Core Range Algebra

Toward a formal model of markup.

Gavin Thomas Nicol
Chief Technology Officer
Red Bridge Interactive, Inc.
email:gtn@rbii.com

March 31, 2002

Abstract

There have been a number of formal models proposed for XML. Almost all of these models focus on treating XML as semistructured data, typically as edge-labeled graphs, or as a tree of nodes. While this approach is fine for more traditional database applications, or situations where structure is of paramount importance, it fails to deal with the issues found in the use of XML as text. In particular, unless special functions are introduced, queries involving text that spans node boundaries are not closed over a set of nodes, and likewise, the returned sets of nodes are not necessarily well-formed XML documents. This paper presents an algebra for data that, when applied to XML, is not only closed over the entire set of operations permissible in more traditional XML query languages, but over the operations that involve text and XML fragments that are not in themselves well-formed.

1. Introduction

There are many situations where a sequence[1] of data is given, and some structure is to be inferred such that the data might be further manipulated. Examples of this include traditional flat file data formats, DNA Sequences, and plain textual data. As an example, take the following piece of text.

```
This is a piece of italic text.
```

This is a sequence of characters, but there is some internal structure in the italicized portion that could possibly be manipulated. In languages such as XML, the structure of the data is made explicit via *embedded markup*, as shown below.

```
<doc>This is a piece of <i>italic</i> text.</doc>
```

This is a convenient way to make the implicit structure of the data explicit, and hence simplifies many forms of processing. Such markup typically needs to be nested with no overlap, though it is acknowledged that real text is often not so conveniently structured, as evidenced by papers dealing[durandMarkup][sperbergDesign1] with the issues of using SGML or XML to markup texts in the humanities.

[1]Here "sequence" is not limited to textual data, but is closer to the mathematical sense.

Given the typical nesting structure, the markup lends itself to interpretation as a tree, so it is hardly surprising to see the tools and API's supporting embedded markup to model it as such[w3cdom][w3cxsl], though there are some exceptions[adler]. One of the questions raised by using embedded markup is whether the markup is part of the text, or whether it is simply metadata about the text. In other words, is the tree the *real* document, or is the sequence of characters all there is? For the XML-savvy, the answer is generally that the tree is the canonical form because it represents the inherent structure of the document. An alternate, and possibly older view of documents, called *parallel markup*[nelsonLit][nelsonEmbed], asserts that the markup is not part of the text, but rather an interpretation imposed over a *range* of characters.
This paper defines an algebra called Core Range Algebra that provides a formalism for the use of ranges as a means of dealing with the structure in sequences of data. The motivation behind Core Range Algebra is to provide a basis for a complete formalism bridging the worlds of structured data, such as XML, and semistructured data such as memos and email messages.

2. Core Range Algebra

This paper introduces Core Range Algebra; an algebra of ranges. Core Range Algebra alone is not powerful enough to describe the markup structures possible with XML, or indeed of any nested embedded markup structure. That said, it is capable of modeling overlapping ranges of data, which XML and most typical markup languages disallow.
Core Range Algebra is defined as an algebra over a *sequence*(*S*) of *items*(*I*) from a finite set forming an *alphabet* (*A*) and is closed over all operations on *A*. At the heart of CRA are the abstract data types SEQUENCE, and RANGE described below. In the descriptions, a period is used to denote access to an attribute of a given object, so `foo.bar` means "the `bar` attribute of `foo`".

Sequence Data Type

The SEQUENCE data type is one of the fundamental abstract data types in computer science: it is a completely ordered, finite set of items. We use the following definitions.

Definitions

item
A member of the alphabet(*A*), or $I \in A$. There is no restriction on the format or structure of an item.

alphabet
A finite set of items(*I*).

sequence
An ordered list of items(*I*) from an alphabet(*A*) such that $I \subseteq A$.

length
The number of ITEMS held by the SEQUENCE.

position
An index into the SEQUENCE which identifies an item. The position will be an integer in 0..*n*-1 where *n* is the length of the sequence.

