Algorithmica (1994) 11: 146–184

Algorithmica

The Derivation of On-Line Algorithms, with an Application To Finding Palindromes[1]

Johan Jeuring[2]

Abstract. A theory for the derivation of on-line algorithms is presented. The algorithms are derived in the Bird–Meertens calculus for program transformations. This calculus provides a concise functional notation for algorithms, and a few powerful theorems for proving equalities of functions. The theory for the derivation of on-line algorithms is illustrated with the derivation of an algorithm for finding palindromes.

An on-line linear-time random access machine (RAM) algorithm for finding the longest palindromic substring in a string is derived. For the purpose of finding the longest palindromic substring, all maximal palindromic substrings are computed. The list of maximal palindromes obtained in the computation of the longest palindrome can be used for other purposes such as finding the largest palindromic rectangle in a matrix and finding the shortest partition of a string into palindromes.

Key Words. Derivation of on-line algorithms, Transformational programming, Bird–Meertens calculus, Segment problems, Theory of lists, Longest palindromic substring, Maximal palindromes.

1. Introduction. In this paper we present a theory for the derivation of on-line algorithms. The algorithms are derived in the Bird–Meertens calculus for program transformation. This calculus provides a concise functional notation for algorithms, and a few powerful theorems for proving equalities of functions. The theorems are used to transform inefficient but clearly correct specifications (which are functional algorithms themselves) into efficient algorithms. Thus, it is necessary to state (and prove) explicitly all the equalities used in the derivation of an algorithm. Most of the functions we use are well known in functional programming, see [5]. Aspects of the Bird–Meertens calculus are described in [23], [2], [3], and [24].

For several classes of problems theories can be developed, such as a theory for problems on lists, see [2] and [18], and a theory for problems on matrices, see [3] and [19]. In this paper we further develop the theory of lists, in particular the theory of segments (in the literature, a segment is also called a substring or a factor). We prove a number of theorems with which many on-line algorithms for segment problems can be derived.

A left-reduction is a function defined on the data type list whose inductive definitional pattern mimics that of the type. We argue that the recursive structure

[1] This research was supported by the Dutch organization for scientific research NWO, under NFI project STOP, project number NF 62/63-518.

[2] Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands. johan@cs.ruu.nl.

Received September 10, 1990; revised August 7, 1991. Communicated by C. K. Wong.

of an on-line algorithm follows that of a left-reduction. In the derivations we present we strive to derive a left-reduction (or a slight generalization of it) for the given specifications. Therefore, the theory we present may be viewed as a theory for the derivation of on-line algorithms.

The theory for on-line algorithms is illustrated with the derivation of an on-line linear-time random access machine (RAM) algorithm for finding the longest palindromic substring in a string. One of the most important theorems used in this derivation is a generalization of a theorem used by Bird *et al.* [4] to derive the algorithm for pattern matching from Knuth, Morris, and Pratt.

An occurrence of a palindrome in a string is called extendible if it is preceded and followed by equal characters, otherwise it is called maximal (including the case when there is no element preceding or following it). For example, in "colon" the substring "l" is an extendible palindrome and the substrings "olo" and "c" are maximal palindromes.

We derive an on-line linear-time RAM algorithm for finding the longest palindromic substring in a string. For the purpose of finding the longest palindromic substring of a string we compute (the positions of) all maximal palindromes occurring in a string. The algorithm for finding (the positions of) all maximal palindromes occurring in a string is used to solve some other problems concerning palindromes, such as the problem of finding the largest palindromic rectangle in a matrix, and the problem of finding the shortest partition into palindromes of a string.

The derivation of the algorithm as presented in this article is a new derivation of the algorithm given in [17]. After publishing [17], Jeuring found that the notion of maximality of palindromes and the algorithm to compute all maximal palindromes occurring in a string had been given before, by Galil and Seiferas [14]. In that article, maximal palindromes are used in an algorithm for recognizing the language $P*$, where P is the set of palindromes. Instead of giving a difficult correctness proof of the algorithm, we derive the algorithm from its specification, using a theory for the derivation of on-line algorithms.

Palindromes have been studied extensively in algorithm and complexity theory. Efficient algorithms for recognizing P, the set of palindromes, and similar problems have been constructed on several computing models. Cole [6] gives a real-time algorithm for recognizing P on an iterative array of finite-state machines. Seiferas [26] shows how to recognize palindromes of even length on a computing model similar to the iterative array. Algorithms for recognizing P on several different Turing-machine models are given in [16]. A lower bound on the complexity of recognizing palindromes on probabilistic Turing machines is derived by Yao [29].

Recognizing initial palindromes in a string was the next problem related to palindromes to be addressed in several papers. Manacher [22] gives a linear-time algorithm on the RAM computing model finding the smallest initial palindrome of even length. He also describes how to adjust his algorithm in order to find the smallest initial palindrome of odd length ≥ 3. Manacher's algorithm is on line, that is, it is linear time, but in between reading two symbols from the input string more than constant time may be spent. The algorithm constructed by Manacher is obtained by Galil [11] using several theoretical results on fast simulations. Using

their algorithm for pattern matching, Knuth *et al.* [20] give an off-line linear-time RAM algorithm for finding the longest initial palindrome of even length. The ideas of Manacher are generalized in [14] and [17] to find (the positions of) all maximal palindromes in a string on line on the RAM computing model. Crochemore and Rytter [7] give a parallel version of this algorithm.

The papers mentioned above contain RAM algorithms for recognizing palindromes. Algorithms on Turing machines have been devised too. Fischer and Paterson [8] give a linear-time off-line Turing-machine algorithm for finding all initial palindromes in a string. Finally, Galil [10], [12] describes a real-time (that is, on line, but in between reading two symbols from the input string constant time is spent) multitape Turing-machine algorithm finding all initial palindromes in a string. Galil's algorithm improves on Slisenko's work [27] on algorithms for finding palindromes.

This paper is organized as follows. Section 2 introduces the notation and specific functions used in the developments in the subsequent sections. Furthermore, some basic fusion theorems are given, such as the fusion theorem for data types in the Boom hierarchy and the Snoc-Lists Fusion Theorem. Section 3 defines the notion of segment, and proves some general theorems concerning segment problems. Furthermore, we derive the generalization of the theorem used to derive the pattern-matching algorithm from Knuth, Morris, and Pratt in [4]. Section 4 gives the algorithm for finding palindromes, together with a discussion on its complexity. Section 5 uses the algorithm for finding the longest palindromic segment and the positions of all maximal palindromes occurring in a string derived in Section 4 to solve some other problems concerning palindromes, such as finding the shortest partition into palindromes of a string. Finally, Section 6 contains some conclusions.

2. Basics of the Bird–Meertens Calculus. In this section we introduce the basic notions and definitions used in the subsequent sections. In the first subsection we briefly describe functions, operators, and cartesian products. Two important concepts in the Bird–Meertens calculus are the notions of catamorphism and fusion. For every data type, catamorphisms are defined and a fusion theorem is given. This process is described in detail by Malcolm [21] and Fokkinga [9]. Section 2.2 defines catamorphisms and gives the fusion theorem for the data types in the Boom-hierarchy, such as set and list. Section 2.3 discusses the data type snoc-list. We introduce left-reductions (catamorphisms on snoc-lists), and we give the Snoc-List Fusion Theorem. Finally, Section 2.4 introduces some auxiliary functions and operators.

2.1. Functions, Operators, and Products. A *function* is an object with three components written $f : s \to t$, where s is a set called the *source* of the function, t is a set called the *target* of the function, and f maps each member x of s to a member of t. This member is denoted $f\ x$, using simple juxtaposition and a little white space to denote application of a function f to an argument x. We use the letters f, g, h, etc., as variables standing for arbitrary functions. Function application is

right-associative, i.e., we have

$$f\ (g\ (h\ x)) = f\ g\ h\ x.$$

The *composition* of two functions $f: s \to t$ and $g: r \to s$ is written $f \cdot g: r \to t$. Composition is associative, that is, for all f, g, and h we have $f \cdot (g \cdot h) = (f \cdot g) \cdot h$. Taking advantage of associativity, chains of compositions are usually written without brackets. An example of a function is the identity function: for each set s, the function $id_s: s \to s$ is the identity function on the set s.

We postulate the existence of the cartesian product. If A and B are types, then their *cartesian product*, $A \times B$, is a type whose objects are all pairs (a, b), where $a \in A$ and $b \in B$. A function with a cartesian product as its source is referred to as a binary operator. The operator $+: nats \times nats \to nats$ is such a function. Typical variable names for binary operators are $\oplus$, $\otimes$, $\odot$, etc. We frequently use infix notation for the application of a binary operator $\oplus$ to an argument (a, b), so instead of writing $\oplus(a, b)$ we write $a \oplus b$. Here and throughout, we adopt the convention that function application is more binding than infix binary application, so the expression $f\ a \oplus g\ b$ should be parsed as $(f\ a) \oplus (g\ b)$. Binary operators can be parametrized, i.e., if $\oplus$ is a binary operator of type $A \times B \to C$, and $a \in A$, we consider the expression $(a\oplus)$ to be a unary function of type $B \to C$. It is defined by

$$(a\oplus)\ b = a \oplus b.$$

The function $(\oplus b)$ is defined similarly. These parametrized operators are also known as "sections." For example, the function $(+3): nats \to nats$ takes a natural number as argument and returns the number increased by three, so $(+3)\ 5 = 8$. The cartesian product type has two primitive operations, the projections $\lll: A \times B \to A$ ("first") and $\ggg: A \times B \to B$ ("second"). The projection operators are defined by

$$a \lll b = a,$$

$$a \ggg b = b.$$

The projection operators are the first examples of polymorphic functions, in the sense that $\lll: A \times B \to A$ for all types A and B. Many more polymorphic functions are encountered in what follows. If operator $\oplus$ has a unit or identity element, then it is written $v_\oplus$. An identity element is also called a neutral element. The element $v_\oplus$ satisfies by definition $v_\oplus \oplus a = a \oplus v_\oplus = a$ for all a. If a unit element exists, it is unique. If a unit element does not exist we may introduce a "fictitious" unit element, also written $v_\oplus$, with the same property. Caution is advised though in doing this. The cases in which we introduce fictitious elements cause no problems. The operator $\times$ is defined on types as well as functions. For functions $f: A \to B$ and $g: C \to D$, function $f \times g: A \times C \to B \times D$ is defined for all pairs $(a, c) \in A \times C$ by

$$(f \times g)\ (a, c) = (f\ a, g\ c).$$

Operator $\times$ binds stronger than composition. We have the following law for operator $\times$:

$$f \times g \cdot h \times j = (f \cdot h) \times (g \cdot j). \tag{1}$$

This law may be proved by taking an arbitrary element (a, b) of the type $A \times B$, where A is the source type of h and B is the source type of j, and showing that the results of both sides applied to (a, b) are equal.

If we swap the components of the type of binary operators we obtain the type $C \to A \times B$. Functions of this type can be constructed as follows. Let $f : C \to A$ and $g : C \to B$ be functions, then the function $f \triangle g : C \to A \times B$ (pronounced "f split g"), is defined by

$$(f \triangle g)\, c = (f\, c, g\, c).$$

Just like operator $\times$, operator $\triangle$ binds stronger than composition. The operators $\times$ and $\triangle$, and the projection functions satisfy the following laws. The proofs of these laws are similar to the proof of law (1) and are therefore omitted.

$$f \times g = (f \cdot \ll) \triangle (g \cdot \gg), \tag{2}$$

$$f \times g \cdot h \triangle j = (f \cdot h) \triangle (g \cdot j), \tag{3}$$

$$(f \cdot h) \triangle (g \cdot h) = f \triangle g \cdot h, \tag{4}$$

$$\ll \cdot f \times g = f \cdot \ll, \tag{5}$$

$$\gg \cdot f \times g = g \cdot \gg, \tag{6}$$

$$\ll \cdot f \triangle g = f, \tag{7}$$

$$\gg \cdot f \triangle g = g, \tag{8}$$

$$\ll \triangle \gg = id. \tag{9}$$

Finite cartesian products are the sets of all pairs, all triples, all quadruples, all quintuples, etc., together with projection functions π_1, π_2, etc. Usually, the projection functions are superscripted with the type on which they are defined, but we suppose that this information is deducible from the context, and therefore it is suppressed. So π_2 might be both of π_2^3 and π_2^4. Operator $\propto$ takes a pair and an element and returns a triple consisting of the three arguments.

$$(x, y) \propto z = (x, y, z). \tag{10}$$

The function $\natural$ takes the first two elements of a triple,

$$\natural(x, y, z) = (x, y), \tag{11}$$

so $\natural \cdot (\propto z) = id$ for all z.

A *relation* is a set of pairs. Let R be a relation. We write $x\,R\,y$ for $(x, y) \in R$. Statements of the form $(f\,x)\,R\,(f\,y)$ occur very often. We introduce a shorthand for these expressions:

$$x\,R_f\,y \equiv (f\,x)\,R\,(f\,y). \tag{12}$$

For example, for $24 \bmod 10 = 34 \bmod 10$ we may write $34 =_{\text{mod } 10} 24$.

