Algorithms from Theorems

Johan Jeuring

RUU-CS-90-3
January 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Algorithms from Theorems

Johan Jeuring

Technical Report RUU-CS-90-3
January 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Algorithms from Theorems

Johan Jeuring*
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
(jt@cwi.nl)

Abstract

In this paper we show how algorithms are derived from their specification in the Bird–Meertens formalism. The Bird–Meertens formalism is a programming methodology which provides a concise functional notation for algorithms and for every data structure a promotion theorem for proving equalities of functions. High-level specifications (which usually are clearly correct, but inefficient functional algorithms) are transformed, using the promotion theorem, into efficient algorithms. Here, we apply the method to problems on permutations (such as sorting), subsequences and partitions. For each of these enumerator functions we prove one theorem which is used in the derivation of algorithms for some concrete examples.

1987 CR Categories: D.1.1, D.2.1, F.3.1, I.2.2, I.2.8
1985 Math. Subj. Class.: 69D11, 69D24, 69K12, 69K18

Note: To appear in the *Proceedings of the IFIP TC2/WG2.2/WG2.3 Working Conference on Programming Concepts and Methods, Sea of Gallilee, Israel, April 2-5, 1990.*

1 Introduction

The purpose of this paper is to show how algorithms are derived in the Bird–Meertens formalism. The Bird–Meertens formalism is a framework in which program construction is viewed as a mathematical activity. It provides a concise functional notation for algorithms, and for every data structure a promotion theorem for proving equalities of functions. Starting with a clearly correct, but possibly very inefficient algorithm, we successively apply (possibly conditional) algebraic identities (instances of the promotion theorem or properties of the constituents) until an efficient algorithm results. Part of the formalism is discussed in this article, for more extensive accounts the reader is referred to [15], [3], [4], [16], and [1].

* This research has been supported by the Dutch organization for scientific research under project-nr. NF 62.518.

We derive algorithms for problems on lists. Examples of such problems are sorting a list, finding the longest upsequence of a list, and finding the occurrences of a given pattern in a list. Algorithms for these problems are well known. Therefore, the derivations of some of these algorithms serve as good examples of the programming methodology we apply.

The specifications we consider are compositions of homomorphisms with enumerator functions. An example is the specification

$$\downarrow_{\#}/ \cdot (\mathsf{all}\ \mathit{ascending}) \lhd \cdot \mathsf{parts}\ ,$$

where parts is a function enumerating all partitions of a list as a set, and $\downarrow_{\#}/ \cdot (\mathsf{all}\ \mathit{ascending}) \lhd$ is a homomorphism defined on sets, which selects the shortest element all of whose components are ascending. The enumerator functions are inverse images of homomorphisms. The inverse image of a function $f : \beta \leftarrow \alpha$ is defined by

$$\mathsf{Inv}(f : \beta \leftarrow \alpha)\ y = \{x : \alpha | f\ x = y\}\ .$$

For example, parts is the inverse image of the flattening homomorphism $+\!\!+/$, which flattens a list of nonempty lists of elements of some type α to a list of elements of type α. Formally,

$$\mathsf{parts} = \mathsf{Inv}(+\!\!+/ : \alpha* \leftarrow \alpha+*)\ ,$$

where $\alpha+$ is the type of nonempty lists, and $\beta*$ is the type of possibly empty lists. Given an inverse image definition, we can derive recursive equations which characterize the defined function. In this paper we start our derivations with these recursive characterizations. The inverse image, together with methods for deriving recursive equations from inverse images, is discussed in [6].

For three enumerator functions we prove a theorem which states the conditions the components of a homomorphism have to satisfy in order to obtain a left–reduction (an efficient algorithm) for the composition of the homomorphism and the enumerator function. These conditions can be derived in a straightforward way. This derivation is discussed in detail for the enumerator functions subs computing all subsequences of a list, and perms computing all permutations of a list. For parts we just state the result of the derivation.

We give three examples in which we apply the developed theory. The algorithms we obtain can easily be translated into a functional language such as Miranda[1]. From the examples we work out it will become clear that programming in the Bird–Meertens formalism is not merely applying the appropriate theorems. Inventive steps remain to be made, especially when a given problem does not satisfy the conditions of the theorem. In that case we have to adjust or generalize the problem in a form such that the conditions of the theorem are satisfied. It is difficult to derive these generalization steps (an attempt is made in [8]), but usually there seems to be not much choice.

In [4] Horner's rule is proved. Horner's rule is a theorem which states that the composition of a homomorphism with the function tails is a left–reduction, provided some conditions hold. Here we give a method for deriving these conditions for arbitrary enumerator functions. Applications of this method can also be found in [13] and [12]. A different method, stressing the importance of inverse images in program synthesis, with which similar results can be obtained is presented in [6].

[1] 'Miranda' is a trademark of Research Software Ltd

This paper is organized as follows. In Section 2 we give a brief introduction to the Bird–Meertens formalism. In Section 3, 4 and 5 we discuss, respectively, subsequences, permutations and partitions. We draw some conclusions in Section 6.

2 Preliminaries

In this section we introduce the basic notions and definitions used in the subsequent sections. In the first subsection we briefly describe the notational conventions for functions we use. Two important concepts in the Bird–Meertens formalism are the notions of homomorphism and promotion. For every data structure, homomorphisms are defined and a promotion theorem is given. This process is described in detail in [14]. Homomorphisms on data structures in the Boom-hierarchy, such as sets, bags, lists and binary trees, are introduced in the second subsection, together with some widely used examples of homomorphisms, such as map and reduction. At the end of this subsection we present the promotion theorem for these data structures. In the third subsection we introduce snoc–lists and left–reductions.

2.1 Functions

Typical names of functions are f, g and h. Function composition is denoted by a small dot $\cdot$, which is associative. So the composition of f and g is written as $f \cdot g$. Function application is denoted by white space. So the application of f to an argument a is written as $f\ a$. Function application associates to the right, i.e., we have

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

The type of a function from type α to type β is denoted by $\beta \leftarrow \alpha$. The argument-types of n-ary functions are tuples, the elements of which are separated by two vertical bars. So, for example, a binary operator $\oplus$ will have type $\gamma \leftarrow \alpha \| \beta$.

Binary operators will often be written in infix notation. Typical names of binary operators are $\oplus$, $\odot$, and $\otimes$. They can be partially parametrized, i.e., if $\oplus$ is a binary operator of type $\gamma \leftarrow \alpha \| \beta$, we consider the expression $(a \oplus)$ to be a unary function of type $\gamma \leftarrow \beta$, and similarly for $(\oplus b)$. These parametrized operators are also known as 'sections'.

The notation $x \leq_f y$ is used to express that $(f\ x) \leq (f\ y)$, and similarly for $=$, $>$, etc.

Given two functions $f : \beta \leftarrow \alpha$ and $g : \gamma \leftarrow \alpha$, we define the shared composition of f and g, denoted by (f, g), by

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

The shared composition of n functions is defined similarly on n–tuples. Functions can also be combined using parallel composition. Given functions $f : \beta \leftarrow \alpha$ and $g : \delta \leftarrow \gamma$, the parallel composition of f and g, written $(f \| g)$, is defined by

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

The functions π_1 and π_2 denote the projection onto the first respectively the second component of a pair. Similar definitions can be given for projections on components of triples etc.

2.2 The Boom-hierarchy

The recursive data structure of sets over some base type α, denoted by $\alpha\dagger$, is introduced by means of the following three constructor rules:

$$\frac{}{1_\cup \in \alpha\dagger} \qquad \frac{a \in \alpha}{\{a\} \in \alpha\dagger} \qquad \frac{\begin{array}{c} x \in \alpha\dagger \\ y \in \alpha\dagger \end{array}}{x \cup y \in \alpha\dagger}$$

where $1_\cup$ is the unit of $\cup$, and $\cup$ is associative, commutative and idempotent. The data structure set is one of the four data structures in the Boom-hierarchy. The Boom-hierarchy, described in [15], consists of four different data structures: trees, lists, bags, and sets. These data structures are obtained from the above scheme for $\alpha\dagger$ by varying the laws satisfied by $\cup$. If $\cup$ satisfies no laws, the above scheme leads to trees with information at the leaves. If $\cup$ is associative we obtain lists ($1_\cup$ and $\cup$ are then written as $1_{+\!\!+}$ respectively $+\!\!+$), and if $\cup$ is associative and commutative we obtain bags (the empty bag is written as $1_\uplus$, and bag union as $\uplus$). In the remaining part of this paper we give definitions and prove theorems for the data structure set. For each of the three other data structures in the Boom-hierarchy we can give similar definitions and theorems.

Given a recursive data structure, homomorphisms on this data structure can be described systematically. By definition, a function h defined on sets is a homomorphism if there exist an associative, commutative and idempotent operator $\oplus$, a function f, and a value e such that

$$\begin{array}{lcl} h\, 1_\cup & = & e \\ h\, \{a\} & = & f\, a \\ h\, (x \cup y) & = & (h\, x) \oplus (h\, y)\,. \end{array}$$

Since $1_\cup$ is the unit of $\cup$, it follows that e should be the unit of $\oplus$ on the range of h. If such a unit element does not exist, we may introduce a fictitious element (see [15]) with the property that it is the unit of $\oplus$. We give the types of the functions and operators involved. If f has type $\beta \leftarrow \alpha$ and $\oplus$ has type $\beta \leftarrow \beta \| \beta$, then h has type $\beta \leftarrow \alpha\dagger$.

It is a well known fact that homomorphisms on sets can be written as the composition of reductions and maps, which are defined as follows. The map operator $*$ takes as arguments a function and a set and returns a set consisting of the original elements to which the function is applied. More precisely, if $f : \beta \leftarrow \alpha$, then $f* : \beta\dagger \leftarrow \alpha\dagger$ is defined by

$$\begin{array}{lcl} f*\, 1_\cup & = & 1_\cup \\ f*\, \{a\} & = & \{f\, a\} \\ f*\, (x \cup y) & = & (f*\, x) \cup (f*\, y)\,. \end{array}$$

The value 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 $\oplus : \alpha \leftarrow \alpha \| \alpha$, then $\oplus/ : \alpha \leftarrow \alpha\dagger$ is defined by

$$\begin{array}{lcl} \oplus/\, 1_\cup & = & 1_\oplus \\ \oplus/\, \{a\} & = & a \\ \oplus/\, (x \cup y) & = & (\oplus/\, x) \oplus (\oplus/\, y)\,. \end{array}$$

where $1_\oplus$ is the, possibly fictitious, unit element of $\oplus$.

An example of a widely used homomorphism is the filter operator $\lhd$, which takes a predicate (i.e. a boolean function) and a set and retains the elements satisfying the predicate in a set, so if $p : \mathsf{bool} \leftarrow \alpha$, then $p\lhd : \alpha\dagger \leftarrow \alpha\dagger$ is defined by :

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

where $\hat{p}\ a = \{a\}$ if $p\ a$ holds and $\hat{p}\ a = 1_\cup$ otherwise. For example, $odd\lhd\ \{3,4,5\} = \{3,5\}$. In general we have for homomorphisms h:

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

for some operator $\oplus$ and function f, a fact expressed by the Homomorphism Lemma from [15].

We introduce two operators which are used frequently in the subsequent sections.

The operator $\uparrow_f$, where f is of type $\beta \leftarrow \alpha$, where β is totally ordered, is a binary operator of type $\alpha \leftarrow \alpha \| \alpha$. It is defined by

$$x \uparrow_f y = \begin{array}{ll} x & \text{if } x <_f y \\ y & \text{if } y <_f x . \end{array}$$

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 zip operator, written Υ, is an operator which takes an operator and two tuples or lists of equal length, and produces a tuple or a list of the same length consisting of the corresponding elements of the arguments combined with the operator, thus:

$$(x_0, \ldots, x_n)\ \Upsilon_\oplus\ (y_0, \ldots, y_n) = (x_0 \oplus y_0, \ldots, x_n \oplus y_n) .$$

We now come to the second important notion of the Bird–Meertens formalism, promotion. Every data structure has its own promotion theorem. Promotion provides a means for proving equalities of functions that avoids the application of induction in the development of algorithms. Inductive arguments tend to be tedious and are less elegant than proofs using promotion. Already in 1975, this was one of the main motivations of Goguen to introduce initiality, see [11]. Before we give the theorem, we first define promotability.

Definition 1 (**$(\oplus,\otimes)$-promotability**) *A function $f : \beta \leftarrow \alpha$ is $(\oplus,\otimes)$-promotable for associative, commutative and idempotent operators $\oplus : \alpha \leftarrow \alpha \| \alpha$ and $\otimes : \beta \leftarrow \beta \| \beta$ if and only if*

$$f\ (x \oplus y) = (f\ x) \otimes (f\ y)$$
$$f\ 1_\oplus = 1_\otimes .$$

We have the following theorem, the proof of which (by structural induction or using the uniqueness property of homomorphisms) can be found in [15], [1], and [14].

Theorem 2 (promotion) *A function $f : \beta \leftarrow \alpha$ is $(\oplus,\otimes)$-promotable if and only if*

$$f \cdot \oplus/ = \otimes/ \cdot f* .$$

2.3 Snoc–lists

Snoc–lists over base type α, denoted by $\alpha\wr$, are introduced by means of two constructor rules:

$$\frac{}{[\,] \in \alpha\wr} \qquad \frac{\begin{array}{c} a \in \alpha \\ x \in \alpha\wr \end{array}}{x \mathbin{-\!\!\prec} a \in \alpha\wr}$$

The difference with the lists from the Boom-hierarchy (also known as join–lists) is that snoc–lists are constructed from left to right whereas join–lists are constructed symmetrically. A function $h : \beta \leftarrow \alpha\wr$ is a homomorphism on snoc–lists if there exists an operator $\oplus : \beta \leftarrow \beta \| \alpha$ and a value $e : \beta$ such that

$$\begin{array}{lcl} h\,(x \mathbin{-\!\!\prec} a) & = & (h\,x) \oplus a \\ h\,[\,] & = & e\,. \end{array}$$

The usual name for homomorphism on snoc–lists is left–reduction, and we will use this terminology in the sequel. The function h defined above is written as $\oplus \mathbin{/\!\!-} e$. As an example, the length of a list, denoted by $\#$, is computed by the left–reduction $((+1) \cdot \pi_1) \mathbin{/\!\!-} 0$. Another example is the concatenation operator $\mathbin{-\!\!\prec\!\!\prec}$ defined by $x \mathbin{-\!\!\prec\!\!\prec} y = (\mathbin{-\!\!\prec} \mathbin{/\!\!-} x)\ y$. Note that $\mathbin{-\!\!\prec\!\!\prec}$ is associative. Again, we have a promotion theorem for this data structure.

Theorem 3 (snoc–lists promotion) *Suppose* $f : \gamma \leftarrow \beta$, $\oplus : \beta \leftarrow \beta \| \alpha$, *and* $\otimes : \gamma \leftarrow \gamma \| \alpha$ *satisfy*

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

and define $u = h\,e$. *Then*

$$f \cdot \oplus \mathbin{/\!\!-} e = \otimes \mathbin{/\!\!-} u\,.$$

From the proof of this theorem (which can be found in [14]) it follows that we may weaken the requirement $f\,(x \oplus a) = (f\,x) \otimes a$. It suffices to require this equality for x in the range of $\oplus \mathbin{/\!\!-} e$.

3 Subsequences

In this section we show conditions under which the composition of a homomorphism with the function **subs** computing all subsequences of a list is a left–reduction. This is done by means of applying (possibly conditional) algebraic identities. Together with an algorithm we derive conditions under which the equality of the specification and the algorithm holds. We apply the resulting theory to an example in which we give a linear–time algorithm for a variant of the Zero–One Knapsack problem.

The function **subs**, computing all subsequences of a list, is defined as the following inverse image.

$$\mathsf{subs} = (+\!\!+/ \cdot \pi_1)* \cdot \mathsf{Inv}(+\!\!+/ \cdot \Upsilon_{+\!\!+})\,.$$

It is recursively characterized by

$$\begin{array}{lcl} \mathsf{subs} & : & \alpha\wr\dagger \leftarrow \alpha\wr \\ \mathsf{subs}\,[\,] & = & \{[\,]\} \\ \mathsf{subs}\,(x \mathbin{-\!\!\prec} a) & = & (\mathsf{subs}\,x) \cup ((\mathbin{-\!\!\prec} a)*\,\mathsf{subs}\,x)\,. \end{array}$$

It follows that **subs** is a left–reduction $\oplus \leftarrow e$, where $e = \{[\,]\}$ and $\oplus$ is defined by $x \oplus a = x \cup ((\prec a)* x)$.

The generic specification of the problems we consider is

$$\otimes/ \cdot g* \cdot \mathbf{subs}\,,$$

where $\otimes$ and g are arbitrary. The number of subsequences is exponential in the length of the list, and therefore the straightforward implementation of our specification requires exponential time. In order to obtain a homomorphism on snoc–lists, or equivalently, a left–reduction $\odot \leftarrow u$ for this specification, conditions will have to be imposed upon $\otimes$ and g. If $\odot$ requires constant time, the left–reduction we derive requires linear time when implemented.

Since **subs** is a left–reduction, we can apply the snoc–lists promotion theorem with $h = \otimes/ \cdot g*$. For that purpose, we have to compute $h\ e$ and we have to find an operator $\odot$ such that $h\ (x \oplus a) = (h\ x) \odot a$, where e and $\oplus$ are the components of the left–reduction given for **subs** above. First we compute $h\ e$.

$$\begin{array}{ll} & h\ e \\ = & \quad \text{definition of } h \text{ and } e \\ & \otimes/\ g*\ \{[\,]\} \\ = & \quad \text{definition of homomorphism} \\ & g\ [\,]\,. \end{array}$$

A definition of $\odot$ which satisfies the above requirement is synthesized as follows.

$$\begin{array}{ll} & h\ (x \oplus a) \\ = & \quad \text{definition of } h \text{ and } \oplus \\ & \otimes/\ g*\ (x \cup ((\prec a)*\ x)) \\ = & \quad \text{definition of homomorphism} \\ & (\otimes/\ g*\ x) \otimes (\otimes/\ g*\ (\prec a)*\ x) \\ = & \quad \text{definition of } h \\ & (h\ x) \otimes ((\otimes/ \cdot g* \cdot (\prec a)*)\ x)\,. \end{array}$$

The left-hand argument of $\otimes$ is of the required form. We proceed with the right-hand argument of $\otimes$, promoting the part $(\prec a)*$ to the left. This is the point at which conditions are imposed upon $\otimes$ and g.

$$\begin{array}{ll} & \otimes/ \cdot g* \cdot (\prec a)* \\ = & \quad \text{map distributivity (promotion theorem)} \\ & \otimes/ \cdot (g \cdot (\prec a))* \\ = & \quad \text{first requirement} \\ & \otimes/ \cdot ((\oslash a) \cdot g)* \\ = & \quad \text{map distributivity} \\ & \otimes/ \cdot (\oslash a)* \cdot g* \\ = & \quad \text{promotion theorem, second requirement} \\ & (\oslash a) \cdot \otimes/ \cdot g*\,. \end{array}$$

During the derivation we encountered the following requirements. First, we require $g \cdot (-\!\!\prec a) = (\oslash a) \cdot g$, i.e., the function g has to satisfy $g\,(x -\!\!\prec a) = (g\,x) \oslash a$ for some operator $\oslash$. By definition of left–reduction, this implies that g is a left–reduction $\oslash \mathbin{/\!\!\rightarrow} u$ for some value u. Second, the conditions of the promotion theorem, Theorem 2 in Section 2, have to be satisfied, that is, $(\oslash a)$ must be $(\otimes, \otimes)$–promotable. If these requirements are satisfied, and if we define $\odot$ by

$$x \odot a = x \otimes (x \oslash a)\,,$$

then we obtain by the snoc–lists promotion theorem

$$\otimes/ \cdot g* \cdot \mathsf{subs} = \odot \mathbin{/\!\!\rightarrow} g\,[\,]\,.$$

Since we required g to be a left–reduction $\oslash \mathbin{/\!\!\rightarrow} u$, we have that $g\,[\,] = u$. The above derivation proves the following theorem.

Theorem 4 (subs–promotion) *Let h be a homomorphism $\otimes/ \cdot g*$ defined on sets, and g a left–reduction $\oslash \mathbin{/\!\!\rightarrow} u$ such that $(\oslash a)$ is $(\otimes, \otimes)$–promotable for all a. Then*

$$h \cdot \mathsf{subs} = \odot \mathbin{/\!\!\rightarrow} u\,,$$

where the operator $\odot$ is defined by

$$x \odot a = x \otimes (x \oslash a)\,.$$

In fact, one of the requirements of the theorem is stronger than necessary. From the derivation one can see that the requirement that $(\oslash a)$ is $(\otimes, \otimes)$–promotable may be restricted to the range of $g*$.

A derivation of an algorithm for the problem of finding the longest upsequence of a list is given in [8]. We give another example.

Example 5 A successful treasure-digger comes across another treasure. Since she likes to have a convenient trip home, all she has with her is a knapsack of a certain small volume V. The treasure-digger wants to find a subset of the treasure which fits in her knapsack and has the largest value. This problem is known as the Zero-One Knapsack problem or the Zero-One Integer Programming problem, see for example [10]. We derive a linear–time algorithm (for fixed V) solving the problem where every element of the treasure has volume equal to a natural number; it is well-known that without these restrictions this problem is NP–complete.

