α) is true of a list if any (every) element satisfies α. These operators essentially treat a list as an unordered set, e.g., if **forall** α holds for a list, then it holds for any permutation of that list. Similarly, the NST select operator σ applies a predicate to each list element and returns (in the same order) those that qualify. Again, the value of the predicate on a given element is independent of the element's position in the list. The ability to express such "order-dependent" predicates is a central goal of our work. As a result, several operators that are primitive in the NST model (e.g., **ord** for sorting and **rdup** for duplicate elimination) are derived operators in our model.

2.2 Rs-Operations

A more recent proposal notes the lack of facilities in existing data models for manipulating sequences [Ginsburg 92]. The authors propose two powerful operators, called Rs-operations, which are based on regular languages and which define a family of list merging and extracting operations. Each operator takes a regular expression as an argument, and the words generated by the expression serve as patterns that direct how lists should be shuffled together or picked apart. For example, the regular expression

$Q = (x_1x_2)*$ generates x_1x_2, $x_1x_2x_1x_2$, ... In a merge operation, Q can be used to produce the perfect shuffle of two equal-length lists. In an extraction, Q can be used to produce the sublist of elements in odd (or even) numbered positions. These operators add considerable power to the users' ability to manipulate lists, although they seem to do so at a physical level. The connection between the logical ordering of a query result and the expression of that query can be difficult to see.

2.3 Time Sequences

The work in this paper is similar in some respects to Segev and Shoshani's time sequences [Segev 87]. However, their emphasis is on the management of sampled, time-varying data. For example, they provide a taxonomy of different kinds of time sequences, e.g., regular vs. irregular, discrete vs. continuous, etc. They provide operations to support these kinds, e.g., interpolation of data values between recorded time points, and their query language includes extensions specialized for specifying time points and intervals. In our work, we are concerned with a lower level concept, sequential data structures, and we provide no special support for those that are associated with a time sequence. On the other hand, our query langauge does not restrict "temporal" predicates to the time dimension, since the temporal operators are available as general query constructs.

2.4 Temporal Databases

The usefulness of temporal modalities for querying has long been recognized in the context of temporal relational databases [McKenzie 91]. Given that temporal queries range over sequences of database states, it is perhaps not surprising that similar modalities should be useful in querying sequences of objects. Indeed, one of the common uses for a list is to maintain recorded histories, e.g., a sign-up sheet, a sequence of experimental observations, etc. Thus, one might wonder if the problem of querying lists can be addressed simply by mapping those lists to a temporal data model.

There are important reasons, however, why this approach is neither desirable nor sufficient. The problem of querying sequences is at once simpler and more complicated than querying in a temporal database. It is simpler in the sense that a list is linear, whereas time in a temporal data model may be multidimensional, e.g., the model may recognize both transaction time and valid time [Snodgrass 87]. Querying a sequence is more complex in that a database object can be an arbitrary composition of structures. One list might contain simple data values, while another contains lists of sets of tuples. In a temporal database, all sequences have the same structure, which is defined by the temporal data model. For example, Tuzhilin and Clifford define a temporal database as a *temporal structure* which maps each time instant to a database [Tuzhilin 90]. Thus every temporal structure is isomorphic to a sequence of tuples in which the attribute names are relation names and each attribute value is a (flat) relation. Other temporal data models define the mapping from time to structures at the tuple level or at the attribute level [McKenzie 91]. These map to different, but still fixed, data structures.

Another difference between sequences of objects and the implied sequences of a temporal database is the interpretation and usefulness of indexing operations. If the temporal model defines time as continuous, then concepts like "the third time instant" or "the time instant before $\mathcal{P}$ becomes true" are meaningless. Even in discrete time models, such concepts are of dubious utility in a query language (and thus, do not appear), since the granularity of time is likely to be quite small, implementation dependent, and ultimately, of little concern to the user. In querying a list, however, absolute and relative indexing are important operations, e.g., "Who is the second person in the list?" or "Return the three observations preceding the first display of superconductivity." Thus, we may expect the temporal modalities "next" and "previous" to play a relatively more prominent role in querying lists than in querying a temporal database.

3 The Temporal Operators

In this section, we introduce the set of temporal operators that will form the basis for querying lists. These operators are similar to those defined in [Lichtenstein 85]. We include both past and future operators, with the basic modalities being *next*, *previous*, *until*, and *since*. One minor difference is that we deal only with finite pasts and futures, reflecting the fact that lists in a database are finite objects. We also make allowances for lists whose elements may be complex objects (e.g. a list of lists) and for empty lists.

3.1 Syntax

In defining a syntax, we immediately face the problem of deciding what an "atomic" formula is. This is because the elements of a list may themselves be complicated structures, and thus, whether a formula is atomic depends on the type of the list to which it is applied. For the moment, we shall postpone this discussion and simply state that an atomic formula is a predicate that can be evaluated by looking at an individual list element and that we have some way

$$
\begin{array}{lll}
\langle L,i\rangle \models \neg\mathcal{P} & \text{iff} & \langle L,i\rangle \not\models \mathcal{P} \\
\langle L,i\rangle \models \mathcal{P}\vee\mathcal{Q} & \text{iff} & \langle L,i\rangle \models \mathcal{P} \text{ or } \langle L,i\rangle \models \mathcal{Q} \\
\langle L,i\rangle \models \bigcirc\mathcal{P} & \text{iff} & (1\le i<n) \Rightarrow \langle L,i+1\rangle \models \mathcal{P} \\
\langle L,i\rangle \models \mathcal{P}\,\mathcal{U}\,\mathcal{Q} & \text{iff} & \exists j : i\le j\le n \,.\, \langle L,j\rangle \models \mathcal{Q} \text{ and} \\
 & & \forall k : i\le k<j \,.\, \langle L,k\rangle \models \mathcal{P} \\
\langle L,i\rangle \models \ominus\mathcal{P} & \text{iff} & (1<i\le n) \Rightarrow \langle L,i-1\rangle \models \mathcal{P} \\
\langle L,i\rangle \models \mathcal{P}\,\mathcal{S}\,\mathcal{Q} & \text{iff} & \exists j : 1\le j\le i \,.\, \langle L,j\rangle \models \mathcal{Q} \text{ and} \\
 & & \forall k : j<k\le i \,.\, \langle L,k\rangle \models \mathcal{P}
\end{array}
$$

Figure 1: The Semantics of $\models$.

of syntactically identifying such a formula. The syntax, then, is as follows. First, every atomic formula is a formula. Then if $\mathcal{P}$ and $\mathcal{Q}$ are formulas, so are: $\neg\mathcal{P}$, $\mathcal{P}\vee\mathcal{Q}$, $\bigcirc\mathcal{P}$, $\ominus\mathcal{P}$, $\mathcal{P}\,\mathcal{U}\,\mathcal{Q}$, $\mathcal{P}\,\mathcal{S}\,\mathcal{Q}$.