2.2. The Boom-Hierarchy. Important data types for lots of problems for which we want to construct algorithms are the data types binary tree, list, bag, and set. These four data types form a nice hierarchy. Meertens [23] attributes this hierarchy to H. J. Boom, and therefore this hierarchy is called the Boom-hierarchy. A data type is considered to be an initial algebra in a category of functor-algebras, see [21] and [9]. We describe data types informally.

The recursive data type *binary tree* over some base type A consist of elements that are constructed as follows. A binary tree is either the empty binary tree $\langle\,\rangle$, or it is a singleton tree $\tau\,a$, where a is an element of type A, or it is the concatenation of two binary trees $x \perp y$. The empty binary tree $\langle\,\rangle$ is the unit of $\perp$.

The other data types from the Boom-hierarchy are obtained by imposing laws upon the constructor $\perp$. If $\perp$ is associative we obtain the data type *join-list* ($\langle\,\rangle$ and $\perp$ are then written as respectively $\square$ and $\#$). If $\perp$ is associative and commutative we obtain the data type *bag*, also known as *multiset*, and if $\perp$ is associative, commutative, and idempotent we obtain the data type set ($\langle\,\rangle$ and $\perp$ are then written as respectively $\{\,\}$ and $\cup$). In the remaining part of this section we give definitions and prove theorems for the data type *set*. For each of the three other data structures in the Boom-hierarchy similar definitions and theorems can be given. All data types in the Boom-hierarchy are denoted by $A*$. Unless otherwise stated, the data type $A*$ is considered to be the data type *set* over base type A in what follows.

Given a data type, catamorphisms on this data type can be described systematically. A catamorphism is a homomorphism of which the source type is a data type. The data type *set* is defined such that given an associative, commutative, and idempotent operator $\oplus : B \times B \to B$, a function $f : A \to B$, and a value $e : B$ that is the unit of $\oplus$ there exists a unique function $h : A* \to B$ satisfying

$$\begin{aligned} h\{\,\} &= e, \\ h\,\tau\,a &= f\,a, \\ h\,(x \cup y) &= (h\,x) \oplus (h\,y). \end{aligned} \tag{13}$$

Function h is called a *catamorphism*. If a unit element does not exist, we may introduce a fictitious element (see [23]) with the property that it is the unit of $\oplus$. To define the notion of catamorphism on the data type *join-list*, the above definition should be repeated without the occurrences of the words idempotent and commutative.

A catamorphism defined on the data type *set* can be written as the composition

of a reduction and a map, which are defined as follows. The *map operator* $*$ takes as arguments a function and a set and returns a set consisting of the images of f of the original elements. More precisely, if $f : A \to B$, then $f* : A* \to B*$ is defined as the following catamorphism.

$$\begin{aligned} f*\{\ \} &= \{\ \}, \\ f*\,\tau\, a &= \{f\ a\}, \\ f*\,(x \cup y) &= (f*\ x) \cup (f*\ y). \end{aligned} \tag{14}$$

The result of applying the *reduction operator* $/$ to an associative, commutative, and idempotent operator $\oplus$ and a set can be obtained by placing $\oplus$ between adjacent elements of the set, so, if $\oplus : A \times A \to A$, then $\oplus/ : A* \to A$ is defined as a catamorphism as follows:

$$\begin{aligned} \oplus/\{\ \} &= v_\oplus, \\ \oplus/\,\tau\, a &= a, \\ \oplus/\,(x \cup y) &= (\oplus/\ x) \oplus (\oplus/\ y), \end{aligned} \tag{15}$$

where $v_\oplus$ is the, possibly fictitious, unit element of $\oplus$.

In general catamorphism h satisfies

$$h = \oplus/ \cdot f*$$

for some operator $\oplus$ and function f, a fact expressed by the Catamorphism Lemma from Meertens [23].

An example of a widely used catamorphism is the length function $\# : A* \to nats$ defined on the data type *join-list*. Define function $1^\bullet : A \to nats$ by $1^\bullet\, a = 1$ for all a. Since the addition operator $+$ on natural numbers is associative, the function $\# = +/ \cdot 1^\bullet*$ is a well-defined catamorphism on the data type *join-list*.

We now come to the second important notion of the Bird–Meertens calculus, *fusion*. Every data type has its own fusion theorem. Fusion provides a means for proving equalities of functions avoiding the application of induction in the development of algorithms. Inductive arguments tend to be tedious and are less elegant than proofs using fusion. As early as 1975, this was one of the main motivations of Goguen [15] to advocate the use of initiality. Before we give the theorem, we first define fusability.

(16) DEFINITION ($(\oplus, \otimes)$-fusability). Let $\oplus : A \times A \to A$ and $\otimes : B \times B \to B$ be associative, commutative, and idempotent operators. A function $f : A \to B$ is $(\oplus, \otimes)$-fusable if

$$\begin{aligned} f\,(x \oplus y) &= (f\ x) \otimes (f\ y), \\ f\ v_\oplus &= v_\otimes, \end{aligned}$$

where x and y range over the image of $\oplus/$, that is, x and y are of the form $\oplus/\, v$, $\oplus/\, w$ for some v and w.

We have the following theorem, a proof of which uses the fact that catamorphisms are unique functions satisfying (13); it can be found in [23] and [21].

(17) THEOREM (Fusion). *A function $f: A \to B$ is $(\oplus, \otimes)$-fusable if and only if $f \cdot \oplus/ = \otimes/ \cdot f*$.*

Consider the expression $f \cdot \oplus/ \cdot g$, where g is a function with target $A*$. To apply Fusion to obtain $\otimes/ \cdot f* \cdot g$, it suffices to show that f is $(\oplus, \otimes)$-fusable for elements of the form $\oplus/\, v$, $\oplus/\, w$ where we may assume in addition that v and w are subsets of $g\, z$ for some z.

Another theorem that we use frequently in the derivations in the following sections is the following.

(18) THEOREM (Map Distributivity). *For all functions $f: B \to C$ and $g: A \to B$ we have*

$$f* \cdot g* = (f \cdot g)*.$$

The *filter operator* $\lhd$, takes a predicate (i.e., a boolean function) and a set and retains the elements satisfying the predicate in a set, so if $p: A \to bool$, then $p\lhd: A* \to A*$ is defined by

$$p\lhd = \cup/ \cdot \hat{p}*, \tag{19}$$

where $\hat{p}\, a = \tau\, a$ if $p\, a$ holds and $\hat{p}\, a = \{\ \}$ otherwise. For example, $odd\lhd \{3, 4, 5\} = \{3, 5\}$. An expression of the form $p?_\omega$, called a *guard*, is defined by $p?_\omega = \otimes/ \cdot \hat{p}$, where $\omega = v_\otimes$. We have

$$\begin{aligned}
& \otimes/ \cdot p\lhd \\
= & \quad \text{definition of filter (19)} \\
& \otimes/ \cdot \cup/ \cdot \hat{p}* \\
= & \quad \text{Fusion, } \otimes/ \text{ is } (\cup, \otimes)\text{-fusable} \\
& \otimes/ \cdot \otimes/* \cdot \hat{p}* \\
= & \quad \text{map distributivity} \\
& \otimes/ \cdot (\otimes/ \cdot \hat{p})* \\
= & \quad \text{definition of guard} \\
& \otimes/ \cdot p?_\omega*.
\end{aligned}$$

The derived equation

$$\otimes/ \cdot p\lhd = \otimes/ \cdot p?_\omega* \tag{20}$$

shows that a filter followed by a reduction is a catamorphism. This transformation is carried out frequently in what follows. A filter and a map can be swapped.

(21) THEOREM (Map-Filter swap). *For all functions* $f: A \to B$ *and predicates* $p: B \to bool$ *we have,*

$$p\lhd \cdot f* = f* \cdot (p \cdot f)\lhd.$$

2.3. The Data Type Snoc-List. The data type *join-list* in the Boom-hierarchy is one way to "implement" the intuitive idea we have of lists. In this section we present another way to represent lists as a data type. The data type *snoc-list* over base type A is denoted by $A\star$. An element of the data type *snoc-list* is either the empty list $\Box$, or $x \prec a$, the concatenation of a snoc-list $x: A\star$ with an element a of type A. The data type *snoc-list* is isomorphic to the data type *join-list*, that is, we can exhibit two injective functions $sj: A\star \to A*$ and $js: A* \to A\star$ satisfying $sj \cdot js = id_{A*}$ and $js \cdot sj = id_{A\star}$.

Given an operator $\oplus: B \times A \to B$ and a value $e: B$, there exists a unique function $h: A\star \to B$, a catamorphism, such that

$$\begin{aligned} h\,(x \prec a) &= (h\,x) \oplus a, \\ h\,\Box &= e. \end{aligned} \tag{22}$$

Many algorithms on lists are catamorphisms on the data type *snoc-list*. Therefore, we give a name to catamorphisms on the data type *snoc-list*; a catamorphism on the data type *snoc-list* is called a *left-reduction*. The unique function h satisfying (22) is written $\oplus \!\!\rightarrowtail\!\! e$. An example of a left-reduction is the function $\#$ which returns the length of a list. It is the left-reduction $((+1) \cdot \ll) \!\!\rightarrowtail\!\! 0$. Another example is the concatenation operator $\Cap$ defined by $x \Cap y = (\prec \!\!\rightarrowtail\!\! x)\, y$. Note that $\Cap$ is associative. The identity left-reduction id is defined by $\prec \!\!\rightarrowtail\!\! \Box$. We use $[a, b, c]$ as an abbreviation for $\Box \prec a \prec b \prec c$.

Let h be a recursively computable function defined on lists, and let φ be a program that computes the value of h on an argument. Following Galil [13] program φ is called *on line* if it processes one element of the argument at a time, and if the value of h on the processed elements is available before the next element from the argument is read. So given an argument x, value $h\,x$ is computed by means of a series of successive approximations $h\,y$ where the y's are initial parts of x. It follows that the recursive structure of an on-line program follows that of a left-reduction. A program is called *real time* if it is on line and if in addition there exists a constant c such that the number of steps required to compute $h\,(x \prec a)$ given the value of $h\,x$ is bounded by c. It follows that a real-time program can be viewed as a left-reduction the operator of which can be evaluated in constant time.

Again, we have a fusion theorem for this data type.

(23) THEOREM (Snoc-List Fusion). *Suppose* $f: B \to C$, $\oplus: B \times A \to B$, *and* $\otimes: C \times A \to C$ *satisfy*

$$f\,(x \oplus a) = (f\,x) \otimes a$$

for x in the image of $\oplus\!\!\rightarrow e$. *Then*

$$f \cdot \oplus\!\!\rightarrow e = \otimes\!\!\rightarrow (f\ e).$$

If h is a left-reduction, value $h\ (x \prec\!\!\!\!\cdot\ a)$ is expressed in terms of value $h\ x$ and a. For function h that is not a left-reduction more information is needed for a recursive definition of h. For each function h, value $h(x \prec\!\!\!\!\cdot\ a)$ can be expressed in terms of $h\ x$, a, and the argument x itself, that is, for each function $h: A\star \rightarrow B$ there exist operator $\oplus : (B \times A\star) \times A \rightarrow B$ and value e of type B such that

$$\begin{aligned} h\ (x \prec\!\!\!\!\cdot\ a) &= (h\ x, x) \oplus a, \\ h\ \square &= e. \end{aligned} \tag{24}$$

Note that the form of these characterizing equations is almost that of the characterizing equations of a catamorphism, except for the tupling with the argument in the left-hand argument of $\oplus$. Given operator $\oplus$ and value e there is a unique function satisfying these equations. Such a function is written $\oplus\!\!\Rightarrow e$, and it is called a *paramorphism* by Meertens. Meertens [24] defines paramorphisms on arbitrary data types, and shows that they possess calculational properties very similar to the properties of catamorphisms. One of these properties is the Parafusion Theorem.

(25) THEOREM (Parafusion). *Suppose* f, $\oplus$, *and* $\otimes$ *satisfy*

$$f\ ((x, y) \oplus a) = (f\ x, y) \otimes a$$

for x in the image of $\oplus\!\!\Rightarrow e$. *Then*

$$f \cdot \oplus\!\!\Rightarrow e = \otimes\!\!\Rightarrow (f\ e).$$

2.4. Combinators and Auxiliary Functions. The operator $\uparrow_f$, where f is of type $A \rightarrow B$, and B is totally ordered, is a binary operator of type $A \times A \rightarrow A$. It is defined by

$$x \uparrow_f y = \begin{cases} x & \text{if } x >_f y, \\ y & \text{if } y >_f x. \end{cases} \tag{26}$$

We do not yet define $\uparrow_f$ on arguments which have equal f-values, except that one of the arguments is the outcome. It might be necessary to define $\uparrow_f$ differently for different problems. If the choice made by the operator $\uparrow_f$ on equal f-values is immaterial to the problem, we will not give its exact definition. The operator $\downarrow_f$ is defined similarly.