The treasure is represented by a list of elements, where elements are pairs of natural numbers, modelling the value and the volume of the element respectively. The subset of treasure which fits in the knapsack and has the largest value is a subsequence of the given list, the sum of the second components of which does not exceed V, and the sum of the first components of which is maximal. Hence we can specify our problem by

$$\uparrow_{\mathsf{val}}/ \cdot ((\leq V) \cdot \mathsf{vol}) \lhd \cdot \mathsf{subs}\,,$$

where vol and val are both left–reductions, defined by

$$\begin{array}{lcl} \mathsf{vol} & = & (+ \cdot (\mathsf{id} \| \pi_2)) \mathbin{/\!\!\rightarrow} 0 \\ \mathsf{val} & = & (+ \cdot (\mathsf{id} \| \pi_1)) \mathbin{/\!\!\rightarrow} 0\,. \end{array}$$

We abbreviate $\uparrow_{\mathsf{val}}$ to $\uparrow$.

In order to apply the **subs**–promotion theorem to our example, we have to rewrite it in the form of a composition of a homomorphism and the function **subs**. In general, we have for arbitrary operator $\oplus$ and predicate q

$$
\begin{array}{ll}
 & \oplus/ \cdot q \triangleleft \\
= & \quad \text{definition of filter} \\
 & \oplus/ \cdot \cup/ \cdot \hat{q}* \\
= & \quad \text{promotion theorem} \\
 & \oplus/ \cdot (\oplus/)* \cdot \hat{q}* \\
= & \quad \text{map distributivity} \\
 & \oplus/ \cdot (\oplus/ \cdot \hat{q})* .
\end{array}
$$

This expression is of the appropriate form.

We try to verify the conditions of the **subs**–promotion theorem. First, we have to find a left–reduction $\oslash \not\rightarrow u$ such that $\uparrow/ \cdot \hat{p} = \oslash \not\rightarrow u$, where p is an abbreviation for $(\leq V) \cdot \mathsf{vol}$. Since $\hat{q}\, x = \{x\}$ if $q\, x$ holds, and $1_{\cup}$ otherwise,

$$
\oplus/\, \hat{q}\, x = \begin{array}{ll} x & \text{if } q\, x \\ 1_{\oplus} & \text{otherwise} . \end{array}
$$

for all predicates q and operators $\oplus$. If furthermore q is *prefix–closed*, that is, for all lists x and y, q satisfies

$$
q\, (x \mathbin{-\!\!\!\ll} y) \Rightarrow q\, x ,
$$

then $\oplus/ \cdot \hat{q}$ is a left–reduction $\oslash \not\rightarrow [\,]$, where $x \oslash a = x \mathbin{-\!\!\ll} a$ if $x \neq 1_{\oplus}$ and $q\, (x \mathbin{-\!\!\ll} a)$ holds, and $x \oslash a = 1_{\oplus}$ otherwise. Since negative volumes do not exist, it is clear that $\mathsf{vol}\, (x \mathbin{-\!\!\ll} a) \leq V \Rightarrow \mathsf{vol}\, x \leq V$, and hence that p is prefix–closed. Hence there exists a left–reduction $\oslash \not\rightarrow [\,]$ equal to $\uparrow/ \cdot \hat{p}$, where $\oslash$ is defined by

$$
x \oslash a = \begin{array}{ll} x \mathbin{-\!\!\ll} a & \text{if } (x \neq 1_{\uparrow}) \wedge ((\mathsf{vol}\, (x \mathbin{-\!\!\ll} a)) \leq V) \\ 1_{\uparrow} & \text{otherwise} . \end{array}
$$

The second condition of the **subs**–promotion theorem requires the section $(\oslash a)$ to be $(\uparrow, \uparrow)$–promotable, that is, we have to show that $1_{\uparrow} \oslash a = 1_{\uparrow}$, which holds by definition, and that

$$
(x \uparrow y) \oslash a = (x \oslash a) \uparrow (y \oslash a) .
$$

However, this equation is not true. This can be seen by taking $y >_{\mathsf{val}} x$, $\mathsf{vol}\, (y \mathbin{-\!\!\ll} a) > V$, and $\mathsf{vol}\, (x \mathbin{-\!\!\ll} a) \leq V$. In this case, the left–hand side expression of the desired equation equals $1_{\uparrow}$ and the right–hand side $x \mathbin{-\!\!\ll} a$.

From the above we conclude that the problem as we stated it cannot be solved with the **subs**–promotion theorem. We have to find a generalization of our problem such that the theorem can be applied. In the solution for the problem of finding the longest upsequence, see [8], the generalization step is to consider subsequences of specific lengths. Here we use a different generalization step, an application of the 'tupling strategy'.

Suppose we compute, instead of the most precious subsequence with volume not exceeding V, the most precious subsequences with volume equal to i for all $i : 0 \leq i \leq V$. The most precious subsequence with volume not exceeding V can be computed

easily given these $V+1$ subsequences. This generalization step is similar to the one applied in one of the examples in [13], where we consider, instead of **subs**, a function enumerating specific subtrees of binary labelled trees.

We want to specify this new problem as the composition of a homomorphism with the function **subs**. Therefore, we determine the desired result of our function, called h, when applied to a singleton set and when applied to the union of two sets. For the singleton case we have

$$h\,\{a\} = f\,a = (f_0\,a, \ldots, f_V\,a)\,,$$

where, for $i : 0 \leq i \leq V$:

$$f_i\,a = \begin{array}{ll} a & \text{if vol } a = i \\ 1_\uparrow & \text{otherwise .} \end{array}$$

And we have

$$h\,(x \cup y) = (h\,x)\;\Upsilon_\uparrow\;(h\,y)\,.$$

Hence, finding the most precious subsequence with volume equal to i for all i in between 0 and C is specified by

$$\Upsilon_\uparrow/ \cdot f* \cdot \mathbf{subs}\,.$$

If f is a left–reduction $\oslash \not\rightarrow u$ such that $(\oslash a)$ is $(\Upsilon_\uparrow, \Upsilon_\uparrow)$–promotable for all a, then we can apply the **subs**–promotion theorem and obtain a left–reduction for our specification.

In order to obtain a left–reduction for f we determine $f\,[\,]$ and $f\,(x \not\prec a)$. We have

$$\begin{array}{ll} & f\,[\,] \\ = & \quad \text{definition of } f \\ & (f_0\,[\,], \ldots, f_V\,[\,]) \\ = & \quad \text{definition of } f_i \\ & ([\,], 1_\uparrow, \ldots, 1_\uparrow)\,, \end{array}$$

and

$$\begin{array}{ll} & f\,(x \not\prec a) \\ = & \quad \text{definition of } f \\ & (f_0\,(x \not\prec a), \ldots, f_V\,(x \not\prec a))\,. \end{array}$$

We want to express $f_i\,(x \not\prec a)$ in terms of $f\,x$ and a, for all i. Our goal is to write $f\,(x \not\prec a)$ as $(f\,x) \oslash a$ for some operator $\oslash$. We have

$$\begin{array}{ll} & (f_0\,(x \not\prec a), \ldots, f_V\,(x \not\prec a)) \\ = & \quad \text{definition of } f_i \\ & (\not\prec a)*\,(\pi_{0-\pi_2\,a}, \ldots, \pi_{V-\pi_2\,a})\;f\;x\,. \end{array}$$

where π_i with i negative is the constant function $1_\uparrow$, and $\not\prec$ is defined such that $1_\uparrow \not\prec a = 1_\uparrow$ for all a. The definition of the map operator on tuples is similar to the definition of the map operator on sets. Thus, we have found an operator $\oslash$ and a value u such that $f = \oslash \not\rightarrow u$. The operator $\oslash$ is defined by

$$x \oslash a = (\not\prec a)*\,(\pi_{0-\pi_2\,a}, \ldots, \pi_{V-\pi_2\,a})\;x\,,$$

and u is given by $u = ([\,], 1_\uparrow, \ldots, 1_\uparrow)$.

