CMU-ITC-083 2 February 1989

The Computational Power of an Algebra for Subsequences

Wilfred J. Hansen
Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract. This paper introduces an algebra for sequences which has a number of desirable properties as a component of a programming language. In the algebra, sequence values are modeled as *subsequence markers*, each of which refers to a selected subsequence of some underlying base sequence. Four primitive functions on markers are described: next(), start(), extent(), and base(). Together with constants, concatenation, and appropriate definitions for comparison and assignment, these are shown to be necessary and sufficient for expressing all sequence functions. The algebra is further shown sufficient to simulate a Turing machine and thus to compute any computable function. Definitions describe values with symbolic expressions so proofs can be done algebraically rather than axiomatically.

1. Introduction

Sequences are fundamental to computing: a file may be a sequence of records, a record a sequence of fields, a field a sequence of characters. In numerical applications the data in arrays are often processed sequentially. It is surprising, then, that little effort has been devoted to algebras for sequences. This paper is an effort to describe formally one such algebra and prove it satisfies Universal Computability, by showing that it can simulate a Turing Machine.

Each data value in the algebra refers to a subsequence of some underlying sequence, called its *base*. Each single value is an indivisible composite of references to the base and the location and extent of a subsequence. This type of data value is called a "subsequence marker", or "marker" for short. A marker can refer to as much as the entirety of its base or as little as the empty subsequence between two elements. Note that markers for empty subsequences can be distinguished if they have different locations. In Figure 1, the base is "'Twas brillig and the slithy toves" and markers m, n, and p refer to subsequences of that base.

Figure 1 near here

Why is each marker value a subsequence rather than a sequence in its own right? Consider searching and parsing algorithms where it is often desirable that the outcome of a search indicate the location of the result as a start for further processing. When search functions are defined with markers as arguments and results it is convenient and natural to express complex searches with composition of functions. This does not obviate the need for pattern matching, but it does simplify the task of matching patterns that do not happen to fit the particular pattern paradigms provided.

We can contrast subsequence markers with another algebra sometimes considered for character strings: the free monoid consisting of concatenation and strings over an alphabet. We can adapt this algebra to searching and parsing problems by defining function argument and return values as a composite of three string values--the portions before the match, of the match, and after the match. The resulting system is considerably more complex and unnatural than the one presented here because the result of each function match must have a selector applied to extract the desired portion of the composite value.

Another model of string computation found in many programming languages is an array of characters with integers or pointers to indicate position and extent of substrings. From a purely formal standpoint this model is inferior because it requires the domain of integers (or pointers) as well as the domain of sequences. In practice, the array model of strings suffers from other deficiencies: Characters do not always fit in a single byte, or even two bytes, so array subscription may not access distinct characters; string values are limited in length to predeclared storage sizes; and it is not always intuitive--especially when modifying a program--as to whether an integer i refers to the i'th character or the preceding or following inter-character position.

A few languages have successful string processing capabilities despite some of the defects noted above. Snobol4 [Griwold, 1971] has strings of any size and features pattern matching, but lacks the notion of referring to a subsequence of a string. Icon [Griswold, 1979] utilizes integers as references to strings and confuses the issue further by defining negative subscripts as counting from the top end of the string down; two different values can refer to each character position. PostScript [Adobe, 1985] does have the notion of referring to a substring, but given a substring there is no way to access adjacent portions of the parent string.

The algebra described below has been implemented at least three times. First in a research language [Hansen, 1987], then as part of the cT system for educational computing [Sherwood & Sherwood, 1988], and most recently in the Ness language [Hansen, 1989] as part of the Andrew system [Morris, et al., 1986]. A subset of Ness will be used for programming examples below. In addition to the functions of the algebra, the subset utilizes marker declarations, assignment, if-then-else, while-do, predicates, compound statements, function definition, call (with call-by-value parameters), and return. Spare as it may be, this subset was sufficient to write a preprocessor from itself into C.

Section 2 defines the fundamental primitives, applications of which are shown in Section 3 where they are used to define other functions to access substrings. Sections 4 and 5 establish that the algebra is sufficient to compute not only all subsequences of a given sequence, but also all functions. Rather than axiomatic definitions and proofs, markers are defined syntactically and proofs are done with syntactic manipulation. This proof technique, and the elegance of the algebra, combine to render most proofs quite straightforward; indeed, the goal of this paper is not so much to prove results about the algebra, but rather to demonstrate its potential for simple expression of algorithms.

2. Subsequence Primitives

The basis of each value in the algebra is an underlying sequence of elements called the base. An element may be an object of any sort; for examples we will use the alphabet of ASCII characters as the elements. The value itself is a reference to a subsequence of its base, so we call the value a *subsequence marker value*.

For the formal exposition in this paper, the notation for subsequence marker values must represent both the marker subsequence and its position within its base. We write base sequences in angle brackets like this

```
        < s1  c >
```

Identifiers within the angle brackets are meta-variables which refer to individual elements (a, b, c, ...) or to sequences of zero or more elements (p, q, r, s1, s2, ...) . More commonly a marker value will also contain square brackets:

Definition: A *subsequence marker value* (or "*marker*") is written as

< s1 [s2] s3 >

where each s $_i$ is a sequence of zero or more elements. The portion s2 is the *marked* or delimited portion of the sequence. The entire sequence < s1 s2 s3 > is the *base* of the marked value. If s2 is of length zero, the marker is an *empty* marker.

It may appear that the definitions below require copying sequences; the intended interpretation--and the implementation of Ness--does not require copying sequences for any of these functions other than concatenation.

Traditional programming constructs are described as follows for marker values. In the descriptions, the construct is followed by an arrow, =>, and then the resulting value as a bracketed marker value.

Constants return a marker for the entire sequence. For character strings this is:

"*some text*" => < ["*some text*"] >

Concatenation (written with " ~ ") constructs a new sequence by juxtaposing the marked portions of the arguments and returning a marker for the entire new sequence.

< s1 [s2] s3 > ~ **< s4 [s5] s6 >** => **< [s2 s5] >**

Comparison of marker values is defined to be sequence comparison of the delimited portion. This is well defined for equality by reference to equality tests between the elements of the sequences. Inequalities are well defined only if there is a lexicographic ordering of the elements. For character string sequences we have such an ordering for any relational operator *relop*:

< s1 [s2] s3 > *relop* **< s4 [s5] s6 >** => **< s2 >** *relop* **< s5 >**

Assignment of marker values assigns the marker value, it does not copy the sequence. After the assignment

```
a := x
```

if x originally has the marker value < s1 [s2] s3 > then a and x will both have that value after the assignment.

Printing a marker value prints the delimited portion.

print(< s1 [s2] s3 >) => <s2 > is printed

Declarations in the examples will utilize the keyword **marker:**

marker var1, var2, ...

Parameter passing is call by value and passes the entire subsequence marker value, including the base. Thus if f has a formal parameter m and is called with f(p) where p is < s1 [s2] s3 > then f can access any portion of the base of p. All parameters of functions defined in this paper are marker values and parameter declarations are elided.

The algebra of subsequence expressions has the following primitive operations; their arguments and values are all subsequence marker values. The effects of these functions on a typical character string are illustrated in Figure 2.

Figure 2 near here

Start(m) returns a marker both of whose limits are at the beginning of the argument:

start(< s1 [s2] s3 >) => < s1 [] s2 s3 >

Next(m) returns a marker for the element following the argument. If the argument was at the end of its base, next() returns the end of the base.

next(< s1 [s2] c s3 >) => < s1 s2 [c] s3 >
next(< s1 [s2] >) => < s1 s2 [] >

Base(m) returns a marker for the entire sequence surrounding the argument.

base(< s1 [s2] s3 >) => < [s1 s2 s3] >

Extent(m, p) produces a marker extending from the beginning of its first argument to the end of its second argument. If the second argument ends before the start of the first, the result is an empty marker at the end of the *second* argument. Extent() expects that its arguments be on the "same" base, that is, a base generated by a single constant or concatenation. If the bases differ, extent() returns a unique empty constant.

```
extent(< s1 [ s2 ] s3 >, < s4 [ s5 ] s6 >)
        =>  < s1 [ s7 ] s6 >,
                            if < s1 s2 s3 > = < s4 s5 s6 > and there
                            is an s7 such that < s2 s3 > = < s7 s6 >
        =>  < s4 s5 [] s6 >,
                            if < s1 s2 s3 > = < s4 s5 s6 > and there
```

is an s8 such that < s8 s2 s3 > = < s6 >

=> < [] >, **otherwise.**

The equality < s8 s2 s3 > = < s6 > implies that s1, which must end at the same place as s8, will end after the start of s6 and therefore after the end of s5. This complexity can be avoided with a more informal definition that omits some square brackets. For this scheme we adopt the rule that a missing right bracket is somewhere to the right of its left bracket, and vice versa. Then the definition can be:

extent(< s1 [s2 s3 >, < s1 s2] s3 >) => < s1 [s2] s3 >
extent(< s1 s2 [s3 >, < s1] s2 s3 >) => < s1 [] s2 s3 >
extent(< s1 [s2] s3 >, < s4 [s5] s6 >)
=> < [] >, where < s1 s2 s3 > ≠ < s4 s5 s6 >

As written, the formal definition does not require identical bases for the two arguments, only that the base strings be equal. There is no way in the algebra to distinguish two markers on the same base from two markers on different but equal bases. However, implementations of extent() may return the empty string unless the two bases are indeed from the same original source.

An example function utilizing the four primitive subsequence functions is given in Appendix A.

In order for a set of functions to be teachable and memorable, they should be based on an underlying set of functions that are as small as possible. The next section will present a number of additional subsequence functions all of which can be defined in terms of the above four primitives. To show that none of the four is redundant we have:

Theorem 2.1 (Necessity): The four primitive functions--start(), next(), base(), and extent()--are all necessary for computation with the algebra. That is, none can be expressed as a functional composition of the others.

Proof: We wish to demonstrate for each of the primitives, P, that there is no expression *E* which behaves as defined above for P and yet does not contain P in the expression *E*. We do this by exhibiting a particular value, *v*, and argue for each primitive that no such expression *E* exists that converts this value appropriately, and therefore no expression *E* exists which implements P for all arguments. The particular value *v* is

< "ab" ["cd"] "ef" >

Note first that

> next(v) = < "abcd" ["e"] "f" >,
> start(v) = < "ab" [] "cdef" >,   and
> base(v) = < ["abcdef"] >.

For extent() no E can exist because the value returned may have to be of any length and the other functions return only markers of zero, one, or all the elements of a base.

For base() no E can exist because none of the other functions can otherwise generate a marker beginning before the left bracket of v.

For next() no E can exist because no other function creates a marker that starts at the end of v.

For start() there can be no E whose return value is generated by base() or next() because they generate values with end brackets after the end of v. If the return value is generated by extent(), its second argument must end at the start of v, but it must ultimately have gotten this value from base() or next(), which it cannot have done.

Since none of the functions can be expressed in terms of the others, all are necessary. □

Experiments with various alternative sets of primitive functions have shown none more convenient for programming than the set above. For instance, by symmetry the algebra could be defined with previous() instead of next() or finish() instead of start(); but either would emphasize right-to-left processing instead of the more natural left-to-right processing. Next() could return an empty marker after the following element, but algorithms often require elements rather than empty markers.

There is an algebra which is formally simpler than the one above, requiring three primitive operations instead of four. It utilizes next() and extent() and a third function we can all startofbase():

StartOfBase(m) returns a marker for the empty sequence at the beginning of the entire sequence surrounding the argument.

> **startofbase(< s1 [s2] s3 >) => < [] s1 s2 s3 >**

Implementation of base() with the three primitives requires only a loop to find the end of the base and an extent() to combine start and end. Start() is slightly trickier:

```
function start(m)
        marker p;
        p := startofbase(m);
        if extent(p, m) = "" then return m; end if;
        while extent(next(p), m) /= m do
```

```
                p := next(p);
            end while;
            return extent(m, p);   -- p ends where m begins
        end function;
```

The set of three functions, however, is far less convenient for programming than the four primitives presented above.

3. Subsequence Functions

Given a marker, many algorithms require computation of markers for nearby subsequences; within the algebra the expressions for such computations are usually concise. This section presents formal definitions for a number of nearby subsequences, exhibits expressions to compute the desired marker, and proves that that expression does indeed have the desired value. As a first example, we compute finish(), the analog of start() which is the empty marker following a marker value.

finish(m) - Returns a marker for the empty subsequence just after m:

finish(< s1 [s2] s3 >) => < s1 s2 [] s3 >.

Lemma 3.1: An expression for finish(m) is start(next(m)).

Proof: If s3 is non-empty it contains a first element c and a continuation s4:

start(next(< s1 [s2] c s4 >))
= start(< s1 s2 [c] s4 >)
= < s1 s2 [] c s4 >
= < s1 s2 [] s3 >

while if s3 is empty we have:

start(next(< s1 [s2] >))
= start(< s1 s2 [] >)
= < s1 s2 [] >
= < s1 s2 [] s3 >

In both cases the expression computes the correct value. □

Much of the remainder of this section is directed toward computing single element subsequences analogous to next(), which is the first element after the marker; we want the first element before the marker and the first and last elements within the marker. We begin with the single element marker which starts at the same place as its argument:

front(m) - Returns a marker for the first element after start(m), if there is one, otherwise m must be empty at the end of its base and this value is returned:

> front(< s1 [c s2] s3 > => < s1 [c] s2 s3 >
> front(< s1 [] c s3 >) => < s1 [c] s3 >
> front(< s1 [] >) => < s1 [] >

Lemma 3.2: An expression for front(m) is next(start(m)). (Proof omitted.) □

Front() returns a single element regardless of whether its argument is the empty string, but sometimes it is preferable to have a function first() which is empty when its argument is. To define first() it is easiest to begin with rest(), a function to compute all but the first element. The implementation of this function exploits the precise definition of extent().

rest(m) - Returns a marker for all elements of m other than the first; but if m is empty, so is rest(m):

> (i) rest(< s1 [c s2] s3 >) => < s1 c [s2] s3 >
> (ii) rest(< s1 [] s3 >) => < s1 [] s3 >

Lemma 3.3: An expression for rest(m) is extent(next(next(start(m))), m).

Proof: The proof of (i) has three cases depending on the lengths of s2 and s3. When s2 has at least one element we write s2 as < c2 s4 > and the proof proceeds thus:

```
rest(< s1 [ c c2 s4 ] s3 >)
        = extent(next(next(start(< s1 [ c c2 s4 ] s3 >))),
                < s1 [ c c2 s4 ] s3 >)
        = extent(next(next(< s1 [] c c2 s4 s3 >)), < s1 [ c c2 s4 ] s3 >)
        = extent(next(< s1 [ c ] c2 s4 s3 >), < s1 [ c c2 s4 ] s3 >)
        = extent(< s1 c [ c2 ] s4 s3 >), < s1 [ c c2 s4 ] s3 >)
        = < s1 c [ c2 s4 ] s3 >
```

In the second case s2 is empty and s3 has one or more elements, while in the third case s2 and s3 are both empty. In both cases the marker is reduced to a empty subsequence at its former end. These cases can be verified by an argument similar to that of the first.

For part (ii) we observe that the first argument to extent() is next(next(start(m))), which cannot yield a marker starting before the beginning of m and the second argument is m, which ends at the end of m. Since m is empty, the extent() must yield a marker equivalent to m. □

first(m) - Returns the first element of m, but if m is empty, so is first(m):

> first(< s1 [c s2] s3 >) => < s1 [c] s2 s3 >
> first(< s1 [] s3 >) => < s1 [] s3 >

Lemma 3.4: An expression for first(m) is extent(m, start(rest(m))). (Proof omitted.) □

allprevious(m) - Returns a marker for the subsequence of the base of m that precedes the start of m:

allprevious(< s1 [s2] s3 >) => < [s1] s2 s3 >

Lemma 3.5: An expression for allprevious(m) is extent(base(m), start(m)). (Proof omitted.) □

allnext(m) - Returns a marker for the subsequence of the base of m that follow the end of m:

allnext(< s1 [s2] s3 >) => < s1 s2 [s3] >

Lemma 3.6: An expression for allnext(m) is extent(finish(m), base(m)). (Proof omitted.) □

last(m) - Returns the last element of m, but if m is empty, so is last(m):

(i) last(< s1 [s2 c] s3 >) => < s1 s2 [c] s3 >
(ii) last(< s1 [] s3 >) => < s1 [] s3 >

Lemma 3.7: Last(m) is computed by this function:

```
function last(m)
        if rest(m) = "" then return m;
        else return last(rest(m));
end function
```

{This function is grossly inefficient. However, when the implementation of the algebra stores elements in contiguous memory last() can be computed by scanning backward in the base sequence.}

Proof: Case (ii) follows trivially from the definition of rest().

Case (i) must be proved by induction on the length of m. If m has one element, then rest(m) is "" by the definition of rest(), so m is correctly returned as its own last element. If m has more than one element, we write it as < s1 [c2 s2 c] s3 >, the else clause is executed and we return last(< s1 c2 [s2 c] s3 >). By the inductive hypothesis, the latter expression gives the correct value. □

previous(m) - Returns a marker for the element preceding m, but if m is at the beginning of its base sequence, previous(m) returns the value start(m):

(i) previous(< s1 c [s2] s3 >) => < s1 [c] s2 s3 >
(ii) previous(< [s2] s3 >) => < [] s2 s3 >}

Lemma 3.8: An expression for previous(m) is last(allprevious(m)). (Proof omitted.) □

In algorithms which transform sequences it is desirable to retain the position of some markers while modifying the base sequence at the position given by another marker. One method for doing this is to define replace(a, s) which modifies the base sequence of a, replacing the marked portion with the marked portion of s and adjusting appropriately all other markers on base(a). Such a function is defined in the Ness implementation, but is unnecessary for the present paper. A second method for remembering the position of a marker is to define two functions length(m) and nextn(a, n). The former computes the length of a sequence and the latter returns the result of applying the next function n times to the argument a. Although these can be useful functions, this paper does not require recourse to the domain of integers. Instead we define a function countdown(m, p) which advances through m by the length of p.

countdown(m, p) - returns a marker for the element of m found by applying next to start(m) once for each element of p:

(i) countdown(< s1 [s2] s3 >, < s4 [s5] s6 >)
=> countdown(< s1 [] s2 s3 >, < s4 [s5] s6 >)
(ii) countdown(< s1 [] s7 >, < s4 [s5] s6 >)
=> < s1 s7 [] >,
where length(s7) ≤ length(s5)
(iii) countdown(< s1 [] s8 c s9 >, < s4 [s5] s6 >)
=> < s1 s8 [c] s9 >,
where length(s8) = length(s5)

Lemma 3.9: This function implements countdown():

```
-- countdown(m, p)
--      return a marker for the single element c from base(m)
--      such that the number of elements in extent(m, start(c))
--      is equal to the number of elements in p (or if p is
--      too long, then c is next(base(m))
--
function countdown(m, p)
        m := next(start(m));
        while p /= "" do
                m := next(m);
                p := rest(p);
            end while;
            return m;
        end function;
```

Proof: Case (i) is covered by the "start(m)" in the first assignment. Case (ii) is not needed for further proofs below and will be omitted.

The trivial portion of case (iii) is when s5 is empty and the loop body is not entered. In this instance, s8 is also empty and the first assignment in the function body makes m refer to c, as required.

For the general part of case (iii) we have the while loop invariant:

> Just before the first assignment in the while loop, the variables have these values:
>
> m is < s1 si [b] sj s9 >
> p is < s4 su [sv] s6 >
>
> where < si b sj > = < s8 c > , < su sv > = < s5 > , and length(si) = length(su).

On first entering the loop, the variables satisfy the invariant with si and su both empty: si has been established by the initial assignment and su is initially empty.

Execution of the two statements of the loop body advances m to the next element (by the definition of next()) and reduces p to have one less element at the front (by the definition of rest()). Thus si and su are each one element longer, sj and sv are each one element shorter, and the invariant is preserved.

When sv eventually becomes empty, the loop exits. Then su must be all of s5, so si is all of s2, b is c, and sj is empty. Thus we have that m is < s1 s2 [c] s3 > which value is returned after the loop, satisfying case (iii). □

Corollary 3.1 (CountFront): If c is a single element and m and n are arbitrary markers, countdown(m ~c ~n, m) has the value < m [c] n >. (Proof omitted.) □

4. Sufficiency

With the aid of the subsequence functions, it is possible to write an expression for any subsequence of a sequence. To demonstrate this, consider the set of *all* subsequences of a sequence. This set consists of each instance of a subsequence starting at one position in the sequence and continuing to the same or another, later position. Here are functions to print all the subsequences of a sequence:

```
function printsubsequences(s)
        printsubsub(s, s);
        if s /= "" then
                printsubsequences(rest(s));
        end if;
end function;

function printsubsub(t, s)
        print (extent(s, start(t)));
        if t /= "" then
                printsubsub(rest(t), s);
        end if;
end function;
```

It is not difficult to show that printsubsequences(s) prints all subsequences of s. We begin with a lemma.

Lemma 4.1 (Tail Recursion): If $\underline{P}$ is an operation and $\underline{Q}$ is a sequence of zero or more variables each preceded by a comma, then the function f() defined by

```
function f(x  Q)
        P(x  Q);
        if x /= "" then
                f(rest(x)   Q);
        end if;
end function;
```

performs $\underline{P}$ once for each *tail* of x, including the final empty subsequence. That is, if x is < s1 [c1 c2 . . . cn] s3 > then $\underline{P}$ is executed for each of

< s1 [c1 c2 . . . cn] s3 >,
< s1 c1 [c2 . . . cn] s3 >,
. . .,
< s1 c1 c2 . . . [cn] s3 >, and
< s1 c1 c2 . . . cn [] s3 >.

Proof: If x is < s1 [] s3 >, then n is zero and $\underline{P}$ is executed once; the then clause is not executed because x = "". When $n > 0$ we argue by induction. A call to f() evaluates $\underline{P}$ once for the current value of x and then calls f recursively for rest(x). By the definition of rest, rest(x) is a tail of x so n is one less and the general case holds by induction. □

Lemma 4.2: The call printsubsub(s, s) prints all subsequences of s that begin at start(s).

Proof: By the Tail Recursion Lemma, printsubsub(s, s) executes print(extent(s, start(t))) for t being each tail of s. This is exactly the subsets of s that begin at the beginning of s. □

Lemma 4.3: The call printsubsequences(s) print all subsequences of s.

Proof: By the Tail Recursion Lemma, printsubsequences(s) executes printsubsub(s, s)

for *s* being each tail of the initial *s*. By the preceding Lemma, this call prints the subsequences beginning at each position within *s*, which is the entire set of subsequences. □

Theorem 4.1 (Sufficiency): The subsequence algebra is sufficient to generate all subsequences of the base of a sequence.

Proof: By the preceding Lemma, if *s* is a sequence, the call printsubsequences(base(*s*)) will print all subsequences of the base of *s*. Since they are printed, they must have been generated. Since only the functions of the algebra have been used to operate on sequence values, those functions must be sufficient. □

Note that the Sufficiency Theorem proves that all subsequences can be generated, but not that any specified subsequence or set of subsequences can be generated. This more general result will be established in the next section.

5. Universal Computability

So far we have seen that the algebra is necessary and sufficient to compute all subsequences of any base. The next, and final, step is to demonstrate that any function whatsoever can be computed; to do so we rely on Church's Thesis that a system can compute any function if it can simulate a Turing Machine:

Theorem 5.1 (Universal Computability): The subsequence algebra is sufficient to simulate a Turing Machine having a tape with symbols of 0 and 1, a read head which is examining one symbol of the tape, a state machine, and five operations: move head Left, move head Right, write a Zero, write a One, and Halt. For each combination of state and symbol under the read head, the state machine specifies an operation and a next state. (Precise specification of each operation is given below.)

Proof: Consider the Turing Machine as a function TM: T -> T where the domain T is the set of all Turing tapes. The proof proceeds by defining an injection H from T to strings and another G from Turing Machines to Ness programs. Then it will show that for any initial tape $t \in T$ and initial state s we have

$$H(TM(t, s)) = (G(TM))(H(t), s'), \qquad (5.1)$$

that is, that the result of applying TM to t and then translating to a string is the same as translating t to a string and applying the Ness program corresponding to TM. (s' is the representation of state s in the translated Turing Machine.) Since H is an injection the only way to have acheived (5.1) is for G(TM) to be a successful simulation of TM.