3.2 Semantics

Let L be a sequence of $n>0$ objects, $L[1],\ldots,L[n]$. We distinguish between an object $L[i]$ in a list, and the *point in the list* that it occupies, denoted $\langle L,i\rangle$. The truth of a temporal formula is defined as truth at a point: $\langle L,i\rangle \models \mathcal{P}$ means that $\mathcal{P}$ is satisfied at the ith point of L. Because it is a common case, we will write $L\models\mathcal{P}$ as a shorthand for $\langle L,1\rangle\models\mathcal{P}$. The semantics of $\models$ is given in Figure 1. The first two rules define negation and disjunction; from these, we define conjunction ($\wedge$) and implication ($\Rightarrow$) in the standard way. The next four rules define the semantics of the temporal operators $\bigcirc$(weak next) and $\mathcal{U}$ (strong until) along with their past counterparts $\ominus$ (weak previous) and $\mathcal{S}$ (strong since). $\bigcirc\mathcal{P}$ ($\ominus\mathcal{P}$) holds at a point in a list if $\mathcal{P}$ holds at the next (previous) point. These operators are weak in the sense that there need not be a next (previous) point in order to satisfy the formulas. That is, $\bigcirc\mathcal{P}$ holds at the last point in a sequence for any $\mathcal{P}$, and analogously for $\ominus\mathcal{P}$ at the first point. Note that $\bigcirc\mathit{false}$ holds only for the last element in a sequence, while $\ominus\mathit{false}$ holds only for the first. This will prove very useful for certain kinds of queries. From the weak forms, we can derive operators for strong next ($\bar{\bigcirc}$) and strong previous ($\bar{\ominus}$):

$$\bar{\bigcirc}\mathcal{P} \equiv_{def} \neg(\bigcirc\neg\mathcal{P}) \quad \bar{\ominus}\mathcal{P} \equiv_{def} \neg(\ominus\neg\mathcal{P})$$

The formula $\bar{\bigcirc}\mathcal{P}$ is satisfied at a point in a sequence iff that point is not the last and $\mathcal{P}$ is satisfied at the next point, and similarly for $\bar{\ominus}\mathcal{P}$. Clearly, these operators can be used for counting in a sequence. We shall write $\bigcirc^m\mathcal{P}$ as a shorthand for $\overbrace{\bigcirc\ldots\bigcirc}^{m}\mathcal{P}$ and similarly for the other forms of next and previous. Furthermore, when $m=0$, we shall understand $\bigcirc^m\mathcal{P}$ to be equivalent to $\mathcal{P}$.

The "since" and "until" operators are strong in the sense that the predicate $\mathcal{Q}$ must eventually hold (in the past or future). That is, $\mathcal{P}\,\mathcal{U}\,\mathcal{Q}$ holds at a point if $\mathcal{P}$ holds continuously from that point until a point at which $\mathcal{Q}$ holds. Similarly, $\mathcal{P}\,\mathcal{S}\,\mathcal{Q}$ holds at a point if $\mathcal{Q}$ held at some previous point, and $\mathcal{P}$ has held ever since. In both cases, the predicate holds at a point if $\mathcal{Q}$ holds at that point. From these operators we derive past and future operators denoting "eventually" ($\Diamond$ $\blacklozenge$) and "always" ($\Box$ $\blacksquare$):

$$\Diamond\mathcal{P} \equiv_{def} \mathit{true}\,\mathcal{U}\,\mathcal{P} \quad \blacklozenge\mathcal{P} \equiv_{def} \mathit{true}\,\mathcal{S}\,\mathcal{P}$$
$$\Box\mathcal{P} \equiv_{def} \neg(\Diamond\neg\mathcal{P}) \quad \blacksquare\mathcal{P} \equiv_{def} \neg(\blacklozenge\neg\mathcal{P})$$

$\Diamond\mathcal{P}$ holds at a point i iff $\mathcal{P}$ holds at i or at some point following i, while $\Box\mathcal{P}$ holds iff $\mathcal{P}$ holds at i and at every following point. Similar descriptions apply to the past forms.

4 Data Model Issues

In this section we discuss two problems that arise when we merge the formalism of temporal logic with MDM, an object-oriented data model [Richardson 91b]. MDM features abstract types, conformity-based subtyping, multisets, and a query algebra. MDM is a "pure" object model in that object identity is pervasive and all access to an object is via a method interface; even access to an object's state is defined in terms of *get* and *put* methods [Snyder 86]. We use a functional notation to denote method invocation. For example, if x is an experimental observation, $\mathit{temp}(x)$ returns the temperature in that observation. The syntax for our query operators is also functional and was inspired by the ENCORE algebra [Shaw 91]. For example, to select the elements of a multiset S that satisfy a predicate p, we write $\mathit{Select}(S,p)$.

4.1 Atomic Formulas

In some temporal logics, an atomic formula is simply one containing no temporal operators. This is certainly a sufficient condition for our purposes (though not necessary, as we shall see). Thus, for any predicate $\mathcal{P}$ containing no temporal operators,

$$\langle L,i\rangle \models \mathcal{P} \quad \text{iff} \quad L[i]\models\mathcal{P}$$

That is, an atomic formula holds at a point in a sequence if it holds for the object at that point. Note that if $\mathcal{P}$ is atomic, then $L\models\mathcal{P}$ means that $\mathcal{P}$ holds for the first object in L.

To define what it means for a predicate to hold for an object, we must address the issue of free variables

in a formula. Traditionally, free variables in an atomic formula are successively bound to each state in a sequence, the implicit assumption being that a state has a tuple-like structure. Such variables are often called *flexible* or *dynamic* because they take on different values at each point in the sequence. (This is in contrast to *rigid* variables, discussed in Section 7.) Suppose we are considering a sequence L of observations from a superconductivity experiment. In standard temporal logic, we would write $\Box(temp < 43)$ to assert that the temperature is always below 43 degrees. Here, the atomic subformula is $(temp < 43)$, and the free variable *temp* is bound to the corresponding attribute in each state in the sequence.

