Attributed Range Algebra

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

March 31, 2002

Abstract

Core Range Algebra defines an algebra of sequences and ranges over sequences that can be used to mark structural boundaries in data, without explicit inline markers. In the context of text, Core Range Algebra offers a single model for both structured texts, such as XML documents, and semi-structured texts, such as email and memos. In addition, Core Range Algebra allows structures to overlap, going beyond the expressive power of popular markup languages, such as XML in the structural dimension. The key limitation of Core Range Algebra is it's limited abilities to model anything other than basic structural boundaries. This paper presents Attributed Range Algebra, an extension of Core Range Algebra that provides greater expressive power such that it can model all structures found in current markup languages.

1. Introduction

In [nicolE2002] we define Core Range Algebra: an algebra of sequences and ranges over sequences that can be used to mark structural boundaries in data, without explicit inline markers. For text, Core Range Algebra offers a single model for both structured texts, such as XML documents, and semi-structured texts, such as email and memos. While Core Range Algebra goes beyond typical markup in that it also allows overlapping structures, it is also limited in it's abilities to model anything other than basic structural boundaries, and the lack of a formal definition of how ranges and sequences are to be constructed.

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  . 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  .

length
The number of ITEMS held by the SEQUENCE.

position
An index into the SEQUENCE which identifies an item. The position will be 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

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*.

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. There is a monoid relationship in this operator given that the empty SEQUENCE represents the unity item, and it also provides us with the means to derive the Kleen 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: RANGE RANGE → BOOLEAN
StartsAfter: RANGE RANGE → BOOLEAN
StartsWithin: RANGE RANGE → BOOLEAN
EndsBefore: RANGE RANGE → BOOLEAN
EndsAfter: RANGE RANGE → BOOLEAN
EndsWithin: RANGE RANGE → BOOLEAN
Within: RANGE RANGE → BOOLEAN
Equals: RANGE RANGE → BOOLEAN

First: SEQUENCE → SEQUENCE
Last: SEQUENCE → SEQUENCE

Create[1]: POSITION LENGTH → RANGE
Extract: RANGE SEQUENCE → SEQUENCE

Start: RANGE → POSITION
End: RANGE → POSITION
Length: RANGE → POSITION

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

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

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 .

StartsAfter(*a*, *b*)
Tests whether *a* starts after *b*, or .

StartsWithin(*a*, *b*)
Tests whether *a* starts within *b*, or .

EndsBefore(*a*, *b*)
Tests whether *a* ends before *b*, or .

EndsAfter(*a*, *b*)
Tests whether *a* ends after *b*, or .

EndsWithin(*a*, *b*)
Tests whether *a* ends within *b*, or .

Within(*a*, *b*)
Tests whether *a* contains *b*, or . Note that this is equivalent to saying 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 .

Operations

First(S)
Given the SEQUENCE *S*, return the first item in *S*, or return an empty SEQUENCE if *S* is empty.

Last(S)
Given the SEQUENCE *S*, return the last item in *S*, or return an empty SEQUENCE if *S* is empty.

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

Extract(*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 .

Move(*R*, *n*)
Move the range R forward of backward by *n*, or .

Resize(R, *n*)
Make the range *R* larger or smaller by n items, or .

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 in increasing order of *length*.

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*.

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.

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*.

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*.

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*.

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. Range Constructors

In the Core Range Algebra, a CREATE operation is defined that takes a position and a length, and creates a RANGE. While this is a necessary operation, it leaves open the question of what is actually creating the ranges. A useful extension to CRA is a *Range Constructor*: a function that given an arbitrary SEQUENCE , returns a SEQUENCE of RANGE.

Definitions

constructor
An operation that takes a sequence, and returns a sequence of ranges.

Syntax

CONSTRUCTOR: SEQUENCE → SEQUENCE (OF RANGES)

Operations

Constructor(*S*)
Given a sequence of ranges *S*, return a new sequence.

The simplest form of Range constructor will take an arbitrary sequence, and using some rule, or set of rules, create a set of associated ranges. In this case the constructor would take one parameter: the range over which the construction rules should be applied. Having applied it's rules, a sequence of zero or more ranges objects would be created and returned.
For example:

T	***h***	***e***		***q***	***u***	***i***	***c***	***k***		***b***	***r***	***o***	***w***	***n***		***f***	***o***	***x***
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Sample text sequence.

Given the above text (a sequence of characters), a speaker of English could easily determine that there are a number of words in the text. Each word has an associated range, and using a range constructor that understood English word boundary processing we could get the following:

{0,3}	{4,5}	{10,5}	{16,3}
1	2	3	4

Derived sequence of ranges

This is simply a SEQUENCE of RANGEs for each word in the text. At this point it should be obvious that a range constructor is very similar to a *lexical analyzer* such as lex[TODO:reference]. Such tools take a stream of bytes, or characters, and produce a SEQUENCe of tokens, while range constructors take a sequence of an arbitrary type and produce a sequence of ranges.

Ranges Construction Using Regular Expressions

Lexical analyzers typically use regular expressions to define a set of patterns to be matched in the input, with each match producing a token. For example:

```
[0-9][0-9]* return(NUMBER);
```

This will match a sequence of digits, and return the `NUMBER` token when a positive match is found.
The applicability of regular expressions to range construction is obvious: given a regular expression, we simply construct a range that matches the start and end of the area matched. For example, a simplistic range constructor for a language using the letters of the English alphabet might be something like the following.

```
[^a-zA-Z][a-zA-Z]+[^a-zA-Z]   construct($1.start, $1.end)
```

Assuming that BOF (Beginning Of File) and EOF (End Of File) are treated as synthetic characters, this will match any sequence of letters from a-z and A-Z surrounded my characters not from that set, and then construct a range for the matched letters. The the following sentence.