The proof is by induction over the number of state transitions executed by TM. To reflect transitions, we convert Relation (5.1) to the indexed form

$$H(TM_i(t, s)) = (G(TM_i))(H(t), s'), \quad (5.2)$$

where the subscript indicates the number of state transitions that have occurred. TM_0 is the identity function, so if $(G(TM_0))(H(t), s') = H(t)$, then (5.2) is trivially preserved. We need only show that each state transition in G(TM) preserves (5.2). The heart of the proof will demonstrate that the relation is preserved by the implementations of each of the five operations. Prior to that, we exhibit the translations H and G:

The tape translation H. Each element of T is an infinite sequence consisting of a finite segment of mixed 0's and 1's preceded and followed by infinite sequences of 0's. The read head is positioned over one of the elements of the finite sequence. Capital letters are variables over the sequence: P, Q, and R for subsequences, A, B, and C for individual 0's and 1's. The symbol under the read head is underlined, so a typical tape is

$$0 \ldots P\,\underline{C}\,Q\,0 \ldots$$

H transforms a tape to a subsequence marker value whose base is the finite string of "0"s and "1"s corresponding to the middle segment of the tape; the infinite tails in both directions are inserted by the Left and Right operations as required. The marked portion of the value is the single character where the read head is positioned. Lower case letters are variables over the strings, so the representation of the tape above is

$$H(0 \ldots P\,\underline{C}\,Q\,0 \ldots) = <s1\ [\,c\,]\ s2>,$$

where s1, c, and s2 have the ASCII characters "0" and "1" for the 0 and 1 symbols, respectively, of P, C, and Q.

That H is an injection can be seen by noting that if two tapes differ one will have a 0 where the other has a 1, but in that case the translations will differ because one will have a "0" where the other has a "1".

The Machine translation G. The translation of any Turing Machine into Ness has an outer framework which is the same for all TM and an inner sequence of lines unique to the particular TM. State names are represented as strings consisting of whatever names are given to the states in TM. Thus the translation G(TM) is a Ness function with a tape and a state name as arguments and a tape as the result. The unvarying framework of this function is:

```
function GTM(t, state)
        marker simulating;
        simulating := "yes";
        while simulating do
                if false then
                << one line for each transition >>
                end if;
        end while;
```

```
            return t;
        end function
```

(The *if false then* line is included only so every transition line can begin with *elif*.)

A "transistion" is the combination of an input state and a value under the read head; the line in the program for input state $\underline{u}$, read head value $\underline{c}$, operation $\underline{op}$, and new state $\underline{v}$ is:

elif state = "$\underline{u}$" and t = "$\underline{c}$" then t := $\underline{Op}$(t); state := "$\underline{v}$"

unless the operation is Halt, in which case the line is

elif state = "$\underline{u}$" and t = "$\underline{c}$" then simulating := "No"

The $\underline{Op}$ function depends on $\underline{op}$, the operation specified by TM for the particular transition:

op		*Op*
write a One	->	OpOne
write a Zero	->	OpZero
move head Left	->	OpLeft
move head Right	->	OpRight

The framework executes the while loop once for each state transition, applying one of the four OpXxxx functions to transform $H(t)_i$ into $H(t)_{i+1}$. The Halt operation sets simulating to "No", so the while loop terminates whenever TM halts. When no state transitions occur, the original value of t is returned. It remains to show that each OpXxxx function preserves relation (5.2).

OpOne() and *OpZero()*. By symmetry it suffices to demonstrate OpOne(). The precise specification of the 'write a One' operation for TM is

$$0 \ldots P\,\underline{C}\,Q\,0 \ldots \;\; \text{-->} \;\; 0 \ldots P\,\underline{1}\,Q\,0 \ldots$$

The Ness function accompanying GTM() is

```
    function OpOne(t)
            return countdown(allprevious(t)  ~ "1"  ~ allnext(t),
                        allprevious(t));
    end function
```

By Corollary "CountFront" and the definitions of allprevious() and allnext(), this function has the effect

OpOne(< s1 [c] s2 >) => < s1 [1] s2 >

so by inspection relation (5.2) is preserved.

OpRight. For TM, the precise specification of 'move head Right' is

$$0 \ldots P\ \underline{B}\ C\ Q\ 0 \ldots \dashrightarrow 0 \ldots P\ B\ \underline{C}\ Q\ 0 \ldots$$

The corresponding Ness function is

```
function OpRight(t)
        if next(t) /= "" then
                return next(t)
        else
                return countdown(base(t)   ~ "0"   ~ "", base(t));
end function
```

(The ~ "" is merely to satisfy the condition of "CountFront")

There are two cases, depending whether the read head is at the end of the finite middle segment of the tape.

Case 1: next(t) /= "". In this case, there must be a character c such that t has the form

< s1 [b] c s2 > .

The first return statement is executed and, by the definition of next(), returns the value:

< s1 b [c] s2 >

Case 2: next(t) = "". The form of t is

< s1 [b] >

and the second return statement is executed. It first constructs by concatenation a new value

< [s1 b 0] >

but by Corollary "CountFront", the value returned is

< s1 b [0] >

In both Cases, the value returned corresponds to

$$H(0 \ldots P\ B\ \underline{C}\ Q\ 0 \ldots)$$

so (5.2) is preserved.

OpLeft. For TM, the precise specification of 'move head Left' is

$$0 \ldots P B \underline{C} Q 0 \ldots \;\text{-->}\; 0 \ldots P \underline{B} C Q 0 \ldots$$

The corresponding Ness function is

```
function OpLeft(t)
        if previous(t) /= "" then
                return previous(t)
        else
                return countdown(""   ~ "0"   ~ base(t), "");
end function
```

(The "" ~ is merely to satisfy the condition of "CountFront")

The proof of OpLeft is similar to that for OpRight except it depends on Lemma "Previous" instead of the definition of next().

To summarize, GTM() is the identity function if no state transitions are executed and (5.2) is preserved by each state transition. Therefore, we conclude by induction on the number of state transitions that (5.1) is preserved for all TM. Since GTM() halts exactly when TM executes the Halt operation, GTM() will be non-terminating whenever TM is. Since GTM() is the transformed function G(TM), the latter simulates TM. □

The algebra is in fact equivalent to a Turing Machine, an assertion which requires proof of the obverse of the above theorem: that a Turing Machine can simulate programs in the algebra. The heart of such a proof is presented in Appendix B.