This approach to flexible variables is inappropriate for our purposes since the "states" in a sequence are actually objects to which we can only apply methods. Furthermore, many queries need to refer to the identities of the objects in a sequence (e.g., the observations themselves), as well as to the methods in their interface (e.g., *temp*). For these reasons, we assume a single flexible variable, named $\cdot$ (dot), that binds to the identity of each object in a sequence. To express that the temperature always remains below 43 degrees, we would write $\Box(temp(\cdot) < 43)$. To assert that a specific object, say A, is somewhere in a list, we would write $\Diamond(\cdot = A)$.

In our formalism, a predicate containing no temporal operators is certainly atomic. However, this is too restrictive a definition in general, since the elements of a list may themselves be lists to which we would like to apply temporal predicates. For example, given a list of lists of observations, we might wish to assert that in at least one of the lists, the temperature is always below 43 degrees. Even though "always less than 43 degrees" is a temporal predicate, it is atomic with respect to the outer list. Therefore, we introduce a syntax for explicitly forcing the interpretation of a formula to "go down a level:"

$$\langle L, i\rangle \models [\![\mathcal{P}]\!] \quad \text{iff} \quad L[i] \models \mathcal{P}$$

Thus, to express the above predicate, we would write $\Diamond [\![\Box(temp(\cdot) < 43)]\!]$. A list satisfies this predicate only if there is some (list) element that satisfies $\Box(temp(\cdot) < 43)$. Note that without the brackets, this predicate has quite a different interpretation. A list L of lists ℓ of observations satisfies $\Diamond\Box(temp(\cdot) < 43)$ if the temperature is below 43 degrees in the first observation in every ℓ for some suffix of L.

4.2 The Empty List

The operators defined in Section 3 assume that every sequence has at least one element. However, a database may certainly contain empty lists. Although there are not many interesting queries that one can ask about empty lists, we would at least like to be able to identify them in a query. It turns out, however, that the natural way of extending the interpretation of the temporal operators to empty lists leads to some surprising results. For example, $\Box\mathcal{P}$ means $\mathcal{P}$ holds at all points in the list. Thus we might expect that the empty list vacuously satisfies such a formula. But then, the empty list must satisfy $\Box$*false*. If we were to accept this interpretation, then the theorem from temporal logic $\Box\mathcal{P} \Rightarrow \mathcal{P}$ would no longer be valid. Rather than lose important theorems in order to incorporate a fairly uninteresting case, we treat the empty list in an *ad hoc* manner (and assume that an implementation will use *ad hoc* techniques for handling this case). In particular, we define the special predicate symbol *empty*, which is satisfied only by the empty sequence. Furthermore, the empty sequence satisfies *no* formulas of the form $\theta\mathcal{P}$ or $\mathcal{P}\phi\mathcal{Q}$, where θ and ϕ are temporal operators. For example, the formula *empty* $\vee \Box\mathcal{P}$ is satisfied by empty lists and by nonempty lists that satisfy $\Box\mathcal{P}$.

5 Adding Lists to MDM

This section details how lists are incorporated into MDM. In order to concentrate on the use of temporal logic for expressing queries, we will omit the description of some of MDM's operators. In particular, we will not discuss the operators *Aggregate*, *Union*, and *Difference* as they apply to lists. For a description of *Aggregate*, see [Richardson 91b]; MDM's list *Union* and *Difference* operators are essentially the same as those of EXTRA/EXCESS [Vandenberg 91].

5.1 List Membership

Membership in a list, like membership in a multiset, is based on identity, and like a multiset, a list may have more than one occurrence of a given element. If R is a list or multiset, then $Card(x, R)$ denotes the number of occurrences of x in R. If x is not an element, then $Card(x, R) = 0$. Thus, we define the "element of" predicate in terms of *Card*:

$$x \in R \equiv_{def} Card(x, R) > 0$$

5.2 Sublists

The concept of a sublist arises naturally in describing the results of list operations. Because of the inherent ordering of list elements, however, there are at least three reasonable definitions of sublist (in increasing order of constraint): unordered, order-preserving, and contiguous. Let ℓ and L be two lists. We say that ℓ is an *unordered sublist* of L, written $\ell \subseteq L$,

iff $\forall x : Card(x, \ell) \leq Card(x, L)$. Next, ℓ is an *order-preserving sublist* of L, written $\ell \sqsubseteq L$, if $\ell \subseteq L$ and the elements in ℓ appear in the same order relative to their appearance in L. That is, for each $1 \leq i \leq |\ell|$, there is some k_i such that $\ell[i] = L[k_i]$, and $k_{i+1} > k_i$ for $1 \leq i < |\ell|$. Finally, ℓ is a *contiguous sublist* of L, written $\ell \sqsubseteq\!\!\!\sqsubseteq L$, if $\ell \sqsubseteq L$ and for $1 \leq i < |\ell|$, $k_{i+1} = k_i + 1$.

5.3 Constructors

We now introduce primitive operators for creating lists and for converting between lists and multisets. For any object x, $[x]$ constructs a singleton list containing x as its only element. (This is similar to the *ARR* operator of EXTRA/EXCESS [Vandenberg 91].) If the type of x is `T`, then the type of $[x]$ is a list of `T`, denoted `[T]`. Similarly, $\{x\}$ constructs a singleton multiset of type `{T}` containing only x.

For any list L of type `[T]`, the operator $Set(L)$ creates a multiset of type `{T}` containing all of the elements of L, i.e., $\forall x : Card(x, Set(L)) = Card(x, L)$. In effect, $Set(L)$ keeps the elements of L while throwing away their ordering. Note that $Set(L)$ is distinct from $\{L\}$; the latter creates a multiset of type `{[T]}` containing the list L.

Since information is lost when a list is converted to a set, it is not immediately clear how to define an inverse operator that converts a set to a list. Rather than define a set-to-list operator that chooses an arbitrary ordering for the list, we introduce a primitive operator that produces all orderings. For any multiset S of type `{T}`, the operator $List^*(S)$ returns the set of all unique permutations of the elements of S. The result has type `{[T]}`. Clearly, since the size of the result is $O(|S|!)$, $List^*$ could never be used directly in any practical sense. However, like the powerset operator [Gyssens 88], the permutation operator is useful for its ability to define the semantics of other interesting (and more practical) operators. We shall see several examples.

5.4 Select

The *Select* operator applies a predicate to the elements of a multiset and returns the subset of qualifying elements: $Select(S, \mathcal{P}) = \{x | x \in S \wedge x \models \mathcal{P}\}$. Suppose S is a set of lists, and $\mathcal{P}$ is a predicate possibly containing temporal operators. Then the selection returns those lists such that $\mathcal{P}$ is satisfied at the first point. For example, $Select(S, \Box(temp(\cdot) < 0))$ retrieves the lists in which the temperature is always less than 0. By using combinations of next and previous operators, we can test a predicate at any point in a list. For example, to select lists where $\mathcal{P}$ holds at the i-th position, we could write: $Select(S, \bar{\bigcirc}^{i-1}\mathcal{P})$. We may also specify that $\mathcal{P}$ holds i positions from the end: $Select(S, \Diamond(\bar{\circledcirc}^{i-1}\mathcal{P} \wedge \bigcirc false))$.