We apply the subs-promotion theorem. The operator $\oslash$ has to be $(\Upsilon_\uparrow, \Upsilon_\uparrow)$-promotable. We verify the two conditions. For every n, the tuple $(1_\uparrow, \ldots, 1_\uparrow)$ of length n is the unit of $\Upsilon_\uparrow$ in the domain of tuples of length n. We define an overall unit uz as a fictitious element. Note that the old units on tuples of specific lengths are not units any more. Since now the domain of tuples has been extended by uz, $(\oslash a)$ can be defined such that $\mathsf{uz} \oslash a = \mathsf{uz}$, thereby acquiring the desired equality. For the second condition we have to show that

$$(x \,\Upsilon_\uparrow\, y) \oslash a = (x \oslash a) \,\Upsilon_\uparrow\, (y \oslash a) .$$

We have, assuming that x_i and y_i with i negative equal $1_\uparrow$,

$$\begin{array}{ll}
 & (x \,\Upsilon_\uparrow\, y) \oslash a \\
= & \quad \text{definition of } \oslash \\
 & (\not\!\prec a) * (\pi_{0-\pi_2\, a}, \ldots, \pi_{V-\pi_2\, a})\ (x \,\Upsilon_\uparrow\, y) \\
= & \quad \text{definition of shared composition, } \pi_i \text{, and zip} \\
 & (\not\!\prec a) * (x_{0-\pi_2\, a} \uparrow y_{0-\pi_2\, a}, \ldots, x_{V-\pi_2\, a} \uparrow y_{V-\pi_2\, a}) \\
= & \quad (\not\!\prec a) \text{ is } (\uparrow, \uparrow)\text{-promotable for some refinement of } \uparrow \\
 & ((x_{0-\pi_2\, a} \not\!\prec a) \uparrow (y_{0-\pi_2\, a} \not\!\prec a), \ldots, (x_{V-\pi_2\, a} \not\!\prec a) \uparrow (y_{V-\pi_2\, a} \not\!\prec a)) \\
= & \quad \text{definition of zip} \\
 & (x_{0-\pi_2\, a} \not\!\prec a, \ldots, x_{V-\pi_2\, a} \not\!\prec a) \,\Upsilon_\uparrow\, (y_{0-\pi_2\, a} \not\!\prec a, \ldots, y_{V-\pi_2\, a} \not\!\prec a) \\
= & \quad \text{definition of } \oslash \text{ and } x \text{ and } y \\
 & (x \oslash a) \,\Upsilon_\uparrow\, (y \oslash a) .
\end{array}$$

It follows that $(\oslash a)$ is $(\Upsilon_\uparrow, \Upsilon_\uparrow)$-promotable.

Since all the requirements of the subs-promotion theorem are satisfied, we have that

$$h \cdot \mathsf{subs} = \odot \not\rightarrow ([\,], 1_\uparrow, \ldots, 1_\uparrow) ,$$

where

$$x \odot a = x \,\Upsilon_\uparrow\, (x \oslash a) .$$

A derivation in the Bird–Meertens formalism of an algorithm for the same problem without the restriction that volumes are natural numbers using backtracking and branch–and–bound is presented in [9]. Variants of the algorithm presented here are described in [17].

4 Permutations

In this section we show when the composition of a homomorphism with the function perms computing all permutations of a list is a left–reduction. In the perms-promotion theorem the conditions under which this is possible are listed. Using the perms-promotion theorem we derive an algorithm for sorting a list.

The function perms can be defined elegantly on bags using the inverse image operator. Define the function bagify which turns a snoc–list into a bag by

$$\mathsf{bagify} = \uplus \not\rightarrow 1_\uplus .$$

Now we define perms as Inv(bagify). A recursive characterization of perms, defined on snoc–lists, is given by

$$\begin{array}{lcl} \mathsf{perms} & : & \alpha\wr\dagger \leftarrow \alpha\wr \\ \mathsf{perms}\ [\,] & = & \{[\,]\} \\ \mathsf{perms}\ (x \prec\!\!\!\times a) & = & \cup/\ (\nabla a)*\ \mathsf{perms}\ x\ . \end{array}$$

where

$$\begin{array}{lcl} \nabla & : & \alpha\wr\dagger \leftarrow \alpha\wr \| \alpha \\ [\,]\nabla a & = & \{[\,] \prec\!\!\!\times a\} \\ (x \prec\!\!\!\times b)\nabla a & = & \{(x \prec\!\!\!\times b) \prec\!\!\!\times a\} \cup ((\prec\!\!\!\times b)*\ (x \nabla a))\ . \end{array}$$

It follows that perms is a left–reduction $\oplus \not\leftarrow e$, where $e = \{[\,]\}$ and $\oplus$ is defined by $x \oplus a = \cup/\ (\nabla a)*\ x$.

Again, the generic specification of the problems we consider is

$$\otimes/ \cdot g* \cdot \mathsf{perms}\ ,$$

where $\otimes$ and g are arbitrary.

Since perms is a left–reduction, we can apply the snoc–lists promotion theorem with $h = \otimes/ \cdot g*$. For that purpose we have to compute $h\ e$ and we have to find an operator $\odot$ such that $h\ (x \oplus a) = (h\ x) \odot a$, where e and $\oplus$ are the components of the left–reduction given for perms above. As in the previous section, we have that $h\ e = g\ [\,]$. For the definition of $\odot$ we calculate as follows.

$$\begin{array}{ll} & h\ (x \oplus a) \\ = & \qquad \text{definition of } h \text{ and } \oplus \\ & \otimes/\ g*\ \cup/\ (\nabla a)*\ x \\ = & \qquad \text{promotion theorem} \\ & \otimes/\ (\otimes/ \cdot g*)*\ (\nabla a)*\ x \\ = & \qquad \text{map distributivity} \\ & (\otimes/ \cdot (\otimes/ \cdot g* \cdot (\nabla a))*)\ x\ . \end{array}$$

Since we want to find an expression in a and $\otimes/\ g*\ x$, the first thing we require is

$$\otimes/ \cdot g* \cdot (\nabla a) = (\oslash a) \cdot g\ ,$$

for some operator $\oslash$. Proceeding with the derivation we get

$$\begin{array}{ll} & \otimes/ \cdot (\otimes/ \cdot g* \cdot (\nabla a))* \\ = & \qquad \text{first requirement} \\ & \otimes/ \cdot ((\oslash a) \cdot g)* \\ = & \qquad \text{map distributivity} \\ & \otimes/ \cdot (\oslash a)* \cdot g* \\ = & \qquad \text{promotion theorem, second requirement} \\ & (\oslash a) \cdot \otimes/ \cdot g*\ . \end{array}$$

The application of the promotion theorem requires $(\oslash a)$ to be $(\otimes, \otimes)$–promotable. Hence we may define $\odot$ to be equal to $\oslash$. We have proved

Theorem 6 (**perms**–**promotion**) *Let* h *be a homomorphism* $\otimes/ \cdot g*$ *defined on sets, such that* $\otimes/ \cdot g* \cdot (\nabla a) = (\oslash a) \cdot g$ *for some section* $(\oslash a)$ *which is* $(\otimes, \otimes)$*–promotable. Then*

$$h \cdot \mathsf{perms} = \oslash \mathrel{\not\rightarrow} u \,,$$

where $u = g\,[\,]$.