The following two operators abbreviate frequently occurring expressions:

$$(x, y, z) \triangledown a = (x, y, z \prec\!\!\!\!\cdot\ a), \tag{27}$$

$$(x, y) \diamond a = (x, y \prec\!\!\!\!\cdot\ a). \tag{28}$$

The function $hd: A\star \to A$ returns the first element of a nonempty list, and the function $tl: A\star \to A\star$ returns all but the first elements of a nonempty list, so

$$hd\,([a] +\!\!\!+ x) = a, \tag{29}$$

$$tl\,([a] +\!\!\!+ x) = x. \tag{30}$$

Functions $lt: A\star \to A$ and $it: A\star \to A\star$ are the couterparts of functions *hd* and *tl*. They are defined by

$$lt\,(x +\!\!\!+ [a]) = a, \tag{31}$$

$$it\,(x +\!\!\!+ [a]) = x. \tag{32}$$

Given a list x and a natural number n we define four functions called *takeb* (take from the back), *takef* (take from the front), *dropb* (drop from the back), and *dropf* (drop from the front) which respectively take the last n elements of x, take the first n elements of x, drop the last n elements of x (which equals taking all but the last n elements of x), and drop the first n elements of x. Since these functions are rarely used, and since their definitions can easily be inferred from the following example definition and the definitions in words given above, we only give the definition of function *dropb*:

$$\begin{aligned} (dropb\ 0)\,x &= x, \\ (dropb\ (n+1))\,x &= \begin{cases} (dropb\ n)\,(it\ x) & \text{if } x \neq \square, \\ \square & \text{otherwise.} \end{cases} \end{aligned} \tag{33}$$

3. Segment Theory. Finding the longest palindromic segment is specified by

$$lps = \uparrow_{\#}/ \cdot ispal \lhd \cdot segs, \tag{34}$$

where *ispal* is the predicate determining whether its argument is a palindrome or not, and *segs* is a function returning all segments of a list. This specification is an example of a segment problem. Segment problems are specified by

$$\oplus/ \cdot f* \cdot segs \tag{35}$$

for arbitrary operator $\oplus$ and function f. Note that by (20) this class of segment problems includes all problems of the form $\oplus/ \cdot p \lhd \cdot segs$ for arbitrary predicate p. Theory dealing with segment problems has been developed in [2] and [4]. However, this theory is not straightforwardly applicable to the palindrome problem. The problem is that information is needed, but this information is not available in the context where it is needed. Here, a segment does not carry information about how it occurs, and this information is needed to find out

whether a palindrome is extendible or maximal. To decide whether a palindromic segment is maximal or extendible, the context of the segment is needed. The generic solution is to tuple with the information needed, and in this particular case we tuple a segment with its context.

Section 3.1 defines the functions for computing segments. These definitions deviate from the definitions given by Bird, even when the contexts of the segments are removed. Section 3.2 derives some fusion theorems for the functions introduced in Section 3.1. The first theorem, the *split3s*-Fusion Theorem, gives a condition a segment problem has to satisfy in order to obtain an efficient algorithm for it. The *split3s*-Fusion Theorem has three important corollaries. The first corollary is called Horner's rule and corresponds to a combination of theorems presented in [3]. The other corollaries of the *split3s*-Fusion Theorem, the Sliding Tails Theorem and the Hopping Tails Theorem, are theorems that unify and generalize results presented in [17] and [4]. The Hopping Tails Theorem is based on ideas presented in [28] and [20]. The form of the functions given in Section 3.1 allows us to give rather simple derivations of these theorems. These fusion theorems are the theorems by means of which on-line algorithms for problems on segments are derived.

3.1. Segments. A segment of a list is a list consisting of a number of consecutive elements from the argument. Formally, list y is a segment of list v if and only if there exist lists x and z such that $v = x \mathbin{+\!\!\!+} y \mathbin{+\!\!\!+} z$. Problems on segments have been studied widely; the amount of literature on just the pattern-matching problem, which requires finding occurrences of a given pattern (segments equal to the pattern) in a list, is enormous. A large number of (solutions to) segment-problems can be found in the book *Combinatorial Algorithms on words* edited by Apostolico and Galil [1].

A segment of a list is tupled with the part of the list in front of it and the part of the list after it. Function $split3s: A\star \to (A\star \times A\star \times A\star)*$ returns all ways in which a list can be split into three parts (which explains its name). Using set-comprehension we have

$$split3s\ v = \{(x, y, z) \mid v = x \mathbin{+\!\!\!+} y \mathbin{+\!\!\!+} z\}. \tag{36}$$

To perform calculations, *split3s* should be defined on the data type list as a catamorphism or a paramorphism, or as a catamorphism or a paramorphism followed by a function that can be evaluated cheaply. Function *split3s* is a left-reduction, but the definition is rather awkward, and therefore it is defined as a paramorphism. For the domain of the paramorphism there are two choices: *split3s* can be defined as a join-list paramorphism or as a snoc-list paramorphism. The choice of definition determines the form of the final algorithm that will be constructed, that is, if *split3s* is defined as a join-list paramorphism, the algorithm that will be constructed is a join-list paramorphism, etc. We want to derive on-line algorithms, and therefore we define *split3s* as a snoc-list paramorphism. Even now we can choose between several definitions. The choice made here seems to be the most appropriate for our purposes when inspecting the length of the derivations

for the different definitions. Two characterizing equations for *split3s* are

$$\begin{aligned} &split3s\ \square = \{(\square, \square, \square)\}, \\ &split3s\ (x \prec\!\!\!\!\cdot\ a) = ((\heartsuit a)*\ split3s\ x) \cup ((\propto \square)*\ split2s\ (x \prec\!\!\!\!\cdot\ a)), \end{aligned} \tag{37}$$

where function $split2s : A\star \to (A\star \times A\star)*$ returns, given a list, all ways in which this list can be split into two parts. Using set-comprehension, *split2s* is defined by

$$split2s\ v = \{(x, y) | v = x \prec\!\!\!\!\cdot\!\!\!\!\cdot\ y\}. \tag{38}$$

Two characterizing equations for *split2s* are

$$\begin{aligned} &split2s\ \square = \{(\square, \square)\}, \\ &split2s\ (x \prec\!\!\!\!\cdot\ a) = ((\diamondsuit a)*\ split2s\ x) \cup \{(x \prec\!\!\!\!\cdot\ a, \square)\}. \end{aligned} \tag{39}$$

Function *split2s* is a paramorphism

$$split2s = \odot \not\Rightarrow \{(\square, \square)\}, \tag{40}$$

where operator $\odot$ is defined by

$$(x, y) \odot a = ((\diamondsuit a)*\ x) \cup \{(y \prec\!\!\!\!\cdot\ a, \square)\}. \tag{41}$$

To obtain a paramorphism followed by a function that can be evaluated cheaply for function *split3s*, we tuple the computation of *split3s* with the computation of *split2s*. Define

$$ss = split3s \triangle split2s. \tag{42}$$

Then, according to equality (7)

$$split3s = \lll \cdot ss. \tag{43}$$

The definition of *ss* as a paramorphism is straightforward. We have

$$ss = \otimes \not\Rightarrow (\{(\square, \square, \square)\}, \{(\square, \square)\}), \tag{44}$$

where operator $\otimes$ is defined by

$$((x, y), z) \otimes a = (((\heartsuit a)*\ x) \cup (\propto \square)*\ ((y, z) \odot a), (y, z) \odot a). \tag{45}$$

Using the new terminology, the specification of the palindrome problem becomes

$$lps = \uparrow_{\#\cdot\pi_2} / \cdot (ispal \cdot \pi_2) \lhd \cdot split3s. \tag{46}$$

Thus we do not find just the longest palindromic segment of a list, but the parts to the left and right of the list of which it is a segment too. The specification can be implemented as a functional program. Given a list of length n, this program requires time $\Omega(n^3)$ to find the longest palindrome in this list. This can be seen as follows. Function *split3s* returns a list containing $\Omega(n^2)$ triples, and every triple is of length $\Omega(n)$. Since computing *ispal* for a list requires linear time, and $\uparrow_{\#\cdot\pi_2}$ can be implemented such that it requires constant time, we have that our specification requires time $\Omega(n^2) \times \Omega(n) = \Omega(n^3)$.

3.2. Fusion Theorems. In this subsection we give the main theorems used in the derivation of on-line algorithms for segment problems. The theorems given here are not specific to the palindrome finding problem; they reappear in almost every derivation of an algorithm for a segment problem, though with *split2s* usually replaced by *tails* (*tails* is the function which, applied to a list, returns all tail segments of that list) and *split3s* replaced by *segs*. The generic specification of segment problems is given by

$$\oplus/ \cdot f* \cdot split3s, \tag{47}$$

where $\oplus$ is an arbitrary operator and f is an arbitrary function. Given a segment problem, we want to obtain an algorithm for it that can be implemented as an efficient program. Our first goal is to find conditions such that the specification equals a paramorphism. Since *split3s* is not a paramorphism, we cannot apply the Parafusion Theorem to the specification. Using (5) we derive, for arbitrary function g,

$$\begin{aligned} & \oplus/ \cdot f* \cdot split3s \\ = & \quad {\scriptstyle (43)} \\ & \oplus/ \cdot f* \cdot \lessdot \cdot ss \\ = & \quad {\scriptstyle (5)} \\ & \lessdot \cdot (\oplus/ \cdot f*) \times g \cdot ss. \end{aligned}$$

For g we can choose any function that suits us, and in the subsequent calculation a natural candidate will emerge. Function *ss* is a paramorphism, so we can apply the Parafusion Theorem to the expression $(\oplus/ \cdot f*) \times g \cdot ss$. For that purpose we have to compute

$$((\oplus/ \cdot f*) \times g)\,(\{(\square, \square, \square)\}, \{(\square, \square)\}),$$

which equals $(f\,(\square, \square, \square), g\,\{(\square, \square)\})$, and we have to find an operator $\oslash$ such that, for all (x, y) in the image of *ss*, and for all a,

$$((\oplus/ \cdot f*) \times g)\,(((x, y), w) \otimes a) = (((\oplus/ \cdot f*) \times g)\,(x, y), w) \oslash a. \tag{48}$$

The definition of an operator $\oslash$ satisfying (48) is synthesized as follows. Meanwhile we will find a candidate for function g:

$$
\begin{aligned}
& ((\oplus/\cdot f*) \times g)\,(((x, y), w) \otimes a) \\
= & \quad \text{definition of } \otimes \text{ (45)} \\
& ((\oplus/\cdot f*) \times g)\,((\heartsuit a)*\, x \cup (\propto \square)*\, ((y, w) \odot a), (y, w) \odot a) \\
= & \quad \text{definition of } \times \\
& (\oplus/f*\, (((\heartsuit a)*\, x \cup (\propto \square)*\, ((y, w) \odot a)), g\,((y, w) \odot a)) \\
= & \quad \text{definition of catamorphism} \\
& ((\oplus/f*\, (\heartsuit a)*\, x) \oplus (\oplus/f*\, (\propto \square)*\, ((y, w) \odot a)), g\,((y, w) \odot a)) \\
= & \quad \text{assumption below, map distributivity} \\
& (((\heartsuit a) \oplus/\, f*\, x) \oplus (\oplus/\,(f \cdot (\propto \square))*\, ((y, w) \odot a)), g\,((y, w) \odot a)).
\end{aligned}
$$

In the last step of this calculation we applied the equality:

$$(\heartsuit a) \cdot \oplus/\cdot f* = \oplus/\cdot f* \cdot (\heartsuit a)*.$$

This equality is easily proved if we assume that $(\heartsuit a)$ is $(\oplus, \oplus)$-fusable and commutes with function f. In the last expression of this calculation we distinguish three subexpressions:

$$
\begin{aligned}
& (\heartsuit a) \oplus/\, f*\, x, \\
& \oplus/\,(f \cdot (\propto \square))*\, ((y, w) \odot a), \\
& g\,((y, w) \odot a).
\end{aligned}
$$

In view of the form of the desired expression (48) the first subexpression $(\heartsuit a) \oplus/\, f*\, x$ need not be developed any further, and the subexpressions $\oplus/\,(f \cdot (\propto \square))*\, ((y, w) \odot a)$ and $g\,((y, w) \odot a)$ have to be expressed in terms of $g\ y$, w, and a. Hence a reasonable choice for g seems to be $\oplus/\cdot(f \cdot (\propto \square))*$. The remaining task is to express $\oplus/\,(f \cdot (\propto \square))*\, ((y, w) \odot a)$ in terms of

$$\oplus/\,(f \cdot (\propto \square))*\, y, w, \quad \text{and} \quad a.$$

We have

$$
\begin{aligned}
& \oplus/\,(f \cdot (\propto \square))*\, ((y, w) \odot a) \\
= & \quad \text{definition of } \odot \text{ (41)} \\
& \oplus/\,(f \cdot (\propto \square))*\, ((\diamondsuit a)*\, y \cup \{(w \prec\!\!\!\!\cdot\, a, \square)\}) \\
= & \quad \text{definition of catamorphism} \\
& (\oplus/\,(f \cdot (\propto \square))*\, (\diamondsuit a)*\, y) \oplus (f\,(w \prec\!\!\!\!\cdot\, a, \square, \square)) \\
= & \quad \text{assumption} \\
& ((\ominus a) \oplus/\,(f \cdot (\propto \square))*\, y) \oplus (f\,(w \prec\!\!\!\!\cdot\, a, \square, \square))
\end{aligned}
$$

for some operator $\ominus$. This assumption is the condition of the *split3s*-Fusion Theorem formulated below. Note that since (x, y) is in the image of *ss*, it follows that y is in the image of *split2s*. Thus we have found that operator $\oslash$ can be defined by

$$((x, y), w) \oslash a = ((x \heartsuit a) \oplus (y \ominus a) \oplus u, (y \ominus a) \oplus u), \tag{49}$$

where u abbreviates $f(w \prec a, \square, \square)$. We summarize the derivation in the following theorem.