Syntax

Empty: SEQUENCE

IsEmpty: SEQUENCE → BOOLEAN
Contains: SEQUENCE ITEM → BOOLEAN
Equals: SEQUENCE SEQUENCE → BOOLEAN

Diff: SEQUENCE SEQUENCE → SEQUENCE
Union: SEQUENCE SEQUENCE → SEQUENCE
Intersect: SEQUENCE SEQUENCE → SEQUENCE
Concat: SEQUENCE SEQUENCE → SEQUENCE

Insert : SEQUENCE POSITION ITEM → SEQUENCE

Delete : SEQUENCE POSITION → SEQUENCE
ItemAt : SEQUENCE POSITION → ITEM or *null*
Last : SEQUENCE → ITEM or null
First : SEQUENCE → ITEM or null
Length : SEQUENCE → INTEGER

Predicates

IsEmpty(*S*)
Tests whether S contains any items, or *S.length > 0*.

Contains(*S*,*I*)
Tests whether *S* contains an item equal to *I*.

Equals(*S*,*T*)
Tests whether *S* and *T* are each of the same length, and that each item at a given position in *S* is equal to the item at the same position in the *T*.

Operators

Diff(*S*,*T*)
Returns a SEQUENCE containing the items in *S* that are not contained in *T*.

Union(*S*,*S*)
Returns a SEQUENCE containing the union of *S* and *T*.

Intersect(*S*,*T*)
Returns a new SEQUENCE holding the intersection of *S* and *T*.

Concat(*S*,*T*)
Concatenate *S* and *T* and return a new SEQUENCE comprised on the members of *S* and *T* in order (*S1…Sn, T1…Tn*).

Operations

Insert(*S*,*P*,*I*)
Insert *I* into *S* at the position *P* and return a new SEQUENCE. If *P* is out of bounds no change occurs (this could also be raised as an error condition). This operation could be defined in terms of insertion (copying) into an empty SEQUENCE.

Delete(*S*,*P*)
Delete the ITEM at position *P* from the SEQUENCE *S* thereby creating a new sequence. If *P* is out of bounds no change occurs (this could also be raised as an error condition). Note that this could also be defined in terms of insertion (copying) into an empty SEQUENCE.

ItemAt(*S*,*P*)
Return the item at position P within the sequence S, or null if P is out of bounds (this could also be raised as an error condition).

Last(*S*)
Return the last ITEM from *S*, or if *S* is empty, *null*.

First(*S*)
Return the first ITEM from *S*, or if *S* is empty, *null*.

Length(*S*)
Return the length of *S*, an integer greater or equal to 0..

Discussion

Perhaps the most important operator over SEQUENCES is the concatenation operator. Given 2 SEQUENCES, a third SEQUENCE is created by joining the 2 SEQUENCES together. This is a monadic operator given that the empty SEQUENCE represents the unity item, and it also provides us with the means to derive the Kleene closure of a subset, which is critical to regular expressions.

Range Data Type

The range data type is defined in terms of a SEQUENCE *S*. A range is essentially a marker for a sub-SEQUENCE of the sequence *S*, and consists of a POSITION within *S,* and the length of the sub-SEQUENCE. We use the following definitions:

Definitions

range
A {*start*, *length*} pair, where *start* is a position and length is the number of items included in the range.These can be accessed using the dot operator. For example, if we have a range **`R`**, we would write **`R.start`** or **`R.length.`**

Syntax

StartsBefore: SEQUENCE SEQUENCE → BOOLEAN
StartsAfter: SEQUENCE SEQUENCE → BOOLEAN
StartsWithin: SEQUENCE SEQUENCE → BOOLEAN
EndsBefore: SEQUENCE SEQUENCE → BOOLEAN
EndsAfter: SEQUENCE SEQUENCE → BOOLEAN
EndsWithin: SEQUENCE SEQUENCE → BOOLEAN
Within: SEQUENCE SEQUENCE → BOOLEAN
Same: SEQUENCE SEQUENCE → BOOLEAN