The following examples show additional selection queries. In all cases, we are selecting from a multiset of lists. The list element type varies but should be clear from the context.

- $Select(S, empty \vee \bigcirc^n false)$. Select lists of length n or less.
- $Select(S, \bar{\bigcirc}^4 \diamondsuit(\cdot = Mary))$. Select lists (of length 5 or more) in which Mary appears somewhere in the first 5 positions.
- $Select(S, \Diamond((\cdot = Mary) \wedge \Diamond(\cdot = John)))$. Select lists in which Mary precedes John.
- $Select(S, (temp(\cdot) \geq 0) \mathcal{U} \Box(temp(\cdot) < 0))$. Select the runs in which the temperature was initially non-negative, then fell below zero for the remainder of the experiment.
- $Select(S, \Box[\![\Diamond(\cdot = Mary)]\!])$. Here, each element in S is a list of lists. The predicate selects the lists in which every element list contains Mary.

We now extend the semantics of the *Select* operator to the case where elements are being chosen from a list. In this case, the elements are retrieved from each point at which the selection predicate holds. Let L be of type `[T]`. Then,

$$Select(L, \mathcal{P}) \equiv_{def} [L[i] \mid \langle L, i \rangle \models \mathcal{P}]$$

where $L[i]$ denotes the element at position i. The result list also has type `[T]` and is order preserving with respect to L. If no points in L satisfy $\mathcal{P}$, then the result is an empty list.

When $\mathcal{P}$ contains no temporal operators, then $Select(L, \mathcal{P})$ is equivalent to list selection in EXTRA/EXCESS and O_2. For example, given an experimental run, L, we can select all the observations for which temperature is less than zero with $Select(L, (temp(\cdot) < 0))$. By adding temporal operators, we can ask more complex queries. For example, $Select(L, \bar{\bigcirc}(temp(\cdot) < 0))$ retrieves each observation that precedes one in which the temperature is below zero, while $Select(L, \diamondsuit(temp(\cdot) < 0))$ retrieves the list's tail beginning with the first observation in which the temperature is below zero.

5.5 Choose1

The operator *Choose1* returns an arbitrarily chosen element from a list or multiset. If the list or multiset is empty, *Choose1* returns *nil*. In [Richardson 91b], we showed how several useful multiset operators could be derived by composing *Choose1* with other primitive operators. Similarly, *Choose1* can derive useful list operators.

5.5.1 Indexed Retrieval Operators

We can compose *Choose1* and *Select* to yield an operator that returns the ith element in a list.

$$L[i] \equiv_{def} \begin{cases} nil & i < 1 \\ Choose1(Select(L, \bar{\ominus}^{i-1} \ominus false)) & i \geq 1 \end{cases}$$

Since the formula $\ominus false$ is satisfied only at the first point in the list, $\bar{\ominus} \ominus false$ holds only at the second, $\bar{\ominus}\bar{\ominus}\ominus false$ at the third, and so forth. Therefore, the selection returns a singleton list containing the ith element of L; *Choose1* then returns this element. Note that if L is empty, or if i is not in the range $1..|L|$, then $L[i]$ returns *nil*. We can similarly derive an operator that extracts contiguous sublists, rather than a single element. One limitation to both operators is that the indices must be "compile-time" constants, since the notation $\bar{\ominus}^m$ is really just a shorthand for an explicit number of $\bar{\ominus}$ operators.

5.5.2 Deriving a Generalized Sort Operator

Suppose S is a multiset and $\mathcal{P}$ is a temporal predicate. We can derive a "temporal sorting" operator whose effect is to arrange the element of S into a list that satisfies $\mathcal{P}$.

$$Sort(S, \mathcal{P}) \equiv_{def} Choose1(Select(List^*(S), \mathcal{P}))$$

The result of $Sort(S, \mathcal{P})$ is a list, L, containing all of the elements of S and such that $\langle L, 1\rangle \models \mathcal{P}$. If no such ordering is possible, the result is *nil*. (*Select* will return an empty set, and *Choose1* of an empty set is defined to be *nil*.) Finding good heuristics for implementing this operator is a major challenge for future research. It would also be interesting to look for general, deterministic algorithms for *Sort*, to see if its apparent $n!$ complexity can be reduced. Given that the problem of determining whether there exists any sequence of length n that satisfies a given $\mathcal{P}$ is NP-complete (from the results of [Sistla 85]), then we know that the complexity of *Sort* is at least NP.

We should point out that we will need to strengthen the predicate language with rigid variables (Section 7) before we can use this operator in the conventional sense. Even so, we can still express useful operations with *Sort*. For example, $Sort(S, true)$ converts the multiset S into an arbitrarily ordered list. The expression $Sort(S, \mathcal{P}\,\mathcal{U}\,\Box\neg\mathcal{P})$ creates a list such that all elements of S satisfying $\mathcal{P}$ precede those that do not; ordering within each group is arbitrary.

The next two operators do not make use of temporal predicates directly. Rather, they benefit indirectly by composition with selection.

5.6 Image

The $Image_n$ operator applies an n-ary function to each element in the cross product of its n arguments, and returns the collected results of each application. The n arguments must either be all multisets or all lists, and the returned object is a multiset or list, respectively. For example, if $n = 1$ and L is a list of observations, $Image_1(L, \lambda x.\ temp(x))$ creates a list of the projected temperature readings. The variable x is bound by λ to each object in L, similar to the way $(\cdot)$ is bound in a temporal formula. However, here we introduce an explicit name because in general, *Image* applies an n-ary function requiring n formal parameters. For example, if f is a binary function, we might write $Image_2(L_1, L_2, \lambda xy.f(x, y))$.

5.7 Partition

The *Partition* operator allows a multiset to be divided into equivalence classes based on a user-defined equivalence relation, expressed as a binary predicate, p. The result is a set of multisets m, such that $p(x, y)$ holds for every pair (x, y) taken from any m and such that $\neg p(x, y)$ holds for every pair taken from different m's. We extend the definition of *Partition* to operate on lists in a straightforward manner. $Partition(L, p)$ returns a list, L', of lists, ℓ, such that each ℓ contains the elements of one partition, each ℓ is order-preserving with respect to L, and the ordering of the ℓ's with L' is according to their first elements (that is, $Image_1(L', \lambda\ell.\ell[1]) \sqsubseteq L$).