(50) THEOREM (*split3s*-Fusion). *Suppose there exists an operator $\ominus$ satisfying equation $\oplus/ \cdot (f \cdot (\propto \square))* \cdot (\diamondsuit a)* = (\ominus a) \cdot \oplus/ \cdot (f \cdot (\propto \square))*$ on the image of split2s. Furthermore, suppose section $(\heartsuit a)$ is $(\oplus, \oplus)$-fusable and commutes with f. Then*

$$\oplus/ \cdot f* \cdot split3s = \lessdot \cdot \oslash \gtrdot (v, v),$$

where v abbreviates $f(\square, \square, \square)$, and operator $\oslash$ is defined by

$$((x, y), w) \oslash a = ((x \heartsuit a) \oplus (y \ominus a) \oplus u, (y \ominus a) \oplus u),$$

where $u = f(w \prec a, \square, \square)$.

The equality

$$\otimes/ (f \cdot (\propto \square))* (\diamondsuit a)* y = (\ominus a) \oplus/ f* y \tag{51}$$

assumed in the *split3s*-Fusion Theorem can be split into separate assumptions on the constituents $\oplus$ and f. There are various ways to obtain equality (51). One way is to assume that function f satisfies $f \cdot (\propto \square) \cdot (\diamondsuit a) = (\ominus a) \cdot f \cdot (\propto \square)$ for some operator $\ominus$ such that section $(\ominus a)$ is $(\oplus, \oplus)$-fusable. Since equality (51) is only required to hold on the image of *split2s*, section $(\ominus a)$ needs only be $(\oplus, \oplus)$-fusable for elements of the form $\oplus/ (f \cdot (\propto \square))* s$, $\oplus/ (f \cdot (\propto \square))* t$, where s, t are subsets from *split2s* z for some list z.

(52) THEOREM (Horner's Rule). *Let f be a function satisfying*

$$f \cdot (\propto \square) \cdot (\diamondsuit a) = (\ominus a) \cdot f \cdot (\propto \square)$$

for some operator $\ominus$ such that section $(\ominus a)$ is $(\oplus, \oplus)$-fusable for elements of the form $\oplus/ (f \cdot (\propto \square))* s$, $\oplus/ (f \cdot (\propto \square))* t$, where s, t are subsets from split2s z for some list z. Furthermore, suppose section $(\heartsuit a)$ is $(\oplus, \oplus)$-fusable and commutes with f. Then

$$\oplus/ \cdot f* \cdot split3s = \lessdot \cdot \oslash \gtrdot (v, v),$$

where v abbreviates $f(\Box, \Box, \Box)$, and operator $\oslash$ is defined by

$$((x, y), w) \oslash a = ((x \heartsuit a) \oplus (y \ominus a) \oplus u, (y \ominus a) \oplus u),$$

where $u = f(w \prec a, \Box, \Box)$.

Consider a specification of the form

$$\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\triangleleft\cdot split3s, \tag{53}$$

of which finding the longest palindromic segment is an instance. Such a problem is called a *longest-p segment* problem. By definition of guard (20) this specification equals

$$\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)?_\omega*\cdot split3s,$$

where $\omega = v_{\uparrow_{\#\cdot\pi_2}}$. We can derive conditions on predicate p such that the conditions of Horner's rule are satisfied. One of these conditions on predicate p is that it is prefix-closed. A predicate p is *prefix-closed*, see [2], if $p\ \Box$ holds, and if p satisfies for all lists x and values a

$$p\,(x \prec a) \quad \Rightarrow \quad p\,x.$$

A *postfix-closed* predicate is defined similarly. The conditions predicate p has to satisfy in order to apply Horner's rule are not satisfied by predicate *ispal*, which is defined by

$$ispal\ x = (x = rev\ x), \tag{54}$$

where *rev* is the function which returns the reverse of a list. Instead of applying Horner's rule to a longest-p segment problem specified in (53), we derive another corollary of the *split3s*-Fusion Theorem for this class of problems. The condition of the *split3s*-Fusion Theorem, (51), reads in the case considered now

$$\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\triangleleft\cdot(\propto\Box)*\cdot(\Diamond a)* = (\ominus a)\cdot\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\triangleleft\cdot(\propto\Box)* \tag{55}$$

for some operator $\ominus$. Using the definition of guard and map distributivity, the composition $\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\triangleleft\cdot(\propto\Box)*$ can be rewritten to

$$\uparrow_{\#\cdot\pi_2}/\cdot((p\cdot\pi_2)?_\omega\cdot(\propto\Box))*.$$

Abbreviate $(p\cdot\pi_2)?_\omega\cdot(\propto\Box)$ to $\surd p$. We have to prove this equality for elements $y = split2s\ z$ for some list z. As an aside, we remark that if this equality holds for some predicate p and some operator $\ominus$, then $\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\triangleleft\cdot split2s$ is a left-

reduction $\ominus' \mapsto ((p \cdot \pi_2)?_\omega (\Box, \Box))$, where operator $\ominus'$ is defined by

$$(x, y) \ominus' a = \natural((x, y, \Box) \ominus a) \uparrow_{\#\cdot\pi_2} (p \cdot \pi_2)?_\omega (x \mathbin{\#\!\!\prec} y \prec a, \Box).$$

A property similar to prefix-closed satisfied by *ispal* is

$$ispal\,([a] \mathbin{\#\!\!\prec} x \mathbin{\#\!\!\prec} [a]) \quad \Rightarrow \quad ispal\,x. \tag{56}$$

Both the property satisfied by *ispal* and the property prefix-closed are captured in the notion c-slow. A predicate is *c-slow* for a constant $c \geq 0$ if, for all lists y with $\#y = c$, all lists x, and all elements a,

$$p\,(y \mathbin{\#\!\!\prec} x \mathbin{\#\!\!\prec} [a]) \quad \Rightarrow \quad p\,x. \tag{57}$$

A prefix-closed predicate is 0-slow. The predicate *ispal* is 1-slow. A predicate can be c-slow for various constants c, consider for example segment-closed predicates (a predicate is *segment-closed* if it is both prefix-closed and postfix-closed). A segment-closed predicate is c-slow for all natural numbers c. For the moment we assume in addition that c-slow predicates hold for all lists of length at most c. For every c-slow predicate p there exists a *derivative* function ∇ such that, for all lists y with $\#y = c$, all lists x, and all values a,

$$p\,(y \mathbin{\#\!\!\prec} x \prec a) = p\,x \wedge a \nabla_x y. \tag{58}$$

For example, predicate *ispal* has derivative function ∇ defined by

$$a \nabla_x [b] = (a = b).$$

Let γ be a function that given a constant c and a list x splits list x into a pair of lists (u, v) such that $x = u \mathbin{\#\!\!\prec} v$ and list v is the tail of length c of x if $\#x \geq c$ and x itself otherwise, $(\gamma\, c)\, x = ((dropb\, c)\, x, (takeb\, c)\, x)$. Consider the operator $\ominus$ that given a triple of lists (x, y, z) and an element a returns the triple of lists (s, t, z) such that $s \mathbin{\#\!\!\prec} t = x \mathbin{\#\!\!\prec} y \prec a$ and t is the longest tail of $x \mathbin{\#\!\!\prec} y \prec a$ satisfying predicate p. We hope to define operator $\ominus$ such that, for c-slow predicates p,

$$\uparrow_{\#\cdot\pi_2} / \cdot (\surd p) * \cdot split2s = \ominus \mapsto (\Box, \Box, \Box).$$

This result is a byproduct of the proof of the fact that operator $\ominus$ satisfies the applicability condition of the *split3s*-Fusion Theorem. Operator $\ominus$ is defined by

$$(x, y, z) \ominus a = \begin{cases} (s, t \mathbin{\#\!\!\prec} y \prec a, z) & \text{if} \quad p\,(t \mathbin{\#\!\!\prec} y \prec a), \\ (x \prec (hd\, y), tl\, y, z) \ominus a & \text{if} \quad \neg p\,(t \mathbin{\#\!\!\prec} y \prec a) \wedge y \neq \Box, \\ ((\gamma\, c)\, (x \prec a)) \propto z & \text{otherwise,} \end{cases} \tag{59}$$

where

$$(s, t) = (\gamma\, c)\, x.$$

Operator $\ominus$ satisfies the condition of the *split3s*-Fusion Theorem if predicate p is c-slow, that is, equation

$$\uparrow_{\#\cdot\pi_2/}\cdot(\surd p)*\cdot(\Diamond a)*\cdot split2s = (\ominus a)\cdot\uparrow_{\#\cdot\pi_2/}\cdot(\surd p)*\cdot split2s \tag{60}$$

holds for c-slow predicates p. We prove this equality in the Appendix. We have the following corollary of the *split3s*-Fusion Theorem.

(61) THEOREM (Sliding Tails). *Let p be a c-slow predicate. Then*

$$\uparrow_{\#\cdot\pi_2/}\cdot(p\cdot\pi_2)\lhd\cdot split3s = \lll\cdot\oslash\nRightarrow(v, v),$$

where v abbreviates $(\Box, \Box, \Box)$, and operator $\oslash$ is defined by

$$((x, y), w)\oslash a = ((x\heartsuit a)\uparrow_{\#\cdot\pi_2}(y\ominus a)\uparrow_{\#\cdot\pi_2}u, (y\ominus a)\uparrow_{\#\cdot\pi_2}u),$$

where u abbreviates $(w \nprec a, \Box, \Box)$, and operator $\ominus$ is defined by

$$(x, y, z)\ominus a = \begin{cases}(s, t \nprec y \nprec a, z) & \text{if } p\,(t \nprec y \prec a),\\ (x \prec (hd\ y), tl\ y, z)\ominus a & \text{if } \neg p\,(t \nprec y \prec a) \wedge y \neq \Box,\\ ((\gamma\ c)\ (x \prec a))\propto z & \text{otherwise,}\end{cases}$$

where

$$(s, t) = (\gamma\ c)\ x.$$

The name of this theorem, the Sliding Tails Theorem, is due to Oege de Moor. It refers to a visual interpretation of the algorithm $\oslash\nRightarrow((\Box, \Box, \Box), (\Box, \Box, \Box))$. Given a list, this paramorphism proceeds from left to right, at each point returning for the part of the list scanned the longest segment satisfying p and the longest tail satisfying p (with their contexts). The longest tail satisfying p is either the extension of the longest tail satisfying p at the previous point, or the tail of that tail, or the tail of the tail of that tail, etc. Thus the longest tail "slides" over the argument.

Consider the case in which the *split3s*-Fusion Theorem is applied to a longest-p segment problem with c-slow predicate p, and operator $\ominus$ is defined by (59). Since $p\,(t \nprec y \prec a)$ holds only if $p\ y$ holds, it follows that $(x, y, z)\ominus a$ evaluates to $(x \prec (hd\ y), tl\ y, z)\ominus a$ as long as $p\ y$ does not hold (and $y \neq \Box$). So

$$(x, y, z)\ominus a = (x \prec (hd\ y) \nprec u, v, z)\ominus a,$$

where (u, v) is the longest element in *split2s tl y* of which the second component satisfies p. Let *lrp* be the function that, given a list, returns the splitting with longest second component satisfying p:

$$lrp = \uparrow_{\#\cdot\pi_2/}\cdot(p\cdot\pi_2)\lhd\cdot split2s. \tag{62}$$

If we can prove that equality (55) holds for c-slow predicate p and some operator $\textcircled{1}$, then, according to the remark made after equality (55), $lrp = \textcircled{1}'\!\!\rightarrow\!\!(\square, \square)$, where operator $\textcircled{1}'$ is defined by $(x, y)\,\textcircled{1}'\,a = \natural((x, y, \square)\,\textcircled{1}\,a)\uparrow_{\#\cdot\pi_2}(x \nprec y \prec a, \square)$. Define operator $\textcircled{1}$ by

$$(63)\quad (x, y, z)\,\textcircled{1}\,a = \begin{cases} (s, t \nprec y \prec a, z) & \text{if } p\,(t \nprec y \prec a), \\ (x \prec (hd\ y) \nprec u, v, z)\,\textcircled{1}\,a & \text{if } \neg p\,(t \nprec y \prec a) \wedge y \neq \square, \\ ((\gamma\ c)\,(x \prec a)) \propto z & \text{otherwise,} \end{cases}$$

where

$$(s, t) = (\gamma\ c)\ x,$$

$$(u, v) = lrp\ tl\ y.$$

The informal discussion above on the evaluation of operator $\ominus$ implies that operator $\textcircled{1}$ defined in (63) is semantically equal to operator $\ominus$ defined in (59). A formal proof of the fact that the operators are equal may be carried out via a proof of the fact that

$$\uparrow_{\#\cdot\pi_2}/\cdot(\surd p)*\cdot split2s = \textcircled{1}\!\!\rightarrow\!\!(\square, \square, \square).$$