Create[2]: POSITION LENGTH → RANGE
Flatten: RANGE SEQUENCE → SEQUENCE
EndOf: RANGE → POSITION
StartOf: RANGE → POSITION
LengthOf: RANGE → INTEGER

Move : RANGE POSITION → RANGE
Resize: RANGE *length* → RANGE

Predicates

The following predicates are defined for RANGES, where each takes 2 RANGES *a* and *b*. Note that all predicates are defined in terms of relationships between the *start* and *length* of RANGES, hence the operations should be very efficient.

StartsBefore(*a*, *b*)
Tests whether *a* starts before *b*, or $a.start ? b.start \quad true$.

StartsAfter(*a*, *b*)
Tests whether *a* starts after *b*, or $a.start ? b.start \quad true$.

StartsWithin(*a*, *b*)
Tests whether *a* starts within *b*, or $a.start \geq b.start \quad a.start \leq EndOf ? b? \quad true$.

EndsBefore(*a*, *b*)
Tests whether *a* ends before *b*, or $EndOf ? a? ? b.start \quad true$.

EndsAfter(*a*, *b*)
Tests whether *a* ends after *b*, or $EndOf ? a? ? EndOf ? b? \quad true$.

EndsWithin(*a*, *b*)
Tests whether *a* ends within *b*, or $EndOf ? a? \geq b.start \quad EndOf ? a? \leq b.end \quad true$.

Within(*a*, *b*)
Tests whether *a* contains *b*, or $a.start \leq b.start \quad EndOf ? a? \geq EndOf ? b? \quad true$. Note that this is equivalent to saying $StartsWithin ? a , b? \quad EndsWithin ? a , b? \quad true$ and can be derived from that.

Same(*a*, *b*)
Tests whether the *start* and *length* of *a* are the same as the *start* and *length* of *b*, or
$a.start = b.start \quad a.length = b.length \quad true$.

[2]We will expand upon the notion of range creation later in this paper.

Operations

Create(*P*,*L*)
Given a POSITION *P* and a length *L*, create a new RANGE. Here both *P* and *L* are positive integers.

Flatten(*R*,*S*)
Given a SEQUENCE *S*, and a RANGE *R*, create a new SEQUENCE holding the sub-SEQUENCE of *S* identified by the POSITION and length of *R*. This is essentially the same as copying all the ITEMS in the sub-SEQUENCE into a newly created SEQUENCE using *Insert*.

EndOf(*R*)
Calculate the last POSITION identified by the RANGE *R*, or $R.start ? R.length$.

StartOf(*R*)
Return the start of the RANGE *R*, or $R.start$.

LengthOf(*R*)
Return the length of the RANGE *R*, or $R.length$.

Move(*R*, *n*)
Move the range R forward of backward by *n*, or $R.start = R.start ? n \, where \, R.start ? n \geq 0$.

Resize(R, *n)*
Make the range *R* larger or smaller by n items, or
$R.length = R.length ? n \, where \, R.length ? n \geq 0$.

Sequences of Ranges

In addition to the fundamental data types SEQUENCE and RANGE, Core Range Algebra makes use of SEQUENCES of RANGES. Functions defined in terms of sequences of ranges can be used to infer structure based on the patterns of containment and other such relationships between ranges.

Syntax

StartOrder: SEQUENCE → SEQUENCE
EndOrder: SEQUENCE → SEQUENCE
Unique: SEQUENCE → SEQUENCE
Ancestor: RANGE SEQUENCE → SEQUENCE
Descendant: RANGE SEQUENCE → SEQUENCE
Parent: RANGE SEQUENCE → RANGE
Child: RANGE SEQUENCE → RANGE

Operations

StartOrder(*S*)
Given a sequence of ranges *S*, return a new sequence such that the ranges are ordered in increasing order based on the *start* of the ranges. If more than one range has the same *start*, these ranges are sorted by *length*.

$$S = Empty \quad Empty$$
$$S \neq Empty \quad S \; where \quad r_i \quad S \, r_i.start ? r_{i ? 1}.start$$

EndOrder(*S*)
Given a sequence of ranges *S*, create a new sequence such that the ranges are ordered in increasing order based on the *EndOf* function on the ranges. If more than one range has the same *EndOf* value, these ranges are sorted in increasing order of *length*.

$$S = Empty \quad Empty$$
$$S \neq Empty \quad S \; where \quad r_i \quad S \, EndOf ? r_i ? ? EndOf ? r_{i ? 1} ?$$

Unique(*S*)
Given a sequence of ranges *S*, create a new sequence such that no two ranges have the same *start* and *length*.

Ancestor(*R, S*)

Given a sequence of ranges *S*, and a range *R*, return all ranges in *S* that are ancestors of *R*, or in other words, all ranges that completely contain R.

$$S = Empty \quad Empty$$
$$r \quad S\,Within?\ r, R?\ ? \quad r \quad S\,R.start?\ r.start \quad EndOf?\ R?\ ?\ EndOf?\ r?$$
$$S \quad Empty$$

Descendant(*R, S*)

Given a sequence of ranges *S*, and a range *R*, return all ranges in *S* that are descendants of *R*, or in other words, all ranges that are completely contained by *R*.

$$S = Empty \quad Empty$$
$$r \quad S\,R.start?\ r.start \quad EndOf?\ r?\ ?\ EndOf?\ r?$$
$$S \quad Empty$$

Parent(*R, S*)

Given a sequence of ranges *S*, and a range *R*, determine which, if any, ranges in *S* directly contain *R* such that their *start* is closest to *R.start*.

$$S = Empty \quad Empty$$
$$S \neq Empty?\ Last?\ StartOrder?\ Ancestor?\ R, S?\ ?\ ?$$
$$S \quad Empty$$

Child(*R, S*)

Given a sequence of ranges *S*, and a range *R*, determine which, if any, ranges in *S* are contained by *R*, and by no other range whose *start* is greater than the *start* of *R*. In other words, determine which, if any, ranges are contained by *R*, and whose parent is *R*.

$$S = Empty \quad Empty$$
$$S \neq Empty \quad r \quad S\,Contains?\ r, R? \quad Equals?\ S, Parent?\ r, S?\ ?$$
$$?\ \quad S \quad Empty$$

Note that the list of children can be ordered by applying order to the result of this function.

Expressions

The following grammar defines the Expression language or Core Range Algebra; a very typical functional language. Expressions are evaluated in a dynamic environment with lexical scoping.

name	*Name*		
function name	*Function*		
variable	*Var*		
constant	*Const*		
comparison operator	Op_{eq}	::= **=** | **!=** | **<** | **>** | **<=** | **>=**	
arithmetic operators	Op_{bin}	::= **+** | **-** | ***** | **/** | **%**	
boolean operators	Op_{bool}	::= **&&** | **||**	
binary operators	*BinaryOp*	::= Op_{eq} | Op_{bin} | Op_{bool}	
unary operators	*UnaryOp*	::= **+** | **-** | **not**	
expression	*Expr*	::= *Const*	constant expression
		| *Name*	name expression
		| *Expr BinaryOp Exp*	binary expression
		| *UnaryOp Exp*	unary expression
		| **if** *Exp* **then** *Exp* **else** *Exp*	conditional expression
		| **let** *Var* **=** *Exp* **do** *Exp*	local binding
		| *Function*(*Expr*; ... *Expr*)	function application
		| **for** *Var* **in** *Expr* **do** *Expr*	iteration
		| **error**	error condition

Core Range Algebra Expression Syntax

In addition to the core expression languages, the functions defined above for the SEQUENCE and RANGE data types can be used in expressions, as can the dot notion.