As an example, suppose L is a merged transaction log in which each entry is tagged with the id of the associated transaction (TID). We may restructure the log into a list of individual transaction logs:

$$L' = Partition(L, \lambda x\, y.\ \mathrm{TID(x)} = \mathrm{TID(y)})$$

The sequence of individual logs in L' will correspond to the order in which the transactions started. Suppose that log records also contain the action performed (ACT) and the log sequence number (LSN), and suppose we wish to consider only the transactions that committed. We can select the logs in L' in which the last record's action is a commit:

$$L'' = Select(L', \Diamond((\bigcirc false) \wedge (\mathrm{ACT}(\cdot) = \mathrm{COMMIT})))$$

Finally, suppose we wish to rearrange the individual logs to reflect the order in which the transactions committed. We will apply the *Sort* operator to do this, but again, we must wait until Section 7 before being able to express the appropriate predicate.

5.7.1 Duplicate Elimination

One reason *Partition* is interesting is that it allows us to define a generalized duplicate elimination operator for multisets [Richardson 91b]. We can easily define a similar operator for lists:

$$Elim(L,p) \equiv_{def} Image_1(Partition(L,p), \lambda \ell.\, \ell[1])$$

The result is an order preserving sublist of L in which only the first member of each equivalence class (as defined by p) is included. For example, $Elim(L, \lambda x\, y\ .\ x = y)$ removes all but the first occurrence of each individual in L.

6 Boolean Circuits

Clearly, in an implementation of the extended MDM algebra, a central algorithm is that which evaluates a temporal predicate on a list. In this section, we present a framework in which to perform such evaluations. As will become clear, this framework allows different evaluation strategies and thus affords many opportunities for optimizations and parallelism. The main idea is that a temporal predicate and a list together define a Boolean circuit [Borodin 77]. A Boolean circuit is a directed acyclic graph in which nodes with in-degree zero are *inputs*, interior nodes are *gates* labelled with $\wedge$, $\vee$, or $\neg$, and nodes with outdegree zero are *outputs*. Evaluating the circuit is equivalent to evaluating the temporal predicate against the list.

The rules for constructing the Boolean circuit for a given formula are based on the familiar fixpoint theorems of temporal logic, e.g.:

$$\mathcal{P}\,\mathcal{U}\,\mathcal{Q} \equiv \mathcal{Q} \vee (\mathcal{P} \wedge \bar{\bigcirc}(\mathcal{P}\,\mathcal{U}\,\mathcal{Q}))$$

That is, $\mathcal{P}\,\mathcal{U}\,\mathcal{Q}$ holds at a point in a sequence if $\mathcal{Q}$ holds or if $\mathcal{P}$ holds and $\mathcal{P}\,\mathcal{U}\,\mathcal{Q}$ holds at the next point. At the last point in the sequence, $\mathcal{P}\,\mathcal{U}\,\mathcal{Q}$ holds only if $\mathcal{Q}$ holds. Similar statements can be made about the past operator, $\mathcal{S}$, and about the other operators derived from $\mathcal{U}$ and $\mathcal{S}$. The process of constructing a Boolean circuit can be viewed as an iterative expansion of the fixpoint theorems for each element in the given list. Note that "construction" here is used in the mathematical sense; as we shall see, an evaluation algorithm need not always materialize the entire graph.

For each $i, 1 \le i \le n$, let $\Sigma[\langle L,i\rangle, \mathcal{P}]$ denote the Boolean circuit that evaluates the assertion, $\langle L,i\rangle \models \mathcal{P}$. We may construct such a circuit according to the rules given in Figure 2. The rules to construct Boolean circuits for the other temporal operators can be derived. For temporal formula $\mathcal{P}$ and list L of length n, it can be shown by induction that the size of $\Sigma[\langle L,1\rangle, \mathcal{P}]$ is $\le 2n|\mathcal{P}|$. This places a bound on the time and space complexity of the evaluation problem.

$$\Sigma[\langle L,i\rangle, \bigcirc\mathcal{P}] = \begin{cases} \Sigma[\langle L,i+1\rangle, \mathcal{P}] & 1 \le i < n \\ true & i = n \end{cases}$$

$$\Sigma[\langle L,i\rangle, \circledcirc\mathcal{P}] = \begin{cases} \Sigma[\langle L,i-1\rangle, \mathcal{P}] & 1 < i \le n \\ true & i = 1 \end{cases}$$

$$\Sigma[\langle L,i\rangle, \mathcal{P}\,\mathcal{U}\,\mathcal{Q}] =$$
$$\begin{cases} \Sigma[\langle L,i\rangle, \mathcal{Q}] \vee \\ (\Sigma[\langle L,i\rangle, \mathcal{P}] \wedge \Sigma[\langle L,i+1\rangle, \mathcal{P}\,\mathcal{U}\,\mathcal{Q}]) & 1 \le i < n \\ \Sigma[\langle L,n\rangle, \mathcal{Q}] & i = n \end{cases}$$

$$\Sigma[\langle L,i\rangle, \mathcal{P}\,\mathcal{S}\,\mathcal{Q}] =$$
$$\begin{cases} \Sigma[\langle L,i\rangle, \mathcal{Q}] \vee \\ (\Sigma[\langle L,i\rangle, \mathcal{P}] \wedge \Sigma[\langle L,i-1\rangle, \mathcal{P}\,\mathcal{S}\,\mathcal{Q}]) & 1 < i \le n \\ \Sigma[\langle L,n\rangle, \mathcal{Q}] & i = 1 \end{cases}$$

Figure 2: Rules for Constructing Boolean Circuits.

Figure 3 shows an example of the Boolean circuit generated by the formula $\Box(temp(\cdot) < 0)$ for a list of experimental observations in which the sequence of temperature readings is 32, 16, 2, -14, -43. The arcs in the circuit have been labeled with the computed truth value that flows along the arc. The value of the predicate at a point in the list is the value on the arc flowing out of the top-level node corresponding to that point. (The construction given above does not explicitly create output nodes, although it would be easy to do so.) In this example, the formula fails to hold at points 1, 2, and 3 in the list, while it does hold at points 4 and 5. Figure 4 shows a more complicated example. Here the formula asserts that when the temperature first falls below zero, it stays below zero for the remainder of the run. This formula holds at every point. As the figures indicate, the Boolean circuits generated by a formula have a regular, repeating structure. In Figure 4, each repeating unit, called a *cell*, is enclosed in a dashed box.

The examples shown in Figures 3 and 4 contain

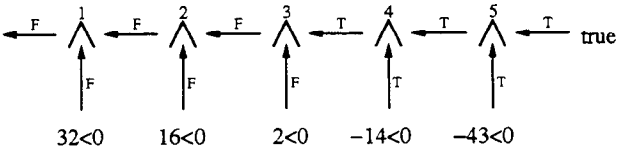

Figure 3: $\Box(temp(\cdot) < 0)$.

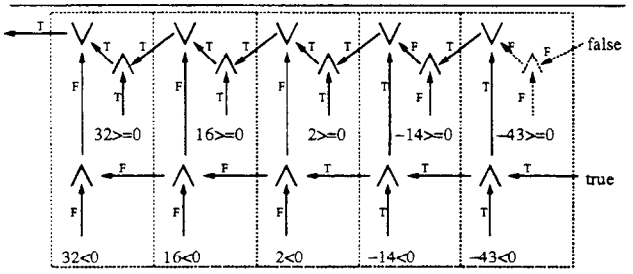

Figure 4: $(temp(\cdot) \geq 0)\,\mathcal{U}\,\Box(temp(\cdot) < 0)$.

only future temporal operators. When this is the case, or when the formula contains only past operators, then we may evaluate $\mathcal{P}$ in a single pass over the list using space proportional to $|\mathcal{P}|$ (rather than $2n|\mathcal{P}|$). Consider the last cell in Figure 4. In an evaluation, Boolean values flow into this cell either from the right (in which case they are known constants) or "up" from atomic predicates applied to the 5th list element. All outputs of the 5th cell can therefore be calculated without reference to any other object in the list. These outputs are then inputs to the 4th cell, and so on. For any cell, the computation requires space proportional to the size of the cell, which is proportional to the size of the original formula. Furthermore, once the outputs of the ith cell have been computed, the space for its inputs can be reclaimed. Therefore, a formula containing only future operators can be evaluated in a single (backward) pass over the list using space proportional to the size of the formula. Similarly, a formula containing only past operators can be evaluated in a single forward pass. In both cases, the time complexity is still proportional to $n|\mathcal{P}|$. It seems likely, though, that in many instances, the cost to access the database for each list element will far outweigh the cost to compute $|\mathcal{P}|$ Boolean operations for that object.

When a formula nests operators of opposing tense, multiple passes are required, with the results from each pass being stored for use in the next. For example, consider the formula $\Diamond(\boxdot\mathcal{P} \wedge \Box\mathcal{Q})$, which asserts that there is some point i such that $\mathcal{P}$ holds at every point from 1 through i and that $\mathcal{Q}$ holds from points i through n. This formula requires 2 passes: a forward pass evaluates $\boxdot\mathcal{P}$ and records its value at each point; a backward pass then evaluates the remainder of the formula, using the cached results from the first pass. Note that a cache can require up to n bits, although in this example, there is a very compact representation. This is because a formula of the form $\boxdot\mathcal{P}$ divides any list into a prefix over which $\mathcal{P}$ holds everywhere and a suffix over which it holds nowhere. Thus, a single integer giving the length of the prefix suffices to cache the results of the first pass. (Similar remarks can be made concerning the operators $\Box$, $\Diamond$, and $\diamondsuit$.) A second point to note is that we may trade space for time in case accessing the database is expensive. In this example, we may elect to evaluate $\mathcal{Q}$ at each point during the first pass (along with $\boxdot\mathcal{P}$), caching the results in a bit vector. Then the second pass operates solely on in-memory data structures.

6.1 Optimizations

The previous section described a framework for evaluating the truth of a formula against a list, and showed that some cases can be evaluated more efficiently than others. Indeed, one advantage of using a declarative language for posing queries is the ability to optimize query processing, and a main focus of current research is the development of such strategies. Here we outline some of the obvious kinds of optimizations that can be expected to improve performance.

- Simplification. We can reduce expression complexity by equivalence- preserving transformations, e.g. $(\Box\mathcal{P}) \wedge (\Box\mathcal{Q}) \equiv \Box(\mathcal{P} \wedge \mathcal{Q})$. Collections of such transformations can be developed and verified formally.

- Early termination. During an evaluation, pieces of the computation (and hopefully, the whole computation) can be "turned off." For example, if a formula contains $\Box\mathcal{P}$ as a subexpression, and if $\mathcal{P}$ evaluates to *false* at some point, then we know that $\Box\mathcal{P}$ must be *false* for every preceding point.

- Heuristically chosen evaluation order. Even with early termination, a backward (or forward) scan is not necessarily the best plan. For example, if we are evaluating $\Box\mathcal{P}$, and $\mathcal{P}$ is false in the first object (as in Figure 3), then a forward scan would discover this right away. Since we are free to materialize and evaluate the Boolean circuit in any order, then we can apply any available knowledge about the data being queried in choosing an evaluation order.

- Parallelism. Boolean circuits were originally studied in connection with parallel complexity theory, so it is no surprise that they should be amenable to parallel evaluation. The highly regular structure of the Boolean circuits induced by temporal predicates allows us to divide the work of evaluation among many processes. For example, each process could be given some "chunk" of the list (some number of cells), and at least the atomic predicates can be evaluated in parallel. The evaluation of temporal predicates then requires communication between the processes. Although this

phase seems to be inherently sequential in nature, various techniques (such as global early termination) can be considered to reduce the severity of the bottleneck.

7 Rigid Variables

So far, we have only been able to express predicates in which the state of each individual object is compared with itself or with global constants. We have not been able to compare the states of different objects in the list. Rigid variables give us this added power. Using rigid variables, we can, for example, specify that the temperature in every observation is within some delta of the temperature in the first observation, or that the temperature in each observation is less than or equal to the one before it. Rigid variables add to both the expressive power of the logic and (unfortunately) to the complexity of the evaluation procedure.

If $\mathcal{P}$ is a temporal predicate, then so is $\Lambda K.\mathcal{P}$. Here, the binding operator Λ introduces the rigid variable K whose scope is $\mathcal{P}$. By convention, we use upper-case letters to denote rigid variables.