Again, as noted after the **subs**–promotion theorem, the section $(\oslash a)$ need only promote over $(\otimes, \otimes)$ on the range of $g*$. Note the difference with the **subs**–promotion theorem: instead of the requirement that g is a left–reduction whose operator satisfies some equality, we have the rather complicated condition that g satisfies $\otimes/ \cdot g* \cdot (\nabla a) = (\oslash a) \cdot g$.

Before we give an example, we have to develop some more theory. We will give an algorithm for sorting a list. The specification with which we will start is

$$\|/ \cdot \mathit{sorted} \lhd \cdot \mathsf{perms} \,,$$

where the predicate *sorted* determines whether a list is sorted, and the binary operator $\|$ is a function which selects one of its arguments. This specification is an instantiation of the general scheme

$$\otimes/ \cdot p \lhd \cdot \mathsf{perms} \,.$$

In the previous section we have shown that $\otimes/ \cdot p \lhd$ equals $\otimes/ \cdot (\otimes/ \cdot \hat{p})*$. In order to prove that the first condition of the **perms**–promotion theorem holds, i.e., $\otimes/ \cdot (\otimes/ \cdot \hat{p})* \cdot \nabla a = (\oslash a) \cdot \otimes/ \cdot \hat{p}$, for some operator $\oslash$, we impose the following conditions upon p.

A predicate q is called *subsequence–closed* if and only if it satisfies for all lists x and y, and all values a,

$$q\,((x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} a) \mathrel{-\!\!\!\prec\!\!\!\!\!\times} y) \Rightarrow q\,(x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} y) \,.$$

Suppose p is a subsequence–closed predicate which holds for the empty list. Now we define $(x \oslash a)$ as $\otimes/ \cdot p \lhd \cdot (x \nabla a)$ for $x \neq 1_\otimes$, and $1_\otimes \oslash a = 1_\otimes$ ($1_\otimes$ is a left–zero of $\oslash$). We will argue that

$$\otimes/ \cdot p \lhd \cdot \nabla a = (\oslash a) \cdot \otimes/ \cdot \hat{p} \,. \tag{1}$$

On arguments for which p holds, $\otimes/ \cdot \hat{p}$ acts as the identity. Hence this equation is trivially true for the empty list. In the case of a nonempty list $x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b$ such that $p\,x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b$ holds, $\otimes/\,\hat{p}\,x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b = x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b$, and again (1) holds. Finally, suppose $\neg p\,x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b$ holds. The right–hand side of (1) is equal to $1_\otimes \oslash a$. For the left–hand side, we use the following implication. If the predicate p is subsequence–closed, then

$$\neg p\,x \Rightarrow \forall y \in (x \nabla a) : \neg p\,y \,.$$

This implication is an immediate consequence of the definition of ∇ and subsequence–closed predicates; its proof is omitted. We have

$$\begin{array}{ll} & \otimes/\ p \lhd\,((x \mathrel{-\!\!\!\prec\!\!\!\!\!\times} b) \nabla a) \\ = & \quad \text{above implication} \\ & \otimes/\ 1_\cup \\ = & \quad \text{definition of reduction} \\ & 1_\otimes \,. \end{array}$$

Hence (1) holds in all cases. We have proved

Lemma 7 *If p is a subsequence–closed predicate which holds for the empty list, and if the operator $\oslash$ is defined by $1_\otimes \oslash a = 1_\otimes$ and*

$$x \oslash a = \otimes/\, p\triangleleft\, (x \nabla a)\,,$$

for $x \neq 1_\otimes$, then

$$\otimes/ \cdot p\triangleleft \cdot (\nabla a) = (\oslash a) \cdot \otimes/ \cdot \hat{p}\,.$$

Example 8 We want to derive an algorithm for sorting a list. The specification of our problem reads

$$|/ \cdot \mathit{sorted}\triangleleft \cdot \mathsf{perms}\,,$$

where the predicate *sorted* is defined by

$$\begin{array}{lcl} \mathit{sorted}\ [\,] & = & \mathsf{True} \\ \mathit{sorted}\ [\,] \prec\!\!\!\cdot\, a & = & \mathsf{True} \\ \mathit{sorted}\ x \prec\!\!\!\cdot\, b \prec\!\!\!\cdot\, a & = & (\mathit{sorted}\ x \prec\!\!\!\cdot\, b) \wedge (a \geq b)\,. \end{array}$$

The operator $|$ is a function which selects one of its arguments. This selector function is associative, idempotent and commutative, but we are not interested in its precise definition. By definition of *sorted* and perms, the set $\mathit{sorted}\triangleleft \cdot \mathsf{perms}$ consists of one or more equal elements. So for example, instead of $|/$ we could just as well have written $\gg /$, or $\ll /$, where $\gg$ is defined by $a \gg b = b$, and $\ll$ is defined by $a \ll b = a$. Note that the reduction $\gg /$ equals the function which is usually written as last, and $\ll /$ equals hd.

Since *sorted* is subsequence–closed and holds for the empty list, we have by Lemma 7

$$|/ \cdot \mathit{sorted}\triangleleft \cdot (\nabla a) = (\oslash a) \cdot |/ \cdot \hat{\mathit{sorted}}\,,$$

where the operator $\oslash$ is defined by $1_| \oslash a = 1_|$ and

$$x \oslash a = |/\ \mathit{sorted}\triangleleft\, (x \nabla a)\,.$$

Hence the first condition of the perms–promotion theorem is satisfied. For the second condition we have to verify that $1_| \oslash a = 1_|$, which holds by definition of $\oslash$, and that

$$(x \oslash a)|(y \oslash a) = (x|y) \oslash a\,. \tag{2}$$

Here we make use of the observations made after the snoc–lists promotion theorem and the perms–promotion theorem. We may suppose that x and y are both elements of $(|/ \cdot \hat{\mathit{sorted}})* z$ for some z, and that $z = \mathsf{perms}\ v$ for some v. It follows that $x = y$ or one or both of them are equal to $1_|$. Equality (2) follows immediately.

We apply the perms–promotion theorem to obtain

$$|/ \cdot \mathit{sorted}\triangleleft \cdot \mathsf{perms} = \oslash \prec\!\!\!\cdot\, [\,]\,.$$

We have derived a cubic time version of insertion-sort. The well known insertion-sort algorithm is obtained if $|/ \cdot \mathit{sorted}\triangleleft \cdot (\nabla a)$ is developed similar to $\otimes/ \cdot g* \cdot \mathsf{perms}$.

Other sorting algorithms can be derived by varying the way of enumerating permutations in the function perms. For example, if we define perms as a homomorphism on join–lists, i.e., if it is defined on the empty list, singletons and $x +\!\!+\, y$ for two lists x and y, we can derive merge–sort with a development similar to the above one.