Summary and Conclusion

This paper has presented an algebra for subsequence expressions, employing as primitives the functions base(), start(), next(), extent(), concatenation, constants, and appropriate definitions for assignment and comparison. The four functions were shown to be necessary and sufficient for computing all subsequences of a given base sequence and were shown--by simulation of a Turing Machine--to implement universal computability. While these properties demonstrate that this is a complete algebra, its real value is as a tool for describing algorithms over sequences. The algebra especially facilitates string parsing since a single value can represent the entire substring matched by a pattern match. The algebra has already been implemented as the string sublanguage for two programming languages and has attracted a number of enthusiastic users.

Acknowledgements. This work began in reaction to Bruce Sherwood's search for a string sublanguage for the cT educational computing environment. I am indebted to him, Judy Sherwood, David Andersen, the Center for Design of Educational Computing, and cT users. Initial development of these ideas was done while in residence at the Department of Computer Science, University of Glasgow, with the support of the Science and Engineering Research Council (grant number GR/D89028), both under the direction of Malcolm Atkinson. The work has also benefitted from conversations with Kieran Clenaghan, David Harper, Joe Morris, John Launchberry, Mark Sherman, and John Howard.

References

[Adobe, 1985] Adobe Systems, Inc., *Postscript Language: Reference Manual,* Addison-Wesley, (Reading, Mass., 1985).

[Griswold, 1971] Griswold, R. E., J. Poage, and I. P. Polonsky, *The Snobol4 Programming Language,* Prentice-Hall (Englewood Cliffs, 1971).

[Griswold, 1979] Griswold, R. E., D. R. Hanson, and J. T. Korb, "The Icon Programming Language: An Overview," *SIGPLAN Notices 14*, 4 (April, 1979) 18-31.

[Hansen, 1987] Hansen, W. J., *Ness - Reference Manual*, Computer Science Dept., Univ. of Glasgow, 1987.

[Hansen, 1989] Hansen, W. J., *Ness: Reference Manual*, Information Technology Center, Carnegie-Mellon University (January, 1989).

[Morris, 1986] Morris, J., Satyarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., Smith, F. D. "Andrew: A distributed Personal Computing Environment," *Comm. ACM*, V. 29, 3 (March, 1986) 184-201.

[Sherwood, 1988] Sherwood, B. A., and Sherwood, J. N., *The cT Language*. Stipes Publishing Company, (Champaign, Illinois, 1988).

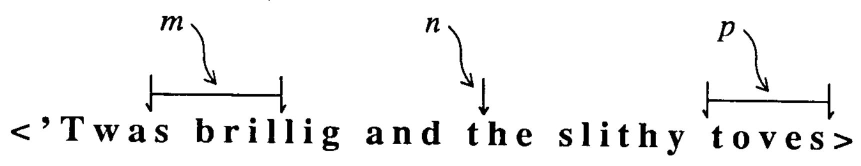
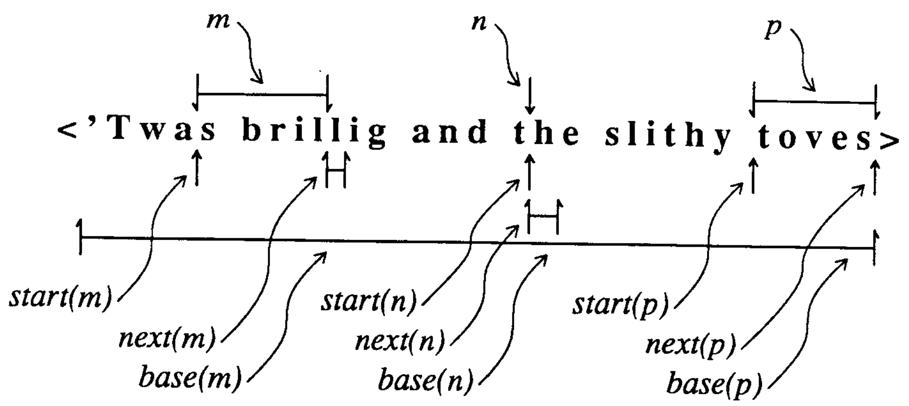


Figure 1. Illustration of three markers. The base sequence is "'Twas brillig and the slithy toves"; marker *m* refers to the subsequence "s bril", *n* refers to the empty subsequence between the first two letters of "the", and *p* refers to "toves".



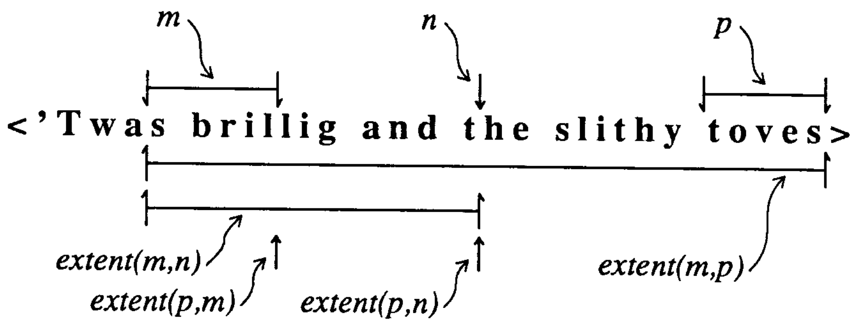


Figure 2. The four primitive functions. The base sequence for all examples is as in Figure 1. The markers below the base show the result of applying the various functions to *m*, *n*, and *p*.

Appendix A. Example function

Here is a function utilizing the algebraic operations introduced in section 2. It copies an input string and produces a modified version with tab characters replaced with enough spaces to make the following text at appropriate column positions.

For each cycle through the main loop, c refers to a different character from the input m. Variable m, is advanced through the text with the expression extent(next(c), m), which--due to the definition of extent()--will be the empty string when c is the last character of m. Each time a tab is encountered, a segment from the original input is copied to the output 'new' and followed with the appropriate number of spaces instead of the tab.

```
-- ExpandTabs(m)
--      return a copy of m with tabs expanded to spaces so
--      subsequent text is at position 9, 17, 25, 33, ...
--
function ExpandTabs(m)

        marker c;        -- current character
        marker tab;      -- replace c, if it is a tab
        marker p;        -- start of current segment
        marker eight;    -- eight spaces;
        marker new;      -- the output string

        eight := "        ";    -- 8 spaces
        tab := eight;    -- distance to 1st tab
        new := "";       -- init output to an empty string
        p := m;          -- set start of first segment

        while m /= "" do
                c := next(start(m));       -- first character
                m := extent(next(c), m);   -- rest of text
                if c = "\t" then
                        -- tab: copy preceding segment
                        --      and expand tab
                        new := new

                                  ~ extent(p, start(c))   --
segment

                                  ~ tab;        -- replacement
spaces
                        p := m;
                        tab := eight;
                elif c = "\n"  or next(tab) = "" then
                        -- newline or 1 space tab;
                        --      restart tab cycle
                        tab := eight;
                else
                        -- non-tab: shorten tab replacement
                        tab := rest(tab);   -- (see Lemma 3.3)
                end if;
        end while;

        return new   ~ extent(p, m);
    end function;
```