A formula containing a rigid variable has the following interpretation:

$$\langle L, i\rangle \models \Lambda K.\mathcal{P} \text{ iff } \langle L, i\rangle \models \mathcal{P}^K_{L[i]}$$

That is, a formula $\Lambda K.\mathcal{P}$ holds at a point i in a sequence if the formula $\mathcal{P}'$ also holds at point i, where $\mathcal{P}'$ is obtained by replacing all free occurrences of K in $\mathcal{P}$ with the object at i. For example, the formula $\Lambda X.\ \bigcirc\,\Box(\cdot \neq X)$ holds at a point i in list L iff $\langle L, i\rangle \models \bigcirc\,\Box(\cdot \neq L[i])$, i.e., iff $L[i]$ is not repeated in the suffix. Therefore, if $L \models \Box(\Lambda X.\ \bigcirc\Box(\cdot \neq X))$, then it contains no duplicates.

Suppose S is a set of experimental runs. To select runs in which the temperature was held constant, we would write: $Select(S, \Lambda X.\ \Box(temp(\cdot) = temp(X)))$. For each list in S, X is bound to the first observation, so the selection criterion becomes $\Box(temp(\cdot) = temp(L[1]))$. Now suppose L is a specific run. The following query selects those observations in which the temperature is a local minimum, i.e., in which the temperature is less then either of its neighbors: $Select(L, \Lambda X.\ \bar{\bigcirc}(temp(\cdot) \geq temp(X)) \wedge \bar{\circledcirc}(temp(\cdot) \geq temp(X)))$. We could select the global minima by replacing $\bar{\bigcirc}$ and $\bar{\circledcirc}$ with $\Box$ and $\boxdot$, respectively. Finally, we can use rigid variables in conjunction with the *Sort* operator to produce sorted lists (in the conventional sense). Suppose S is a set of people. Then $Sort(S, \Box(\Lambda X.(\bigcirc(age(\cdot) \geq age(X)))))$ creates a list of people sorted by increasing age.

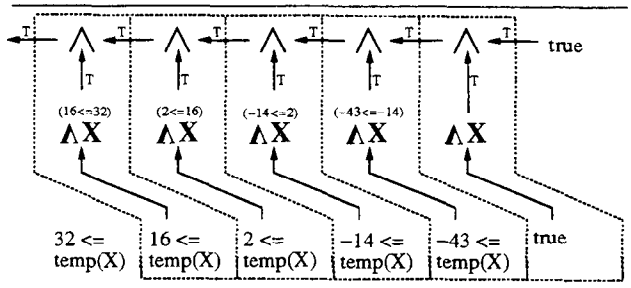

Figure 5: $\Box(\Lambda X.\ \circ(temp(\cdot) \leq temp(X)))$.

7.1 Decision Procedure

This section outlines how the Boolean circuit framework can be extended to handle rigid variables. In Section 6, the decision procedure pushes Boolean constants through a circuit. When an atomic formula contains rigid variables, its truth value at a point is no longer a constant, but is itself an expression. For example, the atomic formula $(temp(\cdot) > temp(X))$ might become $(43 > temp(X))$ at a specific point. This expression can be viewed as a finite representation of the infinite set of temperature values that make the formula true, i.e., the formula holds for all values of K such that $K < 43$. The decision procedure therefore involves building up expressions and pushing those through the circuit. (Strictly speaking, the structure is no longer a Boolean circuit.) Of course, all rigid variables eventually get bound, so the expressions eventually reduce to Boolean constants. When the expression $(43 > temp(X))$ reaches a node in the circuit at which X is bound (by the Λ operator), the truth value can be determined.

To make these ideas concrete, let us consider the predicate which specifies that temperature is monotonically decreasing: $\Box(\Lambda X.\ \bigcirc(temp(\cdot) \leq temp(X)))$. The circuit for this example is shown in Figure 5. First, notice that the presence of the $\bigcirc$ operator in the formula causes the cells to be skewed. Next, the circuit contains a new kind of node corresponding to the Λ expression in the formula. As indicated above, what flows through the circuit (below the Λ nodes) are expressions, rather than Boolean values. For example, in Figure 5, the expression $(16 \leq temp(X))$ flows into the Λ node in the first cell. At a Λ node, the object associated with the current cell is substituted for free occurrences of the rigid variable. In the figure, the result of the substitution is shown in parentheses above each node. Since there is only one rigid variable in this example, the expression reduces to a Boolean constant flowing out of the node.

In this example, the only temporal operator in the scope of the rigid variable X is $\bigcirc$. As a result, the size

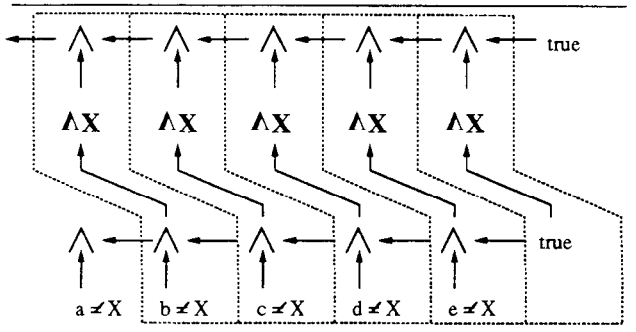

Figure 6: $\Box(\Lambda X.\ \circ\Box(\lambda x.\ x \neq X))$.

of the expression flowing into the Λ nodes is bounded by the size of the cell. If the scope of a rigid variable includes other temporal operators, this is no longer the case. Consider the expression $\Box(\Lambda X.\ \bigcirc\Box(\cdot \neq X))$ which asserts that the list contains no duplicates. This example is depicted in Figure 6. The expression that flows into the Λ node in the fourth cell is $e \neq X$. (The identities of the list elements are denoted by the lower case letters a-e.) But the expression flowing into the third cell is $d \neq X \wedge e \neq X$, and into the second cell is $c \neq X \wedge d \neq X \wedge e \neq X$, etc. In other words, the size of the expression flowing into a node is linear in the length of the list's tail beginning at that point. This is not surprising, since verifying that an element is unique requires global knowledge of the list's other members.

In Figure 6, the expression flowing into each Λ node is identical to the portion of the circuit below and to the right of that node, thus actually constructing the expressions is somewhat redundant. In fact, for this example, we can perform the evaluation by accumulating sets of individuals and then testing for membership. For example, instead of evaluating the expression $d \neq X \wedge e \neq X$ in cell 3, we could evaluate $X \notin \{d, e\}$. We could then add c to the set and pass it to the left. Explicitly maintaining the set of values for X that satisfy (or fail to satisfy) a formula is possible as long as the set is finite. Suppose, however, that the leaf expressions were of the form $x < X$. For any value of x there are infinitely many values of X that satisfy the predicate. In this case, building expressions is attractive, since it allows us to represent infinite sets in a finite way. Furthermore, in some cases, we can avoid the linear growth in the expressions. For example, rather than building the expression $(3 < X) \wedge (-18 < X)$, we can take advantage of the semantics of $<$ and simply pass $3 < X$.