3. Conclusions

Core Range Algebra, as defined in the paper, is a simple algebra that operates over SEQUENCES, RANGES, and SEQUENCES of RANGES. The intent of Core Range Algebra is to provide a basis for manipulating sequences of data, especially text, as a sequence of items, and to provide means for layering higher-level interpretations over that substrate such that operations at the higher level can be defined in terms of the underlying model.

Future Work

What is woefully missing in the existing specification is a means for constructing sequences of ranges from an underlying sequence. Given the Kleene closure, we have a formal basis for creating ranges based on regular expressions. This will not only provide a powerful means for creating ranges over unstructured texts, but also gives a means for inferring structurally recursive data structures, or to infer higher level structures over previously inferred structures.
One obvious further extension is to expand the Core Range Algebra to fully cover XML. An approach much like REX or Regular Fragmentation can be used to build the structures of XML, from the underlying sequences, and with additional functions for range construction such that elements, attributes etc. can be constructed, a full query language can be defined over the Core Range Algebra defined above.

Related Work

There is a fairly large body of work available to support aspects of Core Range Algebra. In terms of data model Alignment Calculus[grahneSafety], a declarative string database query language is in many ways similar, as is SRS[coupayeSRS] a very widely used system in the gene research field. Of course, the basic idea of parallel markup came from Ted Nelson[nelsonLit].
For querying and algebra for semistructured data or XML, there is an large and growing amount of literature. In Buneman[unQL] a complete query language and algebra based on structural recursion is presented. In Fernandez[xmlMonad] a proposal for a formal algebra for XML Query is presented which might become the standard for node-centric processing. There are many other papers related to retrieval of semi-structured data, including SAL[beeriSAL], XIRQL[fuhrXIRQL] and of course, XQuery[w3xxquery] itself.
For the notion of parsing XML using regular expressions, regular fragmentations from Simon St. Laurent[regularFrag] and REX[cameron98] are especially applicable. Core Range Algebra provides a formal basis for both approaches.

4. Bibliography

adler: Adler, Sharon C, DSSSL- Document Style Semantics and Specification Language, 1989
beeriSAL: Beeri, Catriel and Tzaban Yariv, SAL: An Algebra for Seminstructured Data and XML,
cameron98: Cameron, Robert D., REX: XML Shallow Parsing Using Regular Expressions, 1998
coupayeSRS: Coupaye, Thierry, Extracting and COnverting Data from Semistructured Biological Databanks with SRS, 1996
durandMarkup: Durand, David G and DeRose, Steven J. and Mylonas, Elli, , 1996
fuhrXIRQL: Fuhr, Norbert and Grobjohann, Kai, XIRQL: A Query Language for Information Retreival in XML Documents,
grahneSafety: Grahne, Gosta and Nykanen, Matti, , 1997
nelsonEmbed: Nelson, Theodor Holm, Embedded Markup Considered Harmful, Fall 1997
nelsonLit: Ted Nelson, Literary Machines, 1998
regularFrag: St. Laurent, Simon, Regular Fragmentations, 2001
sperbergDesign1: Sperberg-McQueen, C. M. and Burnard, Lou, The Design of the TEI Encoding Scheme, 1995
unQL: Buneman, Peter and Fernandez, Mary, and Suciu, Dan, ,
w3cdom: W3C DOM Working Group, The Document Object Model (DOM) Level 1, 1998
w3cxsl: W3C XSL Working Group, Extensible Stylesheet Language,
w3xxquery: W3C XQuery Working Group, XQuery 1.0: An XML Query Language,
xmlMonad: Fernandez, Mary and Simeon, Jerome and Wadler, Philip, A Semi-Monad for Semi-Structured Data ,