Note the absence of arithmetic for determining the current output position. As an exercise the reader is invited to write a version of ExpandTabs() that keeps track of tab position with integers

Appendix B. Turing machine implementation of the primitive functions

The text has shown how to simulate a Turing machine with the algebra, and therefore that the algebra can compute any computable function. To demonstrate that the algebra is not some more powerful computational engine, we sketch a proof of equivalence to a Turing machine by showing that a Turing machine can simulate the algebra.

Rather than program in machine language for the Turing machine, we write Lisp functions for the four primitives. The representation of a sequence itself is simply a Lisp list: each subsequence marker value is a list of three pointer, one to the base sequence, a second to the first element of the marked portion, and the third to the element after the marked portion. If the marker ends (resp. begins) at the end of its base, the third pointer (resp. second) is NIL. In a marker which represents an empty sequence on an empty base all three pointers are NIL.

```
(defun  next (m)    (list
            (car m)              ; same base
            (caddr m); start just after end of old value
            (cond                ; extend for zero or one element
                        ((null (caddr m))  NIL )        ; end of sequence: zero
                        ( T  (cdr (caddr m)))           ; otherwise: one
            )
))

(defun start (m)  (list
            (car m)              ; same base
            (cadr m)             ; same start as existing value
            (cadr m)             ; empty marker at the start
))

(defun base (m)  (list
            (car m)              ; same base
            (car m)              ; start of base
            NIL                  ; end of base
))

(defun istail (a c)  (and c (or (eq a c) (istail a (cdr c)))))

(defun extent (m p)  (cond
            ((or (null (car m)) (not (eq (car m) (car p))))
                                 ; not on a base, or differing bases
                                 (list NIL NIL NIL))
            ((or (null (cadr m)) (istail (cadr m) (caddr p)))  (list
                                 ; empty marker at start of second argumenrt
                                 (car m)
                                 (caddr p)
                        (caddr p)
          ))
          ( T  (list
                        ; extend from start of first arg to end of second
                        (car m)
                        (cadr m)
                        (caddr p)
          ))
))
```

In a practical implementation, sequence elements would be stored in consecutive locations so the loop implied by istail() would be implemented as no more than an address comparison.

Formal Definition of Replace()

In some applications, such as interactive text editors, it is desirable to be able to modify base strings, so another primitive function--replace()--is required. Informally the meaning of replace(x, y) is to modify the base string of x by removing the current contents of the subsequence x and inserting a copy of the subsequence y. The value returned is a subseq for a value that appears to be y, but is in the place of x in x's base. Replace() subsumes insertion and deletion: for insertion, an empty subseq is replaced with non-empty text; for deletion, a non-empty subseq is replaced with empty text.

An informal definition of replace in the notation we have so far used is this:

```
replace(< s1 [ s2 ] s3 >, < s4 [ s5 ] s6 >)
          => < s1 [ s5 ] s3 >,
                 if < s1 [ s2 ] s3 > is not a constant
          => Error,  otherwise
```

> Replace(x, y) modifies the base of the subseq value x so the subsequence referred to by x now contains the marked portion of string y instead of its former value. The value returned is a subseq delimiting the new copy of the replacement value. All other subseq values on the same base as x are adjusted appropriately, as defined below.

Informally, replace(x, y) affects other subseqs on the same base as x as though the value of y is inserted at the end of x and then x is deleted. Thus subseqs that begin after x are shifted along so they still refer to the same underlying text as they did and subseqs that span x will have new contents for the portion that was x. For subseqs which end at the end of x there are four cases, depending on whether each of x or the other subseq is empty or not; the action for each case is described in Table 1 and illustrated in Figure 1.

	x is empty	x is non-empty
other is empty	other precedes insertion	other follows insertion
other is non-empty		other includes the insertion

Table 1. Adjustment of other subseqs that end where x does. The operation performed is replace(x, y). 'Other' is some other subseq value on the same base as x and ending where x ends. The contents of each cell describe the relation of 'other' to the inserted copy of y.

Rather than theoretical considerations, the basis of Table 1 is practical experience with the algebra. In most cases it has seemed preferable that empty subseqs remain empty after a replace. An empty subseq often indicates a position to begin further processing, so when the x and the other subseq are both empty the other is left preceding the insertion so the insertion will be processed. But when x is non-empty the other subseq originally followed x and should still follow the replacement.

Replace "efgh" in abcd efgh ijkl with "*xyz*"

	The substring	a / bcd / efghijkl	becomes	a / bcd / *xyz*ijkl
	The substring	a / bcde / fghijkl	becomes	a / bcd / *xyz*ijkl
<	The substring	a / bcdefgh / ijkl	becomes	a / bcd*xyz* / ijkl
	The substring	a / bcdefghi / jkl	becomes	a / bcd*xyz*i / jkl
	The substring	abcd / efg / hijkl	becomes	abcd / / *xyz*ijkl
<	The substring	abcd / efgh / ijkl	becomes	abcd / *xyz* / ijkl
	The substring	abcd / efghi / jkl	becomes	abcd / *xyz*i / jkl
	The substring	abcde / fg / hijkl	becomes	abcd / / *xyz*ijkl
<	The substring	abcde / fgh / ijkl	becomes	abcd / *xyz* / ijkl
	The substring	abcde / fghi / jkl	becomes	abcd / *xyz*i / jkl
<>	The substring	abcdefgh / / ijkl	becomes	abcd*xyz* / / ijkl
>	The substring	abcdefgh / i / jkl	becomes	abcd*xyz* / i / jkl

Replace empty string between c and d in abcdef with "*xyz*"

<	The substring	a / bc / def	becomes	a / bc / *xyz*def
	The substring	a / bcd / ef	becomes	a / bc*xyz*d / ef
<>	The substring	abc / / def	becomes	abc / / *xyz*def
>	The substring	abc / d / ef	becomes	abc*xyz* / d / ef

Table 1. The effect of replace() on other subseqs. The replace performed for each group of lines is shown above the group. Each line shows an example of some other subseq on the same base and its value after the replacement. The base string is letters only; the spaces are for readability and the slashes indicate the extent of each other subseq value. In general the replacement is made by inserting the replacement at the finish of the replaced value and then deleting the replaced value. The <'s mark examples where the other subseq ends at the end of the replaced string and the >'s indicates examples where it begins there.