8 Conclusions and Future Work

This paper has described an approach based on temporal logic for incorporating lists into a data model. There are several important contributions of this work. The main contribution is the recognition that temporal logic can be used as the basis of a powerful query language over sequential data structures, and the demonstration of these ideas in the context of MDM. In the process of adding lists to MDM, we have addressed issues that do not normally arise in temporal logic, such as object-identity and the complex structure of list elements. In addition, we have proposed a new framework based on Boolean circuits for evaluating a predicate against a list. This framework affords wide latitude in choosing an evaluation strategy. We have proposed a new approach to defining rigid variables and have described some initial ideas on how the evaluation framework may be extended to handle them. Finally, we have (in a small way) contributed to the area of temporal logic by demonstrating its usefulness in yet another area of computer science.

It is clear that there are many avenues for further research. The possibilities span the range from increasing the expressiveness of the predicate language, to developing a repertoire of evaluation strategies, to building an actual implementation. The focus of current research is on developing evaluation strategies within the Boolean circuit framework, particularly in the presence of rigid variables.

Finally, an intriguing problem is whether reasonably efficient implementations of the *Sort* operator can be found. While it seems unlikely that any general algorithm will be able to perform "normal" sorting in $n \log n$ steps (and therefore, an implementation will probably have to recognize this as a special case), it is nevertheless an interesting challenge to find a general *Sort* algorithm that avoids the $n!$ complexity implied by its definition.

Acknowledgements

Jennifer Widom first introduced me to temporal logic, and she and Moshe Vardi have provided several enlightening conversations. I would like to thank Bobbie Cochrane, Bill Cody, Laura Haas, Peter Schwarz, Moshe Vardi, Jennifer Widom, and the VLDB referees for helping to improve the quality of this paper.

References

[Albano 85] Albano, A., Cardelli, L., and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Transactions on Database Systems*, 10(2), June 1985.

[Bancilhon 87] Bancilhon, F., Briggs, T., Koshafian, S., and Valduriez, P., "FAD, a Powerful and Simple Database Language," *Proc. 13th VLDB Conf.*, Brighton, England, Aug. 1987.

[Bancilhon 89] Bancilhon, F., Cluet, S., and Delobel, C., "A Query Language for the O_2 Object-Oriented Database System," *Proc. 2nd Int'l Workshop on Database Programming Languages,* Gleneden Beach, OR, June 1989.

[Borodin 77] Borodin, A., "On Relating Time and Space to Size and Depth," *SIAM J. Comp.*, 6(4), December 1977.

[Carey 88] Carey, Michael J., DeWitt, David J., and Vandenberg, Scott L., "A Data Model and Query Language for EXODUS," *Proc. ACM SIGMOD Conference,* Chicago, IL, June, 1988.

[Ceri 90] Ceri, S., Crespi-Reghizzi, S., Zicari, R., Lamperti, G., and Lavazza, L., "Algres: An Advanced Database System for Complex Applications," *IEEE Software,* 7(4), July 1990.

[DBPL 89] *2nd International Workshop on Database Programming Languages*, Gleneden Beach, OR, June 1989.

[DBPL 91] *3rd International Workshop on Database Programming Languages*, Nafplion, Greece, August 1991.

[Fegaras 89] Fegaras, L., Sheard, T., and Stemple, D., "The ADABPTL Type System," *2nd Int'l Workshop on Database Programming Languages*, Gleneden Beach, OR, June 1989.

[Ginsburg 92] Ginsburg, S., and Wang, X., "Pattern Matching by Rs-operations: Towards a Unified Approach to Querying Sequenced Data," to appear, *ACM Symposium on Principles of Database Systems,* San Diego, CA, May 1992.

[Güting 89] Güting, R.H., Zicari, R., and Choy, D.M. "An Algebra for Structured Office Documents," *ACM Trans. on Office Information Systems,* 7(4), April 1989.

[Gyssens 88] Gyssens, M., and van Gucht, D., "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. ACM SIGMOD Conference,* Chicago, IL, June 1988.

[Koshafian 86] Koshafian, S., and Copeland, G., "Object Identity," *Proc. ACM OOPSLA Conference,* Portland, OR, September 1986.

[Lichtenstein 85] Lichtenstein, O., Pneuli, A., and Zuck, L., "The Glory of the Past," in *Lecture Notes in Computer Science,* v.193, Springer-Verlag, 1985.

[McKenzie 91] McKenzie, L.E., Jr., and Snodgrass, R., "Evaluation of Relational Algebras Incorporating the Time Dimension in Databases," *ACM Computing Surveys,*, 23(4), December 1991.

[Richardson 91a] Richardson, J., and Schwarz, P., "Aspects: Extending Objects to Support Multiple, Independent Roles," *Proc. ACM SIGMOD Conference,* Denver, CO, May 1991.

[Richardson 91b] Richardson, J., and Schwarz, P., "MDM: An Object-Oriented Data Model," *Third Int'l Workshop on Database Programming Languages,* Nafplion, Greece, Aug. 1991.

[Segev 87] Segev, A., and Shoshani, A., "Logical Modelling of Temporal Data," *Proc. ACM SIGMOD Conference,* San Francisco, CA, May 1987.

[Shaw 91] Shaw, G.M., and Zdonik, S.B., "A Query Algebra for Object-Oriented Databases," *Sixth Int'l. Conf. on Data Engineering,* Los Angeles, CA, February 1991.

[Sistla 85] Sistla, A.P., and Clarke, E.M., "The Complexity of Propositional Linear Temporal Logics," *Journal of the ACM,* 32(3), July 1985.

[Snodgrass 87] Snodgrass, R., "The Temporal Query Lan-
guage TQUEL," *ACM Transactions on Database Systems,* 12(2), June 1987.

[Snyder 86] Snyder, Alan, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. ACM OOPSLA Conference,* Portland, OR, September 1986.

[Tuzhilin 90] Tuzhilin, A., and Clifford, J., "A Temporal Relational Algebra as a Basis for Temporal Relational Completeness," *Proc. 16th VLDB Conference,* Brisbane, Australia, August 1990.

[Vandenberg 91] Vandenberg, S., and DeWitt, D., "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance," *Proc. ACM SIGMOD Conference,* Denver, CO, May 1991.

[Vandenberg 92] Vandenberg, Scott, "A Query Optimizer for Complex Objects," Ph.D. Dissertation, University of Wisconsin-Madison, 1992.

[Wolper 83] Wolper, P., "Temporal Logic Can Be More Expressive," *Information and Control,* vol. 56, 1983.