Sorting algorithms are often used to exemplify programming methodologies. Transformational developments, using the fold-unfold technique, of a number of sorting algorithms are given in [5]. The transformation rules applied there are much more low-level than the ones we apply, and therefore the derivations tend to get much longer. Furthermore, no theory (such as the perms–promotion theorem) is developed. Using a standard technique for deriving divide–and–conquer algorithms, Smith reports a derivation of a sorting algorithm in [18]. Again, the abstract formulation of the applicability conditions we give in the perms–promotion theorem is not present there. Finally, using the deductive synthesis framework developed by Manna and Waldinger several sorting algorithms are derived in [19].

5 Partitions

In this section we show when the composition of a homomorphism with the function parts computing all partitions of a list is a left–reduction. The derivation of the parts–promotion theorem is very similar to the derivations of the perms– and subs–promotion theorems, and is omitted. In an example we derive an algorithm which solves the problem of finding the smallest square such that a partition of a piece of text fits in the square.

The function parts computes all partitions of a list. As an example, a partition of the list $[\,] \prec 1 \prec 3 \prec 2$ is $[\,] \prec ([\,] \prec 1 \prec 3) \prec ([\,] \prec 2)$. The empty list is not allowed in a partition. The inverse image definition of parts has been given in the introduction. The function parts is characterized as a left–reduction as follows.

$$\begin{array}{lcl} \mathsf{parts} & : & \alpha \wr\wr \dagger \leftarrow \alpha \wr \\ \mathsf{parts}\,[\,] & = & \{[\,]\} \\ \mathsf{parts}\,(x \prec a) & = & \cup/\,(\Psi a)*\,\mathsf{parts}\,x\,. \end{array}$$

where the function Ψ is defined by

$$\begin{array}{lcl} \Psi & : & \alpha\wr\wr\dagger \leftarrow \alpha\wr\wr \| \alpha \\ [\,]\Psi a & = & \{[\,] \prec ([\,] \prec a)\} \\ (zs \prec z)\Psi a & = & \{zs \prec (z \prec a), (zs \prec z) \prec ([\,] \prec a)\}\,. \end{array}$$

The number of partitions of a list is exponential in the length of the list, and therefore the algorithm

$$\otimes/ \cdot g* \cdot \mathsf{parts}\,,$$

where $\otimes$ and g are arbitrary, requires exponential time for its evaluation. One way to obtain a more efficient algorithm is to apply the following theorem.

Theorem 9 (parts–promotion) *Let h be a homomorphism $\otimes/ \cdot g*$ defined on lists, such that $\otimes/ \cdot g* \cdot (\Psi a) = (\oslash a) \cdot g$ for some section $(\oslash a)$ which is $(\otimes, \otimes)$–promotable. Then*

$$h \cdot \mathsf{parts} = \oslash \not\rightarrow u\,,$$

where $u = g\,[\,]$.

Here again, it suffices to show promotability of $(\oslash a)$ over $(\otimes, \otimes)$ on the range of $g*$.

Expressions of the form $\downarrow_\# / \cdot (\mathsf{all}\ p)\triangleleft \cdot \mathsf{parts}$ arise in a number of applications. Some theorems in which conditions are given under which this expression semantically equals a left–reduction can be found in [7]. By instantiating the requirements of the above theorem with h being the homomorphism $\downarrow_\#/ \cdot (\mathsf{all}\ p)\triangleleft$, we obtain the following corollary. A predicate p is called *segment–closed* if and only if it satisfies for all lists x and y

$$p\,(x +\!\!\!\!\!\!\times\ y) \Rightarrow (p\,x) \wedge (p\,y)\,.$$

We refine the definition of $\downarrow_\#$ in the following way. Suppose x and y are lists of lists. Then $x \downarrow_\# y = x$ if $x <_\# y$ or if $x =_\# y$ and $(\mathsf{last}\ x) <_\# (\mathsf{last}\ y)$. The proof of the following corollary is omitted, but can be obtained by applying the parts–promotion theorem.

Corollary 10 *Suppose the predicate p is segment–closed and holds for all singletons. Then*

$$\downarrow_\#/ \cdot (\mathsf{all}\ p)\triangleleft \cdot \mathsf{parts} = \odot \mathbin{/\!\!\!\!\!\!\times} i\,,$$

where $i = [\,]$ and $\odot$ is defined by

$$\begin{array}{lcll} [\,] \odot a & = & [\,] +\!\!\!\!\!\!\times ([\,] +\!\!\!\!\!\!\times a) & \\ (zs +\!\!\!\!\!\!\times z) \odot a & = & zs +\!\!\!\!\!\!\times (z +\!\!\!\!\!\!\times a) & \textit{if } p\ z +\!\!\!\!\!\!\times a \\ & & (zs +\!\!\!\!\!\!\times z) +\!\!\!\!\!\!\times ([\,] +\!\!\!\!\!\!\times a) & \textit{otherwise}\,. \end{array}$$

Since the predicate *ascending* is segment–closed and holds for all singletons, the example specification mentioned in the introduction, namely

$$\downarrow_\#/ \cdot (\mathsf{all}\ \textit{ascending})\triangleleft \cdot \mathsf{parts}\,,$$

can be transformed into an efficient algorithm using this corollary. We now derive an algorithm for a partition problem in the following example. Some other algorithms are given in [2] and [7].

Example 11 Suppose we have a piece of text, and we want to find the square with the least area in which the text fits, split into lines but with all words undivided. We suppose that words are encoded as numbers which denote the length of the words. This problem can be specified as follows.

$$\downarrow_{\mathsf{size}}/ \cdot \mathsf{parts}\,,$$

where

$$\begin{array}{lcl} \mathsf{size}\ x & = & (\mathsf{height}\ x) \uparrow (\mathsf{width}\ x) \\ \mathsf{height} & = & \# \\ \mathsf{width} & = & (\uparrow \cdot (\mathsf{id} \| \mathsf{sum})) \mathbin{/\!\!\!\!\!\!\times} 0\,, \end{array}$$

where the function sum is a function computing the sum of a snoc–list. It is defined as the left–reduction $+ \mathbin{/\!\!\!\!\!\!\times} 0$. The function $\downarrow_{\mathsf{size}}$ is underspecified; when $x =_{\mathsf{size}} y$ the result of $x \downarrow_{\mathsf{size}} y$ is not yet specified. We suppose that in the case that $x =_{\mathsf{size}} y$, then $x \downarrow_{\mathsf{size}} y = x \downarrow_\# y$. This adjusted function $\downarrow_{\mathsf{size}}$ is still underspecified, but this degree of specification suffices for our purposes. We briefly describe the derivation of a quadratic-time algorithm, the details are left to the reader.

In order to apply the parts–promotion theorem, we verify the requirements given in the theorem. First, we have to show that $\downarrow_{\mathsf{size}}/ \cdot (\Psi a) = (\oslash a)$ for some operator $\oslash$. If we take this to be the definition of $\oslash$ the first requirement is satisfied. Second, the function $(\oslash a)$ should promote over $\downarrow_{\mathsf{size}}$, i.e., we have to prove that $1_{\downarrow_{\mathsf{size}}} \oslash a = 1_{\downarrow_{\mathsf{size}}}$, and

$$(x \oslash a) \downarrow_{\mathsf{size}} (y \oslash a) = (x \downarrow_{\mathsf{size}} y) \oslash a \,,$$

for all a, and for x and y partitions of some list z.