Operator $\textcircled{1}$ is adjusted as follows. Observe that at each time operator $\textcircled{1}$ is evaluated, the second component of its first argument satisfies p. This observation invites us to use the derivative of predicate p. Redefine operator $\textcircled{1}$ by

$$(64)\quad (x, y, z)\,\textcircled{1}\,a$$

$$= \begin{cases} (s, t \nprec y \prec a, z) & \text{if } \#x \geq c \wedge a\nabla_y t, \\ (x \prec (hd\ y) \nprec u, v, z)\,\textcircled{1}\,a & \text{if } (\#x < c \vee \neg(a\nabla_y t)) \wedge y \neq \square, \\ ((\gamma\ c)\,(x \prec a)) \propto z & \text{otherwise,} \end{cases}$$

where

$$(s, t) = (\gamma\ c)\ x,$$

$$(u, v) = lrp\ tl\ y.$$

It may be shown that operator $\textcircled{1}$ thus defined satisfies the condition of the *split3s*-Fusion Theorem, using an argument similar to that used to show that operator $\ominus$ (59) satisfies the condition of the *split3s*-Fusion Theorem. We obtain

(65) THEOREM (Hopping Tails). *Let p be a c-slow predicate. Then*

$$\uparrow_{\#\cdot\pi_2}/\cdot(p\cdot\pi_2)\lhd\cdot split3s = \ll\cdot\oslash\!\!\Rightarrow\!\!(v, v),$$

where v *abbreviates* $(\Box, \Box, \Box)$, *and operator* $\oslash$ *is defined by*

$$((x, y), w) \oslash a = ((x \heartsuit a) \uparrow_{\#\cdot\pi_2} (y \mathbin{①} a) \uparrow_{\#\cdot\pi_2} u, (y \mathbin{①} a) \uparrow_{\#\cdot\pi_2} u),$$

where u *abbreviates* $(w \prec a, \Box, \Box)$, *and operator* ① *is defined by*

$$(x, y, z) \mathbin{①} a = \begin{cases} (s, t \nVdash y \prec a, z) & \text{if } \#x \geq c \wedge a \nabla_y t, \\ (x \prec (hd\ y) \nVdash u, v, z) \mathbin{①} a & \text{if } (\#x < c \vee \neg(a \nabla_y t)) \wedge y \neq \Box, \\ ((\gamma\ c)\ (x \prec a)) \propto z & \text{otherwise,} \end{cases}$$

where

$$(s, t) = (\gamma\ c)\ x,$$

$$(u, v) = lrp\ tl\ y.$$

Again, the name of this theorem, is due to Oege de Moor and refers to a visual interpretation of the algorithm $\oslash \nRightarrow ((\Box, \Box, \Box), (\Box, \Box, \Box))$. Given a list, this paramorphism proceeds from left to right, at each point returning for the part of the list scanned the longest segment satisfying p and the longest tail satisfying p (with their contexts). The longest tail satisfying p is either the extension of the longest tail satisfying p at the previous point, or the extension of the longest tail satisfying p of the tail of that tail, etc. Thus the longest tail satisfying p "hops" over the argument.

The *split3s*-Fusion Theorem, the Sliding Tails Theorem and the Hopping Tails Theorem are tools with which many on-line algorithms for segment problems can be derived. In particular, the Hopping Tails Theorem is at the heart of a derivation of an algorithm for finding the longest palindromic segment of a string, as we show in the following section, and a derivation of the Knuth, Morris, and Pratt pattern-matching algorithm, see [4].

4. The Algorithm for Finding Palindromes. In this section we derive, using the theory developed in the previous section, a linear-time algorithm for finding the longest palindromic segment of a list. To obtain a linear-time algorithm, we compute (the positions of all) maximal palindromes occurring in a string. Before we give the details of the derivation of the algorithm in Sections 4.2 and 4.3, and a discussion of its complexity in Section 4.4, we first give some basic properties of palindromes.

4.1. Properties of Palindromes. We state a number of properties of palindromes that are used in the subsequent sections.

By definition x is a palindrome iff $rev\ x = x$, where *rev* returns the reverse of a list. The following property follows immediately from this fact.

(66) PROPERTY. *Suppose* $x = y \nVdash z$ *is a palindrome, and* z *is a palindrome. Then* $x = z \nVdash (rev\ y)$.

A similar property is satisfied by maximal palindromes. An occurrence of a palindrome in a string is called *extendible* if it is preceded and followed by equal characters, otherwise it is called *maximal* (including the case when there is no element preceding or following it). For example, in "horror" the substring "rr" is an extendible palindrome, and the substrings "orro" and "ror" are maximal palindromes. We have

(67) PROPERTY. *Suppose* $s = x \Lsh y \Lsh z$ *is a palindrome in which* y *is a maximal palindrome (hence, if* $x \neq \Box$ *and* $z \neq \Box$, *then* $lt\ x \neq hd\ z$*). Then* $s = (rev\ z) \Lsh y \Lsh (rev\ x)$, *in which* y *is again maximal.*

4.2. The First Steps. The specification of the problem of finding the longest palindromic segment has been given in Section 3, see (46).

$$lps = \uparrow_{\#\cdot\pi_2} / \cdot (ispal \cdot \pi_2) \lhd \cdot split3s.$$

In order to obtain a semantically equivalent on-line algorithm for this specification we apply one of the theorems derived in Section 3. Since the predicate *ispal* is 1-slow and has derivative function $a \nabla_x [b] = (a = b)$, we can apply the Hopping Tails Theorem to obtain

$$lps = \lll \cdot \oslash \Rsh ((\Box, \Box, \Box), (\Box, \Box, \Box)),$$

where operator $\oslash$ is defined by

$$((x, y), w) \oslash a = ((x \heartsuit a) \uparrow_{\#\cdot\pi_2} (y \oplus a) \uparrow_{\#\cdot\pi_2} u, (y \oplus a) \uparrow_{\#\cdot\pi_2} u),$$

where u abbreviates $(w \prec a, \Box, \Box)$, and operator $\oplus$ is defined by

$$(x, y, z) \oplus a = \begin{cases} (s, t \Lsh y \prec a, z) & \text{if} \quad \#x \geq 1 \wedge a = lt\ x, \\ (x \prec (hd\ y) \Lsh u, v, z) \oplus a & \text{if} \quad (x = \Box \vee a \neq lt\ x) \wedge y \neq \Box, \\ ((\gamma\ 1)\ (x \prec a)) \propto z & \text{otherwise,} \end{cases}$$

where

$$(s, t) = (\gamma\ 1)\ x,$$

$$(u, v) = lrp\ tl\ y.$$

However, the straightforward implementation of this solution is a cubic-time program. This is explained as follows. Suppose *lps* is applied to a list of length n. Operator $\oslash$ is evaluated n times. Evaluating the expression $x \heartsuit a$ and the occurrences of $\uparrow_{\#\cdot\pi_2}$ can be done in constant time. The evaluation of $y \oplus a$, however, might require quadratic time. The computation of *lrp tl y* is expensive, and the following subsection is devoted to finding means with which the value of *lrp tl y* can be found in constant time on the average.

4.3. Centers, Mirror Images, and Maximal Palindromes. The main idea in the construction of an efficient algorithm for finding the longest palindromic segment of a list is to make use of previously computed palindromes. These previously computed palindromes provide an efficient way to determine the value of *lrp tl y*. Value *lrp tl y* can be expressed in terms of *llp it y*, where *llp* ("longest left palindrome") is the following function:

$$llp = \uparrow_{\#\cdot\pi_1}/\cdot(ispal\cdot\pi_1)\lhd\cdot split2s. \tag{68}$$

Value *llp it y* on its turn can be determined from previously computed maximal palindromes. First we show how to express *lrp tl y* in terms of *llp it y*.

By definition, if x is a palindrome, then

$$tl\ x = rev\ it\ x. \tag{69}$$

Define function *prev* ("pair reverse"), which returns a reverse of a pair of lists, by

$$prev\ (x, y) = (rev\ y, rev\ x). \tag{70}$$

It can be verified that *prev*, *rev*, and *split2s* satisfy the following equation:

$$prev* \cdot split2s = split2s \cdot rev. \tag{71}$$

Furthermore, for elements of the form $\uparrow_{\#\cdot\pi_1}/(ispal\cdot\pi_1)\lhd s$, $\uparrow_{\#\cdot\pi_1}/(ispal\cdot\pi_1)\lhd t$, where s, t are subsets from *split2s z* for some list z, function *prev* is $(\uparrow_{\#\cdot\pi_1}, \uparrow_{\#\cdot\pi_2})$-fusable and function *prev* satisfies

$$ispal\cdot\pi_2\cdot prev = ispal\cdot\pi_1. \tag{72}$$

This series of equalities is used in the following derivation, which proves that *lrp tl y* = *prev llp it y*:

lrp tl y

= y is a palindrome, equality (69)

lrp rev it y

= definition *lrp*, equality (71)

$\uparrow_{\#\cdot\pi_2}/(ispal\cdot\pi_2)\lhd prev*\ split2s\ it\ y$

= map-filter swap, Theorem 21

$\uparrow_{\#\cdot\pi_2}/\ prev*\ (ispal\cdot\pi_2\cdot prev)\lhd\ split2s\ it\ y$

= equality (72), *prev* is $(\uparrow_{\#\cdot\pi_1}, \uparrow_{\#\cdot\pi_2})$-*fusable*

prev $\uparrow_{\#\cdot\pi_1}/(ispal\cdot\pi_1)\lhd split2s\ it\ y$

= definition *llp*

prev llp it y.

The left component of value *llp it y* is computed by means of previously computed maximal palindromes. Given the left component, the right component of *llp it y* is characterized by

$$(73) \qquad \gg llp\ it\ y = (dropf\ (\# \ll llp\ it\ y))\ (it\ y).$$

For the left component of *llp it y* we reason as follows. We abbreviate $\ll llp\ it\ y$ to l.

Define the center of a pair of lists by

$$(74) \qquad ctr\ (x, y) = (\# x) + (\tfrac{1}{2} \# y).$$

A pair of lists is called a palindrome if its second component is a palindrome. Let (x, y) be *lrp w* for some list w. Equivalently, x and y are the first two components of value $\gg(\oslash \Rightarrow((\square, \square, \square), (\square, \square, \square)))\ w$. By definition of x, y, and l, $(x, l) <_{ctr} (x, y)$. Furthermore, (x, l) is a palindrome, and hence there exists a maximal palindrome with center $ctr\ (x, l)$, of which the second component has length at least $\# l$. Suppose all maximal palindromes with center smaller than (or to the left of) $ctr\ (x, y)$ have been computed. These maximal palindromes are enumerated in a list in increasing order of center. This list is called Q, and it is specified by

$$(75) \qquad Q\ x = (<_{ctr} lrp\ x) \lhd sort_{ctr} \natural * ismaxpal \lhd split3s\ x,$$

where *ismaxpal* is a boolean function which takes a triple of lists and determines whether the second element of this triple is a maximal palindrome. Function $sort_{ctr}$ sorts its argument in increasing order of center. The precise definition of $sort_{ctr}$ is not important. The part of the definition of Q after the first filter expression is abbreviated to R, that is,

$$(76) \qquad R = sort_{ctr} \cdot \natural * \cdot ismaxpal \lhd \cdot split3s,$$

and

$$(77) \qquad Q\ x = (<_{ctr} lrp\ x) \lhd R\ x.$$

Given the list of maximal palindromes Q, value l is obtained as follows. Palindrome (x, l) is the longest palindrome of which the second component starts in *it y*. Hence the second component of the maximal palindrome with center $ctr\ (x, l)$ starts at or before the position at which y starts, i.e., its first component has length at most $\# x$. It follows that the unknown maximal palindrome is an element of $((\leq_{\#} x) \cdot \pi_1 \wedge \geq_{ctr} (x, \square)) \lhd Q\ w$. To obtain (x, l) from the maximal palindrome with the same center, the second component of the maximal palindrome is shrunk (at both sides) such that the second component starts at the same position as y. Function $cut\ (x, y)$ shrinks a palindrome such that it starts at the

position at which y starts. Thus, l is the second component of the longest element among

$$(cut\ (x, y))* ((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w.$$

Summarizing

$$l = \gg\uparrow_{\# \cdot \pi_2}/\ (cut\ (x, y))* ((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w,$$

where *cut* is the function which, when supplied with two arguments (x, y) and (u, v), where (u, v) is an element of $((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w$, returns the pair (x, t), where t is the list that starts at the position at which y starts, such that v is an extension of t, that is, (x, t) and (u, v) have the same center and $v \geq_{\#} t$. Formally, we have

$$(cut\ (x, y))\ (u, v) = (x, (takef\ (\#v - n))\ y), \tag{78}$$

where

$$n = 2 \times (\#x - \#u).$$

Since the elements of Q are sorted on their center, the longest element of Q satisfying $(\leq_{\#} x)\cdot \pi_1$ equals the rightmost element satisfying $(\leq_{\#} x)\cdot \pi_1$, that is, the element with largest center, satisfying this condition. Hence

$$\begin{aligned}
&\uparrow_{\# \cdot \pi_2}/\ (cut\ (x, y))* ((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w \\
=\ &\{\ \text{definition of } Q\ (75),\ \text{proviso above}\ \} \\
&\gg/\ (cut\ (x, y))* ((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w \\
=\ &\{\ \text{Fusion}\ \} \\
&(cut\ (x, y))\gg/\ ((\leq_{\#} x)\cdot \pi_1 \wedge \geq_{ctr}(x, \square))\lhd Q\ w.
\end{aligned}$$

To apply Fusion, $cut\ (x, y)$ has to be $(\gg, \gg)$-fusable. It follows that we have to show that $(cut\ (x, y))\ v_{\gg} = v_{\gg}$, and, for all a and b,

$$(cut\ (x, y))\ (a \gg b) = (cut\ (x, y))\ a \gg (cut\ (x, y))\ b.$$

The latter condition is satisfied according to the definition of $\gg$. The former condition can be satisfied as follows. Since $\gg$ does not have a unit element, we extend the domain of $\gg$ by a (fictitious) value $v_{\gg}$, and define this to be the unit element of $\gg$. The domain of $cut\ (x, y)$ is extended with the same value, and $cut\ (x, y)$ is defined on this value in such a way that the condition holds. We have found that

$$l = \gg(cut\ (x, y))\gg/\ ((\leq_{\#} x)\cdot \pi_1 \wedge (\geq_{ctr}(x, \square))\lhd Q\ w.$$

The part $\geq_{ctr}(x, \square)$ occurring in the filter may be omitted since the rightmost element of $((\leq_{\#} x)\cdot\pi_1)\lhd Q\ w$ always satisfies this predicate. Value $lrp\ tl\ y$ is expressed in terms of l as follows:

$$
\begin{array}{ll}
 & lrp\ tl\ y \\
= & \quad \text{definition } prev,\ llp \\
 & prev\ llp\ it\ y \\
= & \quad \text{equality (73), definition } l \\
 & prev\ (l, (dropf\ (\#l))\ (it\ y)) \\
= & \quad \text{definition } prev \\
 & (rev\ (dropf\ (\#l))\ (it\ y), rev\ l) \\
= & \quad l \text{ and } y \text{ are palindromes} \\
 & ((dropb\ (\#l))\ (tl\ y), l).
\end{array}
$$

We have expressed lrp in terms of the list of maximal palindromes Q, but how do we obtain this list? We tuple the computation of Q with the computation of lrp. Recall that $lrp = \textcircled{1}'{+\!\!\!\rightarrow}(\square, \square)$, where operator $\textcircled{1}'$ is defined by

$$(x, y)\ \textcircled{1}'\ a = \natural((x, y, \square)\ \textcircled{1}\ a)\uparrow_{\#\cdot\pi_2}(\square, \square),$$

where operator $\textcircled{1}$ is defined in (64). Value $Q\ (w \prec a)$ is expressed in terms of $Q\ w$ and $lrp\ (w \prec a)$ as follows. By definition of Q (77) we have

$$Q\ (w \prec a) = (<_{ctr} lrp\ (w \prec a))\lhd R\ (w \prec a).$$

Since lrp equals the left-reduction $\textcircled{1}'{+\!\!\!\rightarrow}(\square, \square)$, $lrp\ (w \prec a) = (lrp\ w)\ \textcircled{1}'\ a$. Let $(lrp\ w)\ \textcircled{1}'\ a$ be (j, k). The three cases in the definition (64) of $\textcircled{1}$ (with $a\nabla_y t$ instantiated to $a = lt\ x$) are now distinguished. For the first clause of $\textcircled{1}$ we have $ctr\ (j, k) = ctr\ lrp\ w$, and hence that

$$
\begin{array}{ll}
 & (<_{ctr}(j, k))\lhd R\ (w \prec a) \\
= & \quad \text{case assumption} \\
 & (<_{ctr} lrp\ w)\lhd R\ (w \prec a) \\
= & \quad \text{proviso below} \\
 & (<_{ctr} lrp\ w)\lhd R\ w \\
= & \quad \text{definition } Q \text{ (77)} \\
 & Q\ w.
\end{array}
$$

The middle step in the above calculation is valid because of the fact that no palindrome with center to the left of $lrp\ w$ can contain the last a of $w \prec a$, since

this would contradict the definition of *lrp*. The second clause in the definition of ① is dealt with as follows:

$$
\begin{aligned}
& (<_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a) \\
= & \quad \text{split } <_{ctr} \text{ in two parts} \\
& (<_{ctr} lrp\ w \vee (\geq_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a) \\
= & \quad R \text{ is sorted on centre} \\
& (<_{ctr} lrp\ w) \lhd R\ (w \mathbin{\not\prec} a) \mathbin{\not\Vdash} (\geq_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a).
\end{aligned}
$$

For the left-hand argument of $\not\Vdash$ it has been shown that

$$(<_{ctr} lrp\ w) \lhd R\ (w \mathbin{\not\prec} a) = Q\ w.$$

Simplifying the right-hand argument of $\not\Vdash$ requires more effort. First, note that by the case assumption $y \neq \square \wedge (a \neq lt \ll lrp\ w \vee \ll lrp\ w = \square)$, the maximal palindrome around *ctr lrp w* is *lrp w* itself. Hence

$$
\begin{aligned}
& (\geq_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a) \\
= & \quad \text{case assumption} \\
& [lrp\ w] \mathbin{\not\Vdash} (>_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a).
\end{aligned}
$$

For the right-hand argument of $\not\Vdash$ we reason as follows. Let (h, i) be an arbitrary element from $(>_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a)$. Then, by definition of (j, k), (h, i) is a maximal palindrome, with i a segment of $\gg lrp\ w$, and with center in between *ctr lrp w* and *ctr* (j, k). By Property (67) we then have that i is also a maximal palindrome contained in $\gg lrp\ w$ with center in between $ctr\ (\ll lrp\ w, k)$ and *ctr lrp w*. Hence the mirrored version of (h, i) is an element from

$$(>_{ctr}(\ll lrp\ w, k)) \lhd Q\ w.$$

The maximal palindrome (h, i) is computed from $(>_{ctr}(\ll lrp\ w, k)) \lhd Q\ w$ by means of the function *mir*. Function *mir* is defined by

$$(mir\ (s,t))\ (q,r) = (q \mathbin{\not\Vdash} ((takef\ n)\ t), r), \tag{79}$$

where

$$n = 2 \times (ctr\ (s,t) - ctr\ (q,r)).$$

Value n is positive in the cases in which *mir* is applied. Since Q has to remain sorted on centre, function *rev* is applied. We have

$$
\begin{aligned}
& (>_{ctr} lrp\ w \wedge <_{ctr}(j,k)) \lhd R\ (w \mathbin{\not\prec} a) \\
= & \quad \text{proviso above} \\
& rev\ (mir\ (lrp\ w)) * (>_{ctr}(\ll lrp\ w, k)) \lhd Q\ w.
\end{aligned}
$$

Finally, in the third clause of the definition of $\oplus$ the center is moved one place, and the empty list has been found to be maximal around the previous center. Hence

$$\begin{aligned} & Q\,(w \prec\!\!\!\!\cdot\; a) \\ = & \quad \text{case assumption} \\ & (Q\,w) \mathbin{\#} [(w, \square)]. \end{aligned}$$

We have expressed $Q\,(w \prec\!\!\!\!\cdot\; a)$ in terms of $Q\,w$ and $lrp\,(w \prec\!\!\!\!\cdot\; a)$ for all three clauses in the definition of operator $\oplus$. It follows that

$$(\uparrow_{\#\cdot\pi_2}/\cdot(ispal\cdot\pi_2)\lhd\cdot split3s) \mathbin{\triangle} lrp \mathbin{\triangle} Q$$

can be expressed as a paramorphism. This is done in he following subsection.

4.4. The Algorithm and Its Complexity. The specification of the problem for which we have derived a paramorphism in the previous section reads

$$lps = (\uparrow_{\#\cdot\pi_2}/\cdot(ispal\cdot\pi_2)\lhd\cdot split3s) \mathbin{\triangle} (lrp \mathbin{\triangle} Q),$$

that is, the computation of the longest palindromic segment is tupled with the computation of the longest palindromic tail, and the list of maximal palindromes with center to the left of the center of the longest palindromic tail. For the purpose of computing all maximal palindromes, we append a fictitious element η to the argument, which forces the center of the last palindromic tail to exceed the length of the original argument, so that all maximal palindromes occurring in the argument are computed.

Applying the Hopping Tails Theorem and the results of the previous subsection we obtain

$$lps = \oslash \mathbin{\not\Rightarrow} ((\square, \square, \square), ((\square, \square), \square)),$$

where operator $\oslash$ is defined by

$$((x, (y, z)), w) \oslash a = ((x \mathbin{\triangledown} a) \uparrow_{\#\cdot\pi_2} (\pi_1\,((y, z) \oplus a) \propto \square) \uparrow_{\#\cdot\pi_2} s, (y, z) \oplus a),$$

where s abbreviates $(w \prec\!\!\!\!\cdot\; a, \square, \square)$, and operator $\oplus$ is defined by

$$((x, y), z) \oplus a = \begin{cases} ((it\,x, [lt\,x] \mathbin{\#} y \prec\!\!\!\!\cdot\; a), z) & \text{if} \quad x \neq \square \wedge lt\,x = a, \\ ((x \prec\!\!\!\!\cdot\; (hd\,y) \mathbin{\#} u, v), z') \oplus a & \text{if} \quad (x = \square \vee lt\,x \neq a) \wedge y \neq \square, \\ ((x, [a]), z \prec\!\!\!\!\cdot\; (x, \square)) & \text{otherwise,} \end{cases}$$

where

$$(u, v) = ((dropb\ (\#\, l))\ (tl\ y),\ l),$$
$$l = \gg (cut\ (x, y)) \gg / ((\leq_{\#}\ x) \cdot \pi_1) \lhd z,$$
$$z' = z \prec (x, y) \nparallel rev\ (mir\ (x, y)) * (>_{ctr} (x, l)) \lhd z.$$

An important observation is that we do not use the contents of the maximal palindromes found, but just their position in the argument string. Hence, instead of computing a list of maximal palindromes, it suffices to keep a list of positions of maximal palindromes. This is a simple transformation of the above algorithm. The only part that changes is the definition of the operator ⓘ. We have

$$((x, y), z)\ ⓘ\ a = \begin{cases} ((it\ x, [lt\ x] \nparallel y \prec a), z) & \text{if } x \neq \square \wedge lt\ x = a, \\ ((x \prec (hd\ y) \nparallel u, v), z')\ ⓘ\ a & \text{if } (x = \square \vee lt\ x \neq a) \wedge y \neq \square, \\ ((x, [a]), z \prec (\#\, x, 0)) & \text{otherwise,} \end{cases}$$

where

$$(u, v) = ((dropb\ l)\ (tl\ y),\ takeb\ l\ (tl\ y)),$$
$$l = \gg (cut'\ (lx, ly)) \gg / ((\leq lx) \cdot \pi_1) \lhd z,$$
$$(lx, ly) = (\#\, x,\ \#\, y),$$
$$z' = z \prec (lx, ly) \nparallel rev\ (mir'\ (lx, ly)) * (>_{ctr'} (lx, l)) \lhd z,$$
$$(cut'\ (x, y))(u, v) = (x, v - 2 \times (x - u)),$$
$$(mir'\ (s, t))(q, r) = (q + 2 \times (ctr'\ (s, t) - ctr'\ (q, r)), r),$$
$$ctr'\ (x, y) = x + \tfrac{1}{2} y.$$

This algorithm for finding the longest palindromic segment, the longest palindromic tail, and all maximal palindromes of a list, can be implemented as a RAM program which requires time linear in the length of the list to which it is applied. The parts of this implementation that are not entirely straightforward are the parts corresponding to the computation of l and z' and in particular the parts corresponding to respectively the expression $\gg / ((\leq lx) \cdot \pi_1) \lhd z$ and the expression $(>_{ctr'} (lx, l)) \lhd z$. In both cases the implementation should scan z from right to left starting at its right end. We give a rather informal argument to explain why this implementation is a linear-time program.

One of the key observations is that the number of maximal palindromes occurring in a list is linear in the length of the list. To be precise, given a list x of length n, there are exactly $2n + 1$ palindromes in x. This is proved by showing that there are exactly $2n + 1$ center positions in a list of length n, and around every position there is exactly one maximal palindrome.