T	***h***	***a***	***t***	***'***	***s***		***A***	***n***	***n***	***e***	***-***	***M***	***a***	***r***	***i***	***e***	***?***	***!***
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Sample text sequence.

The regular expression based constructor would create the following sequence of ranges.

{0,4}	{5,1}	{7,4}	{12,5}
1	2	3	4

Derived sequence of ranges

Such a constructor would likely have the regular expression, along with the sequence over which the regular expression should be executed, passed as parameters. In [TODO: Shallow parsing of XML] regular expressions are used to tokenize XML data. Such regular expression-based parsing could (and actually has been) easily be adopted for use in range construction.

Parallel Range Construction

While regular expressions could be used to tokenize data streams, and then construct ranges from the matched patterns, one question is how ambiguous regular expressions are to be handled. For example, the following regular expressions are ambiguous.

```
*day              - Matches anything ending in day.

(Mon|Wednes)day   - Matches Monday or Wednesday.

(Tues|Fri)day     - Matches either Tuesday or Friday.
```

Many lexical scanners do allow parallel matching and will warn of ambiguities in the match rules, or they use declaration ordering, or “greediness” rules in order to handle the ambiguity. These things typically influence language design such that ambiguities are avoided, but this is not always possible, or desirable.
In the case of Range Algebra, one of the primary benefits is precisely it's ability to handle overlapping, and hence non-discrete sequences cleanly, so restricting Range Constructors to only those forms that match discrete sequences would render Range Algebra largely superfluous. In order to handle the ambiguities each regular expression needs to logically be executed independently of all others, with the implication that performance of range constructors is limited by their number and complexity. The fact that range constructors are side-effect free, opens up the possibility of parallel processing of the range constructors, thereby reduction the implicated performance overhead. In the case of range constructors based on regular expressions, each expression could be processed individually, and if a single result was desired, either the **Concat**, or the **Union** operators could be used to group the results, depending on whether and ordered or unordered result is desired. For example, the above regular expressions could be used in a parallel range constructor using something like the following syntax.

```
Union(Constructor(S, “*day”),

      Union(Constructor(S, “(Tues|Fri)day”),

            Constructor(S, “(Mon|Wednes)day”)))
```

Obviously the concrete syntax and language semantics of a Range Algebra implementation would have some effect on the overall applicability of such composition.

4. Attributed Range Algebra

In earlier sections, Core Range Algebra and Range Constructors were introduced. This section develops an extension of Core Range Algebra, which is more suited to representation and interpretation of structured data formats.

Attributed Range Data Type

In the introduction to Core Range Algebra, a range is defined as a {start,length} pair that identify a start and end position within a given sequence. While useful in and of itself, it is equally beneficial to extend this to allow the range to be attributed, or in other words, to attach metadata to the range. Take for example, the following document.

```
EBISU

Ebisu is the Japanese god of money and good fortune.
```

The range {0,5} identifies the subsequence “EBISU”, but tells us nothing more. If the range has added a “type” attribute to get {0,5,type:header}, the attributed range obviously carries more semantics. It is also obvious that this bears some resemblance, in terms of purpose, to XML markup, and if stored, would essentially be a form of external markup[Nelson, et al]. The formal definition of the Attributed Range data type follows.

Definitions

attributed range (arange)

A range[TODO: reference] extended to support an arbitrary set of associated attributes. All attributes, including the well-known attributes start and end, can be accessed using the dot operator. For example, if we have a range **R**, we would write **`R.start`** or **`R.length`**, or if the range has a "foo" attributed, R.foo.

Extended Syntax

Same: ARANGE ARANGE → BOOLEAN
equals: ARANGE ARANGE → BOOLEAN

HasAttribute: ARANGE STRING → BOOLEAN
GetAttribute: ARANGE STRING → SEQUENCE
SetAttribute: ARANGE STRING SEQUENCE → ARANGE
RemoveAttribute: ARANGE STRING → ARANGE

Predicates

The following predicates are defined for ARANGEs.

HasAttribute(arange, attr)
Tests whether *arange* has an attribute attr, or .

Same(arange, arange)[TODO]
Tests whether a given attributed range is the same as another.

Equals(arange, arange)[TODO]
Tests whether a given attributed range is the same as another.

Operations

GetAttribute(R,A)
Given the ARANGE *R*, return the value of the attribute A or null if R has no such attribute.

SetAttribute(R,A,V)
Given the ARANGE *R*, set the attribute A to the value V provided.

RemoveAttribute(R,A)
Given the ARANGE *R*, remove the A attribute from it and return a new range. This is essentially the same as copying the entire range, including all attributes except A.

Regular Expressions
Regular expressions are dependent on the Kleene closure...[TODO]
Ranges -> regular expressions
recursion -> trees
trees -> regular forest grammar

5. Type projection

Range -> node
regular expressions -> schemas
schemas -> validation
partial validation
pure typed lambda calculus

6. Query, extraction, construction

Sequence constructors as a means for transformation
differencing
external dynamic markup – link databases

7. Application to XML

8. 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 Kleen 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.

query, extraction

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. Reference to SGREP
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.

9. 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, UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion,