Use of the Bird Meertens Formalism

Ankur Taly (03005017) Aditya Parameswaran (03005015) Ankit Jain (03005021)

Abstract

In this report, we deal with the paradigm of Constructive Algorithmics or the science of program transformation. We examine the basic ideas of Bird Meertens Formalism and its application to segment problems. We first give the direct application of Bird Meertens Formalism to the Maximum Segment Sum Problem, and also indicate the underlying concepts involved. We then proceed to give an intuitive proof for the Sliding Tails theorem, and demonstrate how it can be applied to a problem.

1 Introduction

Bird Meertens Formalism deals with the theory of proving program correctness by rewriting of a simple specification of the program using rules which are formally proved. We take a primitive specification of the problem (usually the definition or the simple brute force algorithm) and we repeatedly apply rules of the formalism, which have been proved previously, and transform this specification to arrive at a more efficient and refined algorithm via these rewrite rules. For this reason, Bird Meertens Formalism has also been called *Constructive Algorithmics* because it is used to construct algorithms from specifications. But this algorithm comes automatically proved. Due to the wide variety of programs and problems, the number of "Tricks of the trade" is rather large, and one may need a number of different program transformation rules to arrive at a proof of any given program.

Bird Meertens Formalism can be used for two purposes:

- To prove an already existing (and efficient) algorithm by starting from the specification and then trying to re-derive this algorithm
- To take a specification, and apply rules to it to just give a cleaner or a more efficient algorithm

The basic premise of the Bird Meertens Formalism is to provide a rewriting or a transformation of a program. This transformation is written as follows:

```
original specification

=  {justification for first step}

intermediate version

=  {justification for second step}

:
:
```

```
intermediate version

=  {justification for last step}

final version
```

with each of the justifications being formally proved by us. An important note on the use of the Bird Meertens Formalism: It is most useful when dealing with structured data types: for example, trees or lists. Use of Bird Meertens Formalism on arbitrary data can be a hard exercise.

2 Notation

Since the literature on Bird Meertens Formalism is fraught with complicated notation which we are not familiar with, we would like to propose our own notation which we have drawn on from the various sources that we have read.

2.1 Lists

We largely use the haskell notation for lists. A list containing elements $a_1, a_2, ..., a_n$ is denoted as $[a_1, a_2, ..., a_n]$. We have used 3 main ways of constructing lists :-

- cons operator $(:) :: (3 : 4 : 5 : [\,]) \Rightarrow [3, 4, 5]$
- append operator $(+\!\!+) :: [1, 2] +\!\!+ [3, 4] \Rightarrow [1, 2, 3, 4]$
- back-cons operator $(\prec) :: [3, 4] \prec 5 \Rightarrow [3, 4, 5]$

2.2 Operations On Lists

We now define the various operators used over lists in coming sections.

- Car : $car\ [a_1, a_2, ..., a_n] = a_1$
- Last: $last\ [a_1, a_2, ..., a_n] = a_n$
- Cdr : $cdr\ [a_1, a_2, ..., a_n] = [a_2, ..., a_n]$
- Map $(*)$: $f*[a_1, a_2, ..., a_n] = [f(a_1), f(a_2), ..., f(a_n)]$
- Reduce $(/)$: $\oplus/[a_1, a_2, ..., a_n] = a_1 \oplus a_2 \oplus ... \oplus a_n$ (for a associative predicate)
- Predicate Check $(\triangleleft)$: $p \triangleleft [a_1, a_2, ..., a_n] = [b_1, b_2, ..., b_m]$ where $\exists i, j : b_i = a_j$ and $\forall j\ p(b_j)$ is True, and the order of the b_i is the same as the order of the corresponding a_j.
- Fold left $(foldl)$: $foldl\ \otimes\ id\ [a_1, a_2, ..., a_n] = (...((id \otimes a_1) \otimes a_2)... \otimes a_n)$. This function folds a list from the left based on a function and an identity element.

3 Segments

A segment of a list x is a list consisting of a number of consecutive elements from x. Formally, list y is a segment of list l if and only if there exist lists x and z such that $l = x +\!\!+ y +\!\!+ z$. The function *segs* that returns a set of all segments of a list can be defined as :

$$segs\ l = \{y \mid \exists x, z : x +\!\!+ y +\!\!+ z = l\}$$

The function *segs* can be defined as segs : $[a] \Rightarrow [[a]]$

$$\begin{aligned} segs\ [\,] &= \{[\,]\} \\ segs(x \prec a) &= segs\ x \cup tails\ (x \prec a) \end{aligned}$$

where function *tails* is defined as tails : $[a] \Rightarrow [[a]]$

$$\begin{aligned} tails\ [\,] &= [\,] \\ tails(x \prec a) &= (\prec a) * tails\ x \cup \{[\,]\} \end{aligned}$$

Similarly function *inits* can be defined as inits : $[a] \Rightarrow [[a]]$

$$\begin{aligned} inits[\,] &= [\,] \\ inits(x \prec a) &= inits\ x \cup \{(x \prec a)\} \end{aligned}$$

So we have the following:

$$inits[a_1, a_2, ..., a_n] = [[a_1], [a_1, a_2], ...[a_1, a_2, ..., a_n]] \quad (1)$$

$$tails[a_1, a_2, ..., a_n] = [[a_1, a_2, ..., a_n], [a_2, ..., a_n]...[a_n]] \quad (2)$$

Function *segs* can also be expressed using only *inits* and *tails* as follows:

$$segs = flatten \circ tails* \circ inits \quad (3)$$

where *flatten* takes a list of lists and flattens them into a single list. flatten: $[[a]] \Rightarrow [a]$

$$\begin{aligned} flatten([\,]) &= [\,] \\ flatten(x \prec l) &= (flatten\ x) +\!\!+ l \end{aligned}$$

3.1 Segment Problems

The generic specification of **segment problems** is given by:

$$\oplus/ \circ f* \circ segs \quad (4)$$

for arbitrary symbol $\oplus$ and function f. Given a list $\mathcal{L}$ of length n, this function requires at least time $\Omega(n)$ to compute the value of the above equation, since segs returns a set of $\Omega(n^2)$.

But by using the techniques we illustrate in this paper and Equation 3, we can improve the complexity of this to $\mathcal{O}(n)$.

3.2 Prefix Closed Predicate

Predicate p is prefix-closed if it holds for $\mathcal{L}$ then it holds for every initial segment of $\mathcal{L}$. Formally, a predicate p is said to be **prefix-closed** if

$$\forall x,\ y: \quad p(x +\!\!+ y) \Rightarrow p(x)$$

It follows that a prefix-closed predicate must hold for the empty list if it hold at all. Thus, for simplicity we assume

$$\forall p : p[\,] = True$$

For Ex. If p is a predicate that checks whether the elements of a list $\mathcal{L}$ are in ascending order, than p is prefix-closed predicate.

3.3 Suffix Closed Predicate

On similar lines, predicate p is suffix-closed if it holds for $\mathcal{L}$ then it holds for every suffix segment of $\mathcal{L}$. Formally, a predicate p is said to be **suffix-closed** if

$$\forall x,\ y: \quad p(x +\!\!+ y) \Rightarrow p(y)$$

It follows that a suffix-closed predicate must hold for the empty list if it hold at all. Thus, for simplicity we assume

$$\forall p : p[\,] = True$$

For Ex. If p is a predicate that checks whether the elements of a list $\mathcal{L}$ are in ascending order, than p is suffix-closed predicate.

4 Maximum Segment Sum

We try to use the Bird Meertens Formalism for proof of the Maximum Segment Sum problem. This proof has already been done before [1], but we try to simplify.

4.1 The Problem

The Maximum Segment Sum problem takes a list and returns the contiguous segment of the list which has maximum sum. We assume that at least some elements are positive.

Here is an example: The list is $1, -2, 3, 4, -3, 5, -3, -5$. It is clear that the Maximum Segment Sum is the sum of $3, 4, -3, 5$, which is 9.

4.2 The Specification

Here is a simple and clean specification for the problem: *find the maximum of the sums of the segments of the list.* Equivalently, we could write this as: *find all segments of all lengths. find the sum of all of them. take its maximum.* This is clearly a $\mathcal{O}(n^3)$ algorithm (finding segments of all lengths is $\mathcal{O}(n^2)$ with the average length over which we need to sum up being $\mathcal{O}(n)$, giving rise to $\mathcal{O}(n^3)$). We would like to formulate a more efficient algorithm as described in the next few sections.

4.3 Horner's Rule

Here we describe a rule which we use in the derivation of the Maximum Segment Sum Problem.

$$(+/) \circ (\times/)* \circ (\text{tails } l) = foldl\ (\star)\ id\ l \tag{5}$$

where $id \star a = a$, and e is the identity element of $\times$, and

$$(a + b) \times c = (a \times c) + (b \times c) \tag{6}$$
$$a \star b = (a + e) \times b \tag{7}$$

We have appropriately overloaded the operators ($+$ and $\times$) to make it more realistic (and give a better feel for the operators). The intuition behind this is clear. When we have the following:

$$(a_1 \times a_2 \times ... \times a_n) + (a_2 \times ... \times a_n) + ... + (a_n)$$

we can clearly reduce the computation by using a number of identity elements e as follows by considering the redundancy each of these elements:

$$(...((((a_1 + e) \times a_2) + e) \times a_3) + ... + e) \times a_n$$

which we are able to do precisely due to the distributivity equation above. This can be further reduces to

$$(...(((a_1 \star a_2) \star a_3) \star a_4)... \star a_{n-1}) \star a_n$$

which we can simply write as a foldl on $\star$ as:

$$foldl\ (\star)\ id\ [a_1, a_2, ..., a_n]$$

if id satisfies the property above.

4.4 Map Promotion

Here is a simple enough rule dealing with flatten. The application of f to each element of the flattened list is equivalent to the application of $f*$ to each element of the original list, which is a list of lists, and then taking its flatten. This gives us

$$(f*) \circ \texttt{ flatten} = \texttt{flatten} \circ (f**) \tag{8}$$

4.5 Catamorphism Promotion

If $h(a \oplus b) = h(a) \otimes h(b)$, then by induction on the length of the list $\mathcal{L}$, we can see that

$$h(\oplus/\ \mathcal{L}) = \otimes/(h*\ \mathcal{L}) \tag{9}$$

4.6 The Better Algorithm

A simple $\mathcal{O}(n)$ algorithm goes as follows: It maintains two pointers, a current head pointer and a tail pointer. As long as the sum of the list keep increasing, we add the last element to the list. As soon as the sum becomes negative, we discard the list. We also keep a tab on the maximum length list seen so far. Thus effectively, we just move over each element in the list a maximum of two times using both the pointers.

The catamorphism is defined as a reduce over a map. We use it primarily to give us a characterization of the problem in a linear manner.

4.7 The Use of Bird Meertens Formalism for the MSS problem

Here is the specification for the MSS problem:

$$max \circ sum* \circ segs$$

which rephrases the fact that the maximum segment sum is obtained by taking the maximum of the sum of the segments. Now we use a number of simplifications on this. The first of these is to express `segs` as composed of union of the tails of all the initial segments of the list. This gives:

$$max \circ sum* \circ flatten \circ tails* \circ inits$$

On applying Map Promotion on sum and flatten, we get

$$max \circ \ flatten \circ sum** \circ tails* \circ inits$$

Then, we realize that using the catamorphism promotion theorem on max and flatten (which is merely ++ as a reduce on the list), which give us the fact that max can be used over flatten to give another map application of max, we get

$$max \circ \ max* \circ sum** \circ tails* \circ inits$$

Then, the use of the distributivity of map over composition of functions, implies that $f* \circ g* = (f \circ g)*$, and this allows us the simplify the above to:

$$max \circ (max \circ sum* \circ tails)* \circ inits$$

We then apply our last transformation, that of the Horner's rule as follows: the part $max \circ sum* \circ tails$ is clearly of the form of Horner's rule, with sum distributing over max, with $max(a,b)+c = max(a+c, b+c)$, as required in the Horner's Rule Equation number 6. We define $\star$ to be $(max(a,0)+b)$, to give us:

$$max \circ (foldl \ \star id)* \circ inits$$

Now consider the following: if we have a list of elements

$$(id \star a_1, (id \star a_1) \star a_2, ..., (..((id \star a_1) \star a_2)... \star a_n))$$

then this can be written as a foldl such that the current evaluation of the list is taken, the last element is taken of that evaluation, a $\star$ is applied with it and a_i to create the new element of the list.

This gives us a linear characterization of the part of the above expression $(foldl \star id)* \circ inits$. such that

$$max \circ (foldl \ \oplus id)$$

where $a \oplus b = a +\!\!+ ((last\ a) \star b)$.

Thus, we have a simple characterization of the MSS problem as a linear search on the list. This directly corresponds to the description of the algorithm as given above, because this description finds the maximum of the lists by effectively moving two pointers ahead. The first pointer movement is implicit in each element of the list, and the second pointer movement is shown by each element of the above list (the second pointer is at element $1, 2, ..., n$). Between each element of the list, the first pointer moves to give the new location of the first pointer.

Thus we have proved the order $\mathcal{O}(n)$ algorithm that was described above purely using the rules and formalisms of program transformation.

5 Sliding Tails theorem (STT)

Theorem *Let p be a prefix closed predicate and $\mathcal{L}$ be a list then we have*

$$(\uparrow/) \circ (p \lhd) \circ \mathit{tails}\ \mathcal{L} = \mathit{foldl}\ \oplus_p\ [\,]\ \mathcal{L}$$

$$(\uparrow/) \circ (p \lhd) \circ\ \mathit{segs}\ \mathcal{L} = \mathit{first} \circ (\mathit{foldl}\ \otimes_p\ ([\,], [\,])\ \mathcal{L})$$

where operator $\oplus_p$ is defined as

$$x \oplus_p a \quad = \quad \begin{cases} x \prec a & \text{if } p(x \prec a) \\ (\mathit{cdr}\ x) \oplus_p a & \text{if } \neg p(x \prec a) \wedge (\mathrm{x} \neq [\,]) \\ [\,] & \text{otherwise} \end{cases}$$

where operator $\otimes_p$ is defined as

$$(x, y) \otimes_p a = (x \uparrow (y \oplus_p a), (y \oplus_p a))$$

Here $\uparrow$ is an associative operator such that for two lists $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \uparrow \mathcal{B}$ returns the list which is longer.

5.1 Outline of the Proof

5.1.1 Part a

First let us examine the operator $\oplus_p$. $x \oplus_p t$ basically returns the longest tail of $(x \prec t)$ which satisfies the predicate p. The longest tail is found by successively taking *cdr* of the list $(x \prec t)$ and then checking whether it satisfies the predicate p. So we observe that since p is prefix closed we find that the longest tail satisfying p of $x \prec t$ cannot be longer than the t attached at the end of the longest tail satisfying p for x. If this was not the case, then we would a longer tail for x which satisfies p (due to the prefix-closed property). Hence finding the longest tail of $x \prec t$ is the same as finding the longest tail of $\mathcal{M} \prec t$, where $\mathcal{M}$ is the longest tail of x satisfying p. Formally, we can view this as:

$$x \oplus_p t = (\uparrow/(p \lhd (\mathit{tails}\ x))) \oplus_p t \tag{10}$$

Now we prove the sliding tails theorem by induction over the size of this list.

Base case: For the case $\mathcal{L} = [\,]$, the theorem is trivially true.
Hypothesis: Assume the theorem for lists of length k. So for a list $\mathcal{L}_k$ of length k we have

$$\uparrow/\ \circ\ p \lhd \circ\ \mathit{tails}\ \mathcal{L}_k = \mathit{foldl}\ \oplus_p\ [\,]\ \mathcal{L}_k \tag{11}$$

Consider the list of size $k+1$, $\mathcal{L}_{k+1} = \mathcal{L}_k \prec t$ for some element t. We need to prove the theorem for this list. We use the lemma given in the next section for this purpose.

5.1.2 Lemma 1

$$\uparrow/ \circ\ p \lhd \circ\ \mathit{tails}\ (\mathcal{L} \prec t) \quad = \quad \begin{cases} \mathcal{M} \prec t & \text{if } p(\mathcal{M} \prec t) \\ \uparrow/ \circ\ p \lhd \circ\ \mathit{tails}\ ((\mathrm{cdr}\ \mathcal{M}) \prec t) & \text{if } \neg p(\mathcal{M} \prec t) \wedge (\mathcal{M} \neq [\,]) \\ [\,] & \text{otherwise} \end{cases}$$

where $\mathcal{M} = \uparrow/ \circ\ p \lhd \circ\ \mathit{tails}\ (\mathcal{L})$.

(We give the proof of this later) Here we are finding the longest tail of $L \prec t$ which satisfies p. But as we saw above $L \oplus_p t$ also finds the longest tail of $L \prec t$ which satisfies the predicate p. Therefore

$$\uparrow/ \ \circ \ p \triangleleft \ \circ \ tails \ (\mathcal{L} \prec t) = \mathcal{L} \oplus_p t \tag{12}$$

Now by using equation 10, we extend Lemma 1 to

$$\uparrow/ \ \circ \ p \triangleleft \ \circ \ tails \ (\mathcal{L} \prec t) = (\uparrow/ \circ p \triangleleft (tails \ \mathcal{L})) \oplus_p t \tag{13}$$

Now for the list $\mathcal{L}_{k+1} = \mathcal{L}_k \prec t$ we have the theorem as

$$LHS = \uparrow/ \ \circ \ p \triangleleft \ \circ \ tails \ (\mathcal{L}_k \prec t)$$

Thus using 13 we would have

$$LHS = (\uparrow/p \triangleleft (tails \ L_k)) \oplus_p t$$

Next using the hypothesis we get

$$LHS = (foldl \ \oplus_p \ [\,] \ L_k) \oplus_p t$$

Therefore,

$$LHS = (foldl \ \oplus_p \ [\,] \ (L_k \prec t)) = RHS$$

Hence proved. □

5.1.3 Part b

The second part of the theorem is very similar to the first part. Here the only difference is that we have a pair of lists that is formed on folding using $\otimes_p$. The $\otimes_p$ basically folds the list from the left using the operator $\oplus_p$ and stores the value in the second element of the pair. In the first element we keep track of the current maximum segment which satisfies the predicate p. Each time we fold, the second element of the pair gives us the longest tail that satisfies p. We compare this with the first element and update the maximum appropriately. Hence finally after folding the entire list the first element of the pair obtained gives the longest segment that satisfies the predicate p. □

5.1.4 Proof of Lemma 1

An informal proof of Lemma 1 would be as follows. Let $\mathcal{L} = [x_1, x_2, .., x_n]$ be the list. Now we have to find the longest tail of the list $\mathcal{L}' = [x_1, x_2, .., x_n, t]$ which satisfies the predicate p. Suppose the longest tail of $\mathcal{L}$ which satisfies p is $[x_i, x_{i+1}, ..., x_n]$. Therefore

$$\mathcal{M} = \uparrow/ \circ \ p \triangleleft \circ \ tails \ (\mathcal{L}) \ = \ (x_i, x_{i+1}, ..., x_n)$$

Since p is prefix closed if $[x_j, x_{j+1}, ..., x_n, t]$ is the longest tail of $\mathcal{L}'$ satisfying p then $j \geq i$. Thus the longest tail would be $\mathcal{M} \prec t$ if $p(\mathcal{M} \prec t)$ is true. Otherwise we try to attach t to the list $(cdr \ \mathcal{M})$. Again we find out the longest tail of $(cdr \ \mathcal{M}) \prec t$ satisfying p and repeat the above procedure(If $(cdr \ \mathcal{M})$ does not exist then we return $[\,]$).

5.2 Complexity of Sliding Tails Theorem

Let the time for the computation of the truth value of $p(\mathcal{L})$ be $O(f(n))$ where n is the size of the list $\mathcal{L}$. Then the time for the computation of $(foldl \otimes_p ([\,], [\,])\ \mathcal{L})$ and hence for $(\uparrow/)\circ(p \triangleleft)\circ\ segs\ \mathcal{L}$ would be $\mathcal{O}(n.f(n))$.

However the complexity might become better if the predicate p was also suffix-closed. In this case we observe that each time when the operator $\otimes_p$ is computed during the iterations of *foldl*, the first argument of $\otimes_p$ would always satisfy the predicate p. Now if we can compute the truth value of $p(x \prec t)$, given that $p(x)$ is True, in $\mathcal{O}(1)$ time then the time for the computation of $(\uparrow/) \circ (p \triangleleft) \circ$ segs $\mathcal{L}$ would be $\mathcal{O}(n)$. This is because of the fact that p is suffix closed which implies that if $p(x)$ is true, for any suffix y, $p(y)$is true as well. This means that $p(cdr\ x)$ is true given $p(x)$ is true, and since the computation of $p(cdr\ x) \prec a$ can be done in $\mathcal{O}(1)$ time given the fact that $p(cdr\ x)$ is true, we have the following: At each step we move one of the pointers (either the pointer at the head of x, or the pointer at the additional element a), and compute p of the new list in $\mathcal{O}(1)$ time, thereby giving a complexity of $\mathcal{O}(n)$.

Thus, we have a general $\mathcal{O}(n)$ procedure to find the longest-p-segment where p satisfies the following properties:

- p is prefix closed
- p is suffix closed
- given that $p(x)$ is true, $p(x \prec a)$ can be computed in $\mathcal{O}(1)$ time

6 Longest Ascending Contiguous Subsequence problem

6.1 The Longest Ascending Subsequence problem

Consider a sequence $S\ :\ x_1, x_2, ..., x_n$. A contiguous ascending subsequence of this sequence is a sequence $x_i, x_{i+1}, ..., x_j$ such that $i, j \in \{1, 2, ..., n\}$ and $x_i < x_{i+1} < ... < x_j$. Thus the problem is to find out the longest ascending contiguous subsequence (LAS) of a given sequence $\mathcal{L}$.

6.2 Existing Algorithm

The simple brute force algorithm for finding the LAS is fairly straightforward, it can be expressed as follows:

$$\uparrow/ \circ (asc \triangleleft) \circ segs$$

Now we analyze the complexity of this algorithm. The number of *segs* is $\mathcal{O}(n^2)$, and for each of those, we need to find if the predicate *asc* satisfies it, and then check if it is the *max*. Predicate *asc* check takes $\mathcal{O}(n)$, due to which the combined complexity of the algorithm is $\mathcal{O}(n^3)$

6.3 Solution using Sliding Tails theorem

We model the LAS problem using sliding tails theorem. Let the sequence be given by a list $\mathcal{L}$ and let the function *las* give the longest continuous ascending subsequence of $\mathcal{L}$. Then we have

$$\begin{aligned} las\ \mathcal{L} &= \uparrow/\ asc \triangleleft (segs\ \mathcal{L}) \\ &= first\ \circ\ (foldl \otimes_{asc}\ ([\,],[\,]))\ \mathcal{L} \end{aligned}$$

Here *asc* is a prefix closed as well as suffix closed predicate which acts on lists. For any list $\mathcal{P} = [p_1, p_2, ..., p_k]$,

$$asc(\mathcal{P}) = \text{True iff } p_1 < p_2 ... < p_n$$

6.4 Complexity of computation of solving the LAS problem using STT

Here the predicate *asc* is both prefix and suffix closed. Moreover given that $asc(x)$ is true we can compute $asc(x \prec t)$ in $\mathcal{O}(1)$ time. Hence the function *las* can be computed in the $\mathcal{O}(n)$ time.

6.5 Example

We apply the function las to the list (1,2,7,3,4,5,6,2,9).The various iterations obtained on applying $foldl$ to this list with operator $\otimes_{asc}$ and id ([ ] , [ ]) are as follows

Iteration	Intermediate value	Next element in the list
1	$([\,],[\,])$	1
2	$([1],[1])$	2
3	$([1,2],[1,2])$	7
4	$([1,2,7],[1,2,7])$	3
5	$([1,2,7],[3])$	4
6	$([1,2,7],[3,4])$	5
7	$([1,2,7],[3,4,5])$	6
8	$([3,4,5,6],[3,4,5,6])$	2
9	$([3,4,5,6],[2])$	9
10	$([3,4,5,6],[2,9])$	

Therefore, $first\,([3,4,5,6],[2,9]) = [3,4,5,6]$. Thus the function $las = (first \circ foldl \otimes_{asc} ([\,],[\,]))$ correctly returns the longest contiguous ascending subsequence which is [3,4,5,6].

7 Conclusion

This report tries to present a insight into the science of Constructive Algorithmics, through that of Bird Meertens Formalism. We take only problems dealing with segments, i.e. continuous subsets of lists. We study the Maximum Subset Sum problem through the Gibbons [1] Approach, and the associated theory that goes with the Maximum Subset Sum Problem (for example, that of Horners rule).

We then give an intuitive derivation for Sliding Tails theorem that we came up with. We use this to prove the efficient $\mathcal{O}(n)$ algorithm for longest contiguous subsequence and indicate how this Sliding Tails Theorem may be used to prove or derive other similar algorithms for predicates that are prefix and/or suffix closed. We also indicate the improvements brought about by the extra knowledge of suffix closed property of the predicate, and also show the utility of the suffix property of the predicate. (i.e. that if $p(\mathcal{L})$ holds, then it is easy ($\mathcal{O}(1)$) to check that $p(\mathcal{L} \prec a)$ holds or not)

Our contention is that mere intuitive reasoning backed with formal proof techniques is sufficient for a complete understanding of the application of Bird Meertens Formalism to Segment Problems. The Sliding Tails Theorem and Horners Rule are of considerable importance in Bird Meertens Formalism for segment problems, and this report indicates the utility of both of them via examples.

Bird Meertens Formalism, while it has a sound theoretic foundation, still lacks application in real world problems due to the nature of human involvement and intuition, which is rather large. Further work, if any needs to be done in developing some concrete mechanized procedures for the same.

References

[1] Gibbons J. : *Introduction to the Bird Meertens Formalism*

[2] Bird R., Gibbons J., Jones G. : *Formal Derivation of a Pattern Matching Algorithm*

[3] Jeuring J. : *The Derivation of On-line Algorithms, with an Application to Finding Palindromes*