Given a list of length n, the implementation of operator $\oslash$ is evaluated exactly n times. However, in the definition of operator $\oslash$ there is an occurrence of operator $\textcircled{1}$ which is defined recursively. Obviously, the implementations of the first and third clause in the definition of operator $\textcircled{1}$ require constant time for their evaluation. For the implementation of the second clause in the definition of $\textcircled{1}$ we have that the total number of steps made in this clause is linear in the number of positions the center of the longest palindromic tail under consideration has moved. Hence the implementation of operator $\textcircled{1}$ requires constant time on the average. Since the implementation of operator $\textcircled{1}$ requires constant time on the average, the implementation of paramorphism $\oslash \not\Rightarrow((\square, \square, \square), ((\square, \square), \square))$ requires linear time.

5. Variations. In this section we consider some variations and applications of the algorithm for finding the longest palindromic segment of a list and the list of all maximal palindromes of a list. Section 5.1 shows that the algorithm given in Section 4 can be slightly generalized. Section 5.2 presents an algorithm for finding the longest segment in a list which is the concatenation of two palindromes. Section 5.3 gives an algorithm for finding the shortest partition in palindromes of a list, and Section 5.4 outlines an algorithm which determines the largest palindromic rectangle in a matrix. The algorithms we give are not derived in so much detail as the algorithm derived in the previous section.

5.1. The Longest Nonpalindromic Segment. In this section we show that the algorithm for finding the longest palindromic segment given in Section 4 can be used to solve a similar problem. Until now, we have considered the computation of "plain" palindromes. The crucial property of palindromes used in the derivation of the algorithm is

$$ispal\,([a] \not\Leftarrow v \not\Leftarrow [b]) = (ispal\ v) \wedge (a = b).$$

Suppose we generalize this by replacing $=$ by some relation $\odot$. What properties does $\odot$ have to satisfy in order to be able to apply the algorithm from the previous section to the problem of finding the longest p-segment, where p satisfies

$$p\,([a] \not\Leftarrow v \not\Leftarrow [b]) = (p\ v) \wedge (a \odot b).$$

All steps remain valid if we suppose that $\odot$ is symmetric, i.e.,

$$a \odot b = b \odot a.$$

For example, suppose we define the predicate *nispal* by

$$nispal\,([a] \not\Leftarrow x \not\Leftarrow [b]) = (nispal\ x) \wedge (a \neq b).$$

Then we can apply the derivation from Section 4 with $a = b$ replaced by $a \neq b$,

and have a linear-time algorithm for finding the longest segment which has absolutely no overlap with its reverse.

5.2. The Longest Two-Palindrome. In this section we sketch the derivation of a linear-time algorithm for finding the longest segment of a list which is the concatenation of two palindromes. Galil and Seiferas [14] give an algorithm on a two-way deterministic pushdown automaton for recognizing the language $\{x \mathbin{\#\!\!\!/} y \mid x \text{ and } y \text{ are palindromes of even length}\}$. A generalization of this problem is finding the longest segment of a list which is the concatenation of two palindromes. We give a linear-time RAM algorithm for this problem.

We can specify the problem of finding the longest segment that is the concatenation of two palindromes by

$$ltps = \uparrow_{\#\cdot\pi_2/} \cdot (tp \cdot \pi_2) \lhd \cdot split3s, \tag{80}$$

where the predicate tp (for "two-palindrome") is defined by

$$tp\ x = \exists y, z : x = y \mathbin{\#\!\!\!/} z \wedge ispal\ y \wedge ispal\ z. \tag{81}$$

The predicate tp is not c-slow, and hence the theory developed in Section 3 is not applicable to this problem. We use the list of maximal palindromes to solve this problem.

Maximal palindromes and two-palindromes are related in the following sense. Let x be a two-palindrome, that is, there are palindromes y and z such that $x = y \mathbin{\#\!\!\!/} z$. The palindromes y and z are contained in maximal palindromes y' and z' with the same center. In order to talk about centers we have to know the parts of the argument list in front of y, z, y', and z', so define $s = (yl, y)$, $t = (zl, z)$, $u = (yl', y')$, and $v = (zl', z')$. Here and throughout a pair of lists of which the second component is a maximal palindrome is called a maximal palindrome. Concerning the length of x, we have

$$\begin{aligned}
& \#x \\
= & \quad \{ x = y \mathbin{\#\!\!\!/} z \text{ is a two-palindrome} \} \\
& \#(y \mathbin{\#\!\!\!/} z) \\
= & \quad \{ \text{properties of } \mathbin{\#\!\!\!/}, \#, \text{ and } ctr \} \\
& 2 \times (ctr\ t - ctr\ s) \\
= & \quad \{ v \text{ and } u \text{ are the corresponding maximal palindromes} \} \\
& 2 \times (ctr\ v - ctr\ u).
\end{aligned}$$

Hence, to every two-palindrome x correspond one or more (depending on how many ways x can be split in two palindromes) pairs of maximal palindromes (u, v) such that $\#x = 2 \times (ctr\ v - ctr\ u)$. On the other hand, given two maximal palin-

dromes u and v which have some overlap, we can construct one or more lists x *satisfying tp*, all of which have length $2 \times (ctr\ v - ctr\ u)$. Two maximal palindromes u and v have *overlap* if $u\ W\ v$ holds. The relation W is defined by

$$(82)\quad (u_1, u_2)\ W\ (v_1, v_2) = (\#u_1) + (\#u_2) \geq (\#v_1) \wedge (\#v_1) + (\#v_2) \geq (\#u_1).$$

Given two maximal palindromes which have overlap of length m, we can construct $m + 1$ two-palindromes of length m.

From the above discussion it follows that instead of computing the longest segment which is the concatenation of two palindromes, we can compute two maximal palindromes u and v which have overlap, such that $ctr\ v - ctr\ u$ is as large as possible. Hence (80) transforms into

$$(83)\quad ltps = f\uparrow_g / W \lhd (Q\ x\ \mathsf{X}\ Q\ x),$$

where f is a function that returns a two-palindrome given a pair of maximal palindromes which have overlap, g is the function defined by

$$(84)\quad g = abs \cdot - \cdot ctr \times ctr,$$

where *abs* computes the absolute value of an integer, Q computes the list of all maximal palindromes, and X is the cross operator, see [3], which may be defined by

$$(85)\quad x\ \mathsf{X}\ y = \{(a, b) | a \in x \wedge b \in y\}.$$

Given a point p in the list (there are $n + 1$ points in a list of length n: a point in a list is a position between two elements or the position before or after the list), the largest pair under g of maximal palindromes which have overlap in point p is the pair (u, v) in which $u = (u_1, u_2)$ is the unique maximal palindrome with smallest center such that $\#u_1 \leq p \leq \#u_1 + \#u_2$, and $v = (v_1, v_2)$ is the unique maximal palindrome with greatest center such that $\#v_1 \leq p \leq \#v_1 + \#v_2$. For suppose the pairs of maximal palindromes (u, v) and (w, v), with $v >_{ctr} u$ and $v >_{ctr} w$, have overlap, and suppose $u <_{ctr} w$. Then $(u, v) >_g (w, v)$. Conversely, suppose the pairs of maximal palindromes (u, v) and (u, w), with $u <_{ctr} v$ and $u <_{ctr} w$, have overlap, and suppose $w <_{ctr} v$. Then $(u, w) <_g (u, v)$. This suggests we compute two lists. The first list, returned by the composition of functions $F \cdot Q$, contains, for every point p in the list, the maximal palindrome with smallest center, the second component of which contains p, in the list. The second list, returned by the composition of functions $G \cdot Q$, contains, for every point p in the list, the maximal palindrome with greatest center, the second component of which contains p. Since Q returns all maximal palindromes of a list, these lists are returned by functions composed with

function Q. So

$$= \underset{\text{proviso above}}{\uparrow_g / W \lhd (Q\ x \mathbin{\mathsf{X}} Q\ x)}$$

$$(\uparrow_g \mapsto e)\ (F\ Q\ x\ \Upsilon\ G\ Q\ x),$$

where Υ is the zip operator, see [3], defined for lists of equal length by

$$\square\ \Upsilon\ \square = \square,$$

$$(x \prec a)\ \Upsilon\ (y \prec b) = (x\ \Upsilon\ y) \prec (a, b), \tag{86}$$

and e is a fictitious element satisfying $e \uparrow_g a = a$ for all a. Functions F and G are left-reductions defined as follows. Let *dup* be a function that takes a natural number n and an element a and returns the list containing n elements a; so $(dup\ 3)\ c = [c, c, c]$.

$$F = \oplus \mapsto \square,$$

$$g = rev \cdot \oplus \mapsto \square \cdot rev,$$

where operators $\oplus$ and $\otimes$ are defined by

$$x \oplus a = \begin{cases} [a] & \text{if } x = \square, \\ x & \text{if } lt\ x \geq_h a, \\ x \prec ((dup\ (h\ a - h\ lt\ x))\ a) & \text{otherwise,} \end{cases}$$

$$h = + \cdot \# \times \#,$$

$$x \otimes a = \begin{cases} [a] & \text{if } x = \square, \\ x & \text{if } lt\ x \leq_j a, \\ x \prec ((dup\ (j\ lt\ x - j\ a))\ a) & \text{otherwise,} \end{cases}$$

$$j = \# \cdot \lessdot.$$

A (or all, depending on the definition of f) longest segment which is the concatenation of two palindromes can be computed using the algorithm

$$f\ (\uparrow_g \mapsto e)\ (F\ Q\ x\ \Upsilon\ G\ Q\ x), \tag{87}$$

which is a linear-time algorithm if we use the position in the list of a string instead of the string itself.

5.3. The Shortest All-Palindromes Partition. Finding the shortest partition of a list all of whose elements are palindromes is another problem for which we can

use the algorithm for finding maximal palindromes. This problem is related to the problem of finding the longest segment of a list which is the concatenation of two palindromes in the following way. Suppose we have computed the longest segment of a list which is the concatenation of n palindromes, for all n: $1 \leq n \leq \#x$. Obviously, if $n = \#x$, this longest segment is equal to x. The length of the shortest all-palindromes partition of the list x is equal to the smallest number n such that the longest segment of x which is the concatenation of n palindromes is equal to x.

A partition of a list is a set of lists of lists which yield the argument when flattened. The partitions of a list are computed by means of the function *parts*. A definition as a left-reduction of *parts* is given in [18]. We have, for example,

$$parts\ [a, b] = \{[[a, b]], [[a], [b]]\}.$$

The shortest all-palindromes partition is specified by

$$\downarrow_{\#}/ \cdot (all\ ispal) \lhd \cdot parts, \tag{88}$$

where the predicate *all p* is defined by

$$all\ p = \wedge / \cdot p*. \tag{89}$$

For example, the shortest all-palindromes partition of the string "abacab" is the partition into "a" and "bacab," and the shortest all-palindromes partition of the string "abacac" is the partition into "aba" and "cac."

The theory for partition problems developed in, for example, [25] and [18] is not applicable to our problem. However, if we compute some extra information, we can derive a left-reduction for our problem. Suppose we return, besides the shortest all-palindromes partition of a list x, the shortest all-palindromes partition of all elements in *inits x*, where function *inits* is defined by

$$inits = sort_{\#} \cdot \pi_1 * \cdot split2s.$$

Since $x = lt\ inits\ x$, the new specification reads

$$(\downarrow_{\#}/ \cdot (all\ ispal) \lhd \cdot parts) * \cdot inits. \tag{90}$$

We have that

$$(\downarrow_{\#}/ \cdot (all\ ispal) \lhd \cdot parts) * \cdot inits = \oplus \leftarrow\!\!\!\!\rightarrow e, \tag{91}$$

where $e = \{\square\}$, and the operator $\oplus$ is defined by

$$x \oplus a = x \prec (\downarrow_{\#}/ (all\ ispal) \lhd parts\ (x \prec a)). \tag{92}$$

If we also compute the longest tail palindrome and the maximal palindromes, we can determine the right-hand argument of $\lessdot$ in time linear in the length of the argument. The final algorithm requires quadratic time for its evaluation. This construction is a rather standard one in dynamic programming, and the details of the derivation are therefore omitted.

5.4. The Largest Palindromic Rectangle in a Matrix. The algorithm for finding palindromes in a list can be used to give an algorithm for finding palindromes in a matrix. The type of matrices has been given in [3] and [19]. The definition of $\#$ and *ispal* can be extended to matrices as follows. The function $\#$ computes the number of elements in a matrix. A matrix is called a palindrome (satisfies *ispal*) when all its columns and rows are palindromes.

The following informally described algorithm, the complexity of which is quadratic in the number of matrix elements, solves this problem. Apply the left-reduction which returns just all maximal palindromes (so not the longest palindromic segment) to every row and column, resulting in a matrix twice the width and height of the original matrix with, at every entry, the length of the longest row and column palindrome around that center. This can be done in linear time. Then, for every entry, we have to find the largest palindrome matrix around it. This can be done in linear time for every entry, and therefore we can obtain an algorithm which requires time quadratic in the number of matrix elements. The details of the algorithm are omitted; the precise formulation of the algorithm requires quite a number of definitions and a long derivation.

6. Conclusions. We have given a theory for the derivation of on-line algorithms. Two applications of this theory are a derivation of the algorithm for pattern-matching from Knuth, Morris, and Pratt, see [4], and a derivation of algorithm for finding the longest palindromic substring of a string.