To formally model replace() we must introduce the notion of a memory with a collection of base strings each having some number of subseqs. The entire contents of memory is represented in curly braces:

{ . . . , < s1 [s2] s3 >, . . . < s4 [s5] s6 > , . . . }

To indicate that subseqs share the same base, we label the brackets of each subseq. Thus two subseqs on the same base in memory would be:

$\{ \ldots, < s1\ [_j\ s2\ [_k\ s3\ ]_j\ s4\ ]_k\ s5 >, \ldots \}$

where the subsequence referred to by *j* is s2 s3 and that of *k* is s3 s4. We require that the labels on all pairs of brackets be unique, so a label both identifies a base and refers to a subsequence within it.

For convenience in the formal definition, we introduce the notion of an "extended sequence" composed of intermixed brackets and elements from a base sequence. To distinguish the two forms of sequence the component items in an extended sequence are called "constituents"; each constituent is either an element of a base sequence or a left or right bracket which represents one end of a subseq on that base. For each label in a well-formed extended sequence there is one left bracket and one right bracket with that label and the left bracket precedes the right. The Greek letters appearing as meta-linguistic variables below take as values subsequences of extended sequences (but not references to these subsequences).

Using bracket labels to indicate subseq values, the function we wish to define is

replace(α, β) => v

which has two subseq arguments and returns a third. The memory representation prior to executing the function is

$$\{ \ldots < sb1\ [_\alpha\ sb2\ ]_\alpha\ sb3 > \ \ldots < sb4\ [_\beta\ sb5\ ]_\beta\ sb6 > \ldots \}$$

where

> **sb*i*** is a subsequence of the extended sequence which includes all the elements of s*i* and all adjacent brackets other than those explictly shown.

The subsequences α and β may overlap on the same base, so the formal definition must make a copy of β. It does so by introducing a new base string in the memory representation with an otherwise unused label denoted by χ. The first rule just makes this copy:

$$\textbf{(1)}\ \{ \ldots < \mathbf{sb1}\ [_\alpha\ \mathbf{sb2}\ ]_\alpha\ \mathbf{sb3} > \ \ldots < \mathbf{sb4}\ [_\beta\ \mathbf{sb5}\ ]_\beta\ \mathbf{sb6} > \ldots \}$$

$$=> \{ \ldots < \mathbf{sb1}\ [_\alpha\ \mathbf{sb2}\ ]_\alpha\ \mathbf{sb3} > \ \ldots < \mathbf{sb4}\ [_\beta\ \mathbf{sb5}\ ]_\beta\ \mathbf{sb6} > \ \ldots < [_\chi\ \mathbf{s5}\ ]_\chi > \}$$

For the case of a non-empty α, we rewrite sb2 as s7 c b7, where

c is the last element--that is, a non-bracket--in sb2 (by assumption, sb2 has at least one character)
s7 is all elements in sb2 that precede c
b7 is all brackets from anywhere in sb2, taken in their original order

The required effect is to remove s7 and c and place all the brackets of b7 after the insertion. Note that the copy χ is deleted and the placement is shown for the result subseq ν :

$$\begin{array}{l}\textbf{(2)}\ \{\ldots<\ \textbf{sb1}\ [_\alpha\ \textbf{sb2}\ ]_\alpha\ \textbf{sb3}>\\ \qquad\qquad\qquad \ldots<\ [_\chi\ \textbf{s5}\ ]_\chi>\}\\ \quad \Rightarrow\{\ldots<\ \textbf{sb1}\ [_\alpha\ [_\nu\ \textbf{s5}\ ]_\nu\ ]_\alpha\ \textbf{b7}\,\textbf{sb3}>\\ \qquad\qquad \ldots\}\end{array}$$

For an empty α we assume that sb2 is also empty and rewrite sb1 and sb3 extracting and splitting the set of all brackets between the last non-bracket element of sb1 and the first character of sb3:

sx1 is all of sb1 except the brackets on its right. If sb1 has no elements, sx1 is empty.
sx3 is all of sb3 except the brackets on its left. If sb3 has no elements, sx3 is empty.
b2 is all brackets between sx1 and sx3.

We rewrite b2 as b1 b3 where:

b1 is is all brackets from b2 which are right brackets or the left brackets of empty subsequences.
b3 is all left brackets from b2 which do not begin empty subsequences.

Since $[_\alpha$ and $]_\alpha$ bound an empty subsequence between s1 and s3, they are constituents in b1. The third rule provides for inserting the replacement text s5 between b1 and b3, and again deleting the copy χ and indicating the return value ν .

$$\begin{array}{l}\textbf{(3)}\ \ \{\ldots<\ \textbf{sx1}\ \textbf{b1}\ \textbf{b3}\ \textbf{sx3}\ >\\ \qquad\qquad\qquad \ldots<\ [_\chi\ \textbf{s5}\ ]_\chi>\}\\ \quad \Rightarrow\{\ldots<\ \textbf{sx1}\ \textbf{b1}\ [_\nu\ \textbf{s5}\ ]_\nu\ \textbf{b3}\ \textbf{sx3}\ >\\ \qquad\qquad \ldots\}\end{array}$$

It may be desirable to create an empty string into which text can be inserted with replace(). For this we provide a primitive read-write constant:

newbase() => < [] >

Creates a new, modifiable and empty, base string.

With newbase() we can now redefine concatentation and the append operation:

s ˜ t == base(replace(end(replace(newbase(), s)), t)).

The result contains the concatentation of the marked segments from *s* and *t*.

v ˜:= t == v := base(replace(finish(base(v)), t)), (where v is a variable).

The value of variable *v* is appended with *t* and *v* is given the entire result as its new value.

With these definitions, one common coding sequence can be

```
t := newbase();
while . . . do {
        . . .
        t ˜:= expression;
        . . .
}
```

The variable *t* is initialized to a modifiable empty string and then text is appended to it within the loop.