In general, these equations do not hold. Let $y = [\,] \prec ([\,] \prec 3 \prec 2)$ and $x = [\,] \prec ([\,] \prec 3) \prec ([\,] \prec 2)$ be partitions of $z = [\,] \prec 3 \prec 2$. Obviously, $x <_{\mathsf{size}} y$. However, it can be calculated that $(x \oslash 7) \downarrow_{\mathsf{size}} (y \oslash 7) = y \oslash 7$. Since $y \oslash 7 \neq x \oslash 7$, the equation is false.

From the above we conclude that it is not possible to find an efficient algorithm for the problem we consider using the parts–promotion theorem. Therefore, we have to find a generalization for our problem. Many generalizations can be tried. Partitions of specific length might be considered, as de Moor and Bird do in [8] for their solution of the so–called Mark Thatcher problem. In our case this does not help. We apply the following generalization step.

The maximum of a snoc–list of natural numbers is computed by the function m, which is defined by $m = \uparrow \not\rightarrow 0$. Note that

$$\mathsf{size}\ \downarrow_{\mathsf{size}}/\ \mathsf{parts}\ x \leq (\#\, x) \uparrow (m\, x) \,,$$

since the partition of x into singletons has size $(\#\, x) \uparrow (m\, x)$. Note furthermore that if $m\, x > \#\, x$, then the partition of x into singletons has size $m\, x$, and this partition is one of the minimally sized ones, and so the problem is solved. Let us consider the problem with $m\, x \leq \#\, x$.

For the generalization step, define for i in between $m\, x$ and $\#\, x$ inclusive

$$s_i = \downarrow_{\mathsf{height}}/ \cdot ((\leq i) \cdot \mathsf{width}))\triangleleft \cdot \mathsf{parts} \,.$$

By definition of i, the partition of x into singletons satisfies $((\leq i) \cdot \mathsf{width})$, and hence $s_i\, x \neq 1_{\downarrow_{\mathsf{height}}}$ for all i such that $m\, x \leq i \leq \#\, x$. Abbreviate $\downarrow_{\mathsf{size}}/\ \mathsf{parts}\ x$ to z. It is not difficult to verify the following equality

$$z =_{\mathsf{size}} s_{\mathsf{size}\, z}\ x \,.$$

Furthermore, we have by definition of z for all i such that $m\, x \leq i \leq \#\, x$,

$$s_i\ x \geq_{\mathsf{size}} s_{\mathsf{size}\, z}\ x \,.$$

Hence we have for our problem

$$\begin{array}{ll} & \downarrow_{\mathsf{size}}/\ \mathsf{parts}\ x \\ =_{\mathsf{size}} & \qquad \text{definition of } \mathsf{size} \\ & \downarrow_{\mathsf{size}}/\ \{s_{m\, x}\ x, \ldots, s_{\#\, x}\ x\} \,. \end{array}$$

It follows that if we can compute s_i efficiently for all i in between $m\,x$ and $\#\,x$, we can compute $\downarrow_{\mathsf{size}}/$ parts x efficiently (here we make use of the underspecification of $\downarrow_{\mathsf{size}}$: we want to find the size of the smallest square in which the given text fits, and we are not interested in how to fit the text in the square. If we are interested in how to fit the text in the square, we can define a valuation function v and refine the definition of $\downarrow_{\mathsf{size}}$ as follows

$$x \downarrow_{\mathsf{size}} y = \begin{array}{ll} x & \text{if } x <_{\mathsf{size}} y \\ y & \text{if } y <_{\mathsf{size}} x \\ x \downarrow_v y & \text{otherwise .} \end{array}$$

The generalization step we make is not guaranteed to work for the refined problem $\downarrow_{\mathsf{size}}/\mathsf{parts}$). For the computation of s_i we can use Corollary 10. Note that the predicate $(\leq i)\cdot\mathsf{width}$ is equivalent to the predicate $\mathsf{all}\,((\leq i)\cdot\mathsf{sum})$. Since $i \geq m\,x$, $(\leq i)\cdot\mathsf{sum}$ holds for singletons. Because the elements of x are natural numbers, the predicate $(\leq i)\cdot\mathsf{sum}$ is segment–closed. Hence the two conditions of the corollary are satisfied. Corollary 10 gives us a left–reduction for each s_i, $m\,x \leq i \leq \#\,x$. If n is the length of the list, we compute $O(n)$ left–reductions (one for each i in between the maximum and the length of the list) which all can be evaluated in time $O(n)$. Hence the algorithm for $\downarrow_{\mathsf{size}}/$ parts x we have obtained requires time $O(n^2)$ for its evaluation. This algorithm is not asymptotically optimal; there exists an $O(n \log n)$ algorithm.

6 Conclusions

We have presented three theorems and three examples which illustrate the derivation of algorithms in the Bird–Meertens formalism. The theorems we proved are applicable to a large class of problems. Lots of algorithms for operations research problems are immediate consequences of the theorems. More generally, at the International Summer School on Constructive Algorithmics, de Moor has shown that it is possible to derive algorithms for lots of dynamic programming problems using the formalism described here. Instead of enumerator functions, he starts his derivations with inverses of homomorphisms. The results are slightly more general than our results, but also more complex. We feel that the elegance of our approach lies in the total absence of inductive arguments. Another advantage of our approach is its consistency. Given a specification in the form of a homomorphism composed with an enumerator function, we first derive the 'promotion' theorem for the enumerator function. If the homomorphism does not satisfy the conditions of the promotion theorem, generalization steps have to be sought for. Often, these generalization steps can be derived in a standard fashion, see [8].

Acknowledgements. The algorithms presented here were derived when I was in Oxford visiting Oege de Moor and Richard Bird. I want to thank them both for providing a pleasant stay and an inspiring environment. Lambert Meertens invented the partition problem solved in the example of Section 6. The comments and explanations of Maarten Fokkinga and Lambert Meertens are gratefully acknowledged.

References

[1] R.C. Backhouse. An exploration of the Bird–Meertens formalism. Technical Report Computing Science Notes CS 8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[2] R.S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6:159–189, 1986.

[3] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[4] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[5] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[6] O. de Moor. Inverses in program synthesis. Lecture Notes International Summer School on Constructive Algorithmics, Hollum-Ameland, The Netherlands, 1989.

[7] O. de Moor. List partitions. Lecture Notes International Summer School on Constructive Algorithmics, Hollum-Ameland, The Netherlands, 1989.

[8] O. de Moor and R.S. Bird. Lecture notes on nub theory. Lecture Notes International Summer School on Constructive Algorithmics, Hollum-Ameland, The Netherlands, 1989.

[9] M.M. Fokkinga. An exercise in transformational programming—backtracking and branch-and-bound. Technical Report Memorandum INF-88-22, Department of Computing Science, University Twente, 1988.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP–Completeness.* W.H. Freeman and company, 1979.

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

[12] J. Jeuring. The derivation of an algorithm for finding palindromes. Lecture Notes International Summer School on Constructive Algorithmics, Hollum–Ameland, Part 2, 1989.

[13] J. Jeuring. Deriving algorithms on binary labelled trees. In P.M.G. Apers, D. Bosman, and J. van Leeuwen, editors, *Proceedings SION Computing Science in the Netherlands*, pages 229–249, 1989.

[14] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989. LNCS 375.