We have derived an on-line linear-time RAM algorithm for finding the longest palindromic segment of a list. As a spin-off, we also find the positions of all maximal palindromes in a list. This algorithm improves on the algorithm given in [22], which determines just the initial palindromes of a list. Since every initial palindrome is by definition maximal, we find all initial palindromes in linear time. The idea of computing the list of maximal palindromes has been given before in [14]. However, instead of a derivation explaining the relation of the algorithm for finding palindromes with other segment problems, a difficult correctness proof of the algorithm is given there. The algorithm which returns the list of maximal palindromes can be used to solve some other problems, such as finding the shortest all-palindromes partition of a list.

A number of slightly different algorithms can be given to find the longest palindrome in a list. We chose to develop the one given in this article because it can be derived rather straightforwardly, in a fashion similar to the derivation of other segment problems, such as the algorithm from Knuth, Morris, and Pratt for pattern matching.

A slight (constant) increase in efficiency can be obtained in the computation of

the longest tail palindrome. It is not very difficult to prove that the filter predicate $(\leq_{\#} x)\cdot\pi_1$ occurring in the algorithm can be replaced by $(=_{\#} x)\cdot\pi_1$. The computation of the list of maximal palindromes is somewhat more difficult now. This results in an algorithm which requires fewer applications of ⓘ. This version has been presented in [17]. Because our algorithm satisfies the real-time predictability condition, see [13], our on-line RAM algorithm for finding the longest palindromic segment can be transformed, using a standard construction, into a real-time RAM algorithm. For a real-time Turing machine algorithm for finding palindromes the reader is referred to [10] and [12].

As noted in the Introduction, the longest palindrome in a list can be determined if all maximal palindromes have been computed. Hence it would have sufficed to compute just the maximal palindromes of a list. This results in an algorithm very similar to the one given here; instead of a left-reduction over lists it consists of a left-reduction over center positions.

Acknowledgments. Maarten Fokkinga, Lambert Meertens, and Jaap van der Woude made numerous useful suggestions, comments, and remarks about the presentation and derivation of the algorithm, for which I thank them.

Appendix. Proof of the Sliding Tails Theorem. In this appendix we give the remaining part of the proof of the Sliding Tails Theorem. We prove the equation

$$\uparrow_{\#\cdot\pi_2}/\cdot(\sqrt{p})*\cdot(\diamondsuit a)*\cdot split2s = (\ominus a)\cdot\uparrow_{\#\cdot\pi_2}/\cdot(\sqrt{p})*\cdot split2s. \tag{93}$$

Actually, we prove the following. Let (y, z) be $\uparrow_{\#\cdot\pi_2}/(p\cdot\pi_2)\triangleleft split2s\ x$. Then, by definition of $(\sqrt{p})$, $\uparrow_{\#\cdot\pi_2}/(\sqrt{p})*\ split2s\ x = (y, z, \square)$. We show that

$$\uparrow_{\#\cdot\pi_2}/(\sqrt{p})*(\diamondsuit a)*\ split2s\ x = (y, z, \square)\ominus a. \tag{94}$$

The proof of (94) uses the fact that predicate p is c-slow, in particular the fact that p has a derivative ∇ satisfying

$$p(x \prec a) = p\ t \wedge a\nabla_t s, \tag{95}$$

where $(s, t) = ((takef\ c)\ x, (dropf\ c)\ x)$. Furthermore, it uses a crippled version of the equality

$$p\triangleleft\cdot tails = p\triangleleft\cdot tails\cdot\uparrow_{\#}/\cdot p\triangleleft\cdot tails, \tag{96}$$

which states that the set of all tails ($tails = \gg*\cdot split2s$) satisfying p equals the set of all tails satisfying p of the longest tail satisfying p. To replace *tails* by *split2s* in (96), let (y, z) be $\uparrow_{\#\cdot\pi_2}/(p\cdot\pi_2)\triangleleft split2s\ x$. Then (96) is transformed into

$$(p\cdot\pi_2)\triangleleft split2s\ x = (p\cdot\pi_2)\triangleleft((y\prec) \times id)*\ split2s\ z. \tag{97}$$

Finally, at the place in the calculation where this equality is applied we have the expression $(p \cdot (dropf\ c) \cdot \pi_2) \lhd$ instead of $(p \cdot \pi_2) \lhd$. This is the cause of an extra complication in the statement of the equality. Define the binary operator $\rightsquigarrow$ ("shift"), which takes a natural number n and a pair of lists (x, y) and prepends the tail of length n of x (or x itself if $\# x \leq n$) to y, by

$$0 \rightsquigarrow (x, y) = (x, y),$$

$$n + 1 \rightsquigarrow (x, y) = \begin{cases} (x, y) & \text{if } x = \Box, \\ n \rightsquigarrow (it\ x, [lt\ x] \nVdash y) & \text{otherwise.} \end{cases} \tag{98}$$

Let (u, v) be $c \rightsquigarrow (y, z)$, then (97) is transformed into the following equation:

$$(p \cdot (dropf\ c) \cdot \pi_2) \lhd split2s\ x = (p \cdot (dropf\ c) \cdot \pi_2) \lhd ((u \nVdash) \times id) * split2s\ v. \tag{99}$$

The proof of (93) is by calculation.

$$\begin{aligned} & \uparrow_{\# \cdot \pi_2} / (\surd p) * (\Diamond a) * split2s\ x \\ = & \quad \text{definition of } \surd p, \text{ definition of guard} \\ & \uparrow_{\# \cdot \pi_2} / (p \cdot \pi_2) \lhd (\propto \Box) * (\Diamond a) * split2s\ x \\ = & \quad \text{map distributivity, map-filter swap} \\ & \uparrow_{\# \cdot \pi_2} / ((\propto \Box) \cdot (\Diamond a)) * (p \cdot \pi_2 \cdot (\propto \Box) \cdot (\Diamond a)) \lhd split2s\ x. \end{aligned}$$

Consider the expression $p \cdot \pi_2 \cdot (\propto \Box) \cdot (\Diamond a)$. To apply (99), this composition of functions is rewritten as follows:

$$\begin{aligned} & p\ \pi_2\ (\propto \Box)\ (\Diamond a)\ x \\ = & \quad \pi_2^3 \cdot (\propto \Box) = \pi_2^2, \text{ superscripts are omitted} \\ & p\ \pi_2\ (\Diamond a)\ x \\ = & \quad \pi_2 \cdot (\Diamond a) = (\nVdash a) \cdot \pi_2 \\ & p\ (\nVdash a)\ \pi_2\ x \\ = & \quad (95) \\ & a \nabla_t s \wedge p\ t, \end{aligned}$$

where $(s, t) = ((takef\ c)\ (\pi_2\ x), (dropf\ c)\ (\pi_2\ x))$. Abbreviate function

$$a \nabla_{(dropf\ c)} (takef\ c) \cdot \pi_2$$

to (Δa). Let (u, v) be $c \rightsquigarrow (y, z)$. Then

$$
\begin{aligned}
& \uparrow_{\#\cdot\pi_2}/\,((\propto\Box)\cdot(\Diamond a))*\,(p\cdot\pi_2\cdot(\propto\Box)\cdot(\Diamond a))\triangleleft\ split2s\ x \\
= \quad & \text{proviso above} \\
& \uparrow_{\#\cdot\pi_2}/\,((\propto\Box)\cdot(\Diamond a))*\,((\Delta a)\wedge(p\cdot(dropf\ c)\cdot\pi_2))\triangleleft\ split2s\ x \\
= \quad & (p\wedge q)\triangleleft = p\triangleleft\cdot q\triangleleft \\
& \uparrow_{\#\cdot\pi_2}/\,((\propto\Box)\cdot(\Diamond a))*\,(\Delta a)\triangleleft\,(p\cdot(drof\ c)\cdot\pi_2)\triangleleft\ split2s\ x \\
= \quad & (99) \\
& \uparrow_{\#\cdot\pi_2}/\,((\propto\Box)\cdot(\Diamond a))*\,(\Delta a)\triangleleft\,(p\cdot(dropf\ c)\cdot\pi_2)\triangleleft\,((u\not\#)\times id)*\ split2s\ v \\
= \quad & \text{above steps in reverse order} \\
& \uparrow_{\#\cdot\pi_2}/\,(p\cdot\pi_2)\triangleleft\,((\propto\Box)\cdot(\Diamond a))*\,((u\not\#)\times id)*\ split2s\ v \\
= \quad & \text{moving } ((u\not\#)\times id)* \text{ to the left} \\
& ((u\not\#)\times id)\uparrow_{\#\cdot\pi_2}/\,(p\cdot\pi_2)\triangleleft\,((\propto\Box)\cdot(\Diamond a))*\ split2s\ v.
\end{aligned}
$$

To move $((u\not\#)\times id)*$ to the left we have to show that $((u\not\#)\times id)$ commutes with $(\Diamond a)$ and $(\propto\Box)$, that $p\cdot\pi_2\cdot((u\not\#)\times id) = p\cdot\pi_2$, and, finally, that $((u\not\#)\times id)$ is $(\uparrow_{\#\cdot\pi_2}, \uparrow_{\#\cdot\pi_2})$-fusable. The proofs of these facts are omitted. Since (u, v) equals $c \rightsquigarrow (y, z)$ it follows that we have expressed $\uparrow_{\#\cdot\pi_2}/\,(\sqrt{p})*\,(\Diamond a)*\ split2s\ x$ in terms of (y, z). Define operator $\ominus$ by

$$(100)\qquad (y, z, \Box)\ominus a = ((u\not\#)\times id)\uparrow_{\#\cdot\pi_2}/\,(p\cdot\pi_2)\triangleleft\,((\propto\Box)\cdot(\Diamond a))*\ split2s\ v,$$

where $(u, v) = c \rightsquigarrow (y, z)$. By distinguishing the cases from the definition of operator $\ominus$ in (59) we can show that operator $\ominus$ defined in (100) is equal to operator $\ominus$ defined in (59). Thus we obtain (93).

References

[1] A. Apostolico and Z. Galil, editors. *Combinatorial Algorithms on Words*, NATO ASI Series F, volume 12. Springer-Verlag, Berlin, 1985.

[2] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. NATO ASI Series F, volume 36. Springer-Verlag, Berlin, 1987.

[3] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, pages 151–216. NATO ASI Series F, volume 55. Springer-Verlag, Berlin, 1989.

[4] R. S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[5] R. S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[6] S. N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, 18(4):349–365, 1969.

[7] M. Crochemore and W. Rytter. Parallel computations on strings and arrays. In C. Choffrut and T. Lengauer, editors, *Proceedings of the 7th Annual Symposium on Theoretical Aspects of Computer Science*, pages 109–125. Lecture Notes in Computer Science, Vol. 415. Springer-Verlag, Berlin, 1990.

[8] P. C. Fischer and M. S. Paterson. String matching and other products. In R. M. Karp, editor, *SIAM–AMS Proceedings on Complexity of Computation*, volume 7, pages 113–126, 1974.

[9] M. M. Fokkinga. Law and Order in Algorithmics. Ph.D. thesis, Twente University, 1992.

[10] Z. Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proceedings of the Eighth Annual Symposium on Theory of Computing*, pages 161–173, 1976.

[11] Z. Galil. Two fast simulations which imply some fast string matching and palindrome-recognition algorithms. *Information Processing Letters*, 4:85–87, 1976.

[12] Z. Galil. Palindrome recognition in real time by a multitape Turing machine. *Journal of Computers and Systems Sciences*, 16:140–157, 1978.

[13] Z. Galil. String matching in real time. *Journal of the ACM*, 28(1):134–149, 1981.

[14] Z. Galil and J. Seiferas. A linear-time on-line recognition algorithm for "Palstar." *Journal of the ACM*, 25(1):102–111, 1978.

[15] J. A. Goguen. Memories of ADJ. *Bulletin of the EATCS*, 39:97–102, 1989.

[16] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[17] J. Jeuring. Finding palindromes. In *Proceedings SION Computing Science in the Netherlands*, pages 123–140, 1988.

[18] J. Jeuring. Algorithms from theorems. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, Amsterdam, 1990.

[19] J. Jeuring. The derivation of hierarchies of algorithms on matrices. In B. Möller, editor, *Constructing Programs from Specifications*, pages 9–32. North-Holland, Amsterdam, 1991.

[20] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1978.

[21] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[22] G. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346–351, 1975.

[23] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. CWI Monographs, volume 1. North-Holland, Amsterdam, 1986.

[24] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, 1990. To appear in *Formal Aspects of Computing.*

[25] O. de Moor. List partitions. To appear in *Formal Aspects of Computing.*

[26] J. I. Seiferas. Iterative arrays with direct central control. *Acta Informatica*, 8:177–192, 1977.

[27] A. O. Slisenko. Recognizing a symmetry predicate by multihead Turing machines with input. In V. P. Orverkov and N. A. Sonin, editors, *Proceedings of the Steklov Institute of Mathematics*, number 129, pages 25–208, 1973.

[28] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, volume 14, pages 1–11, 1973.

[29] A. C. Yao. A Lower Bound to Palindrome Recognition by Probabilistic Turing Machines. Technical Report STAN-CS-77-647, Computer Science Department, Stanford University, 1977.