Solving a combinatorial problem by transformation of abstract data types

Eerke A. Boiten

Department of Informatics, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen
email: `eerke@cs.kun.nl`

Abstract

Techniques from the area of formal specification are shown to be useful in the analysis of combinatorial problems. A problem description is given, using an abstract data type. By gradual elimination of the equivalences on the data types a unique representation of the type is derived which reduces the new problem to a known one.

Keywords

Abstract data types, bracketing problems, permutations, problem reduction.

This research has been sponsored by the Netherlands Organization for Scientific Research (NWO), under grant NF 63/62-518 (the STOP — Specification and Transformation of Programs — project).

1 Introduction

In 1965, C.H.A. Koster [Kos65] described an operator for creating permutations of strings. It is denoted by $\lceil\,|\,\rfloor$[1], and has the well-known interpretation from proof reading: $\lceil a | b \rfloor$ denotes the string *ba*, and $\lceil\lceil \mathrm{e}|\mathrm{pl}\rfloor|\lceil \mathrm{am}|\mathrm{ex}\rfloor\rfloor$ denotes the string *example*. Koster used the $\lceil\,|\,\rfloor$ operator for the description of transducers using affix grammars. Recently [Kos90], he posed the question whether the set of permutations that can be generated using only $\lceil\,|\,\rfloor$ in a nested fashion (Koster calls these *inversions*; we follow van Leijenhorst [vL90] in calling them *K-permutations*) differs significantly from the set of (ordinary) permutations of a string. For example, $[2,4,1,3]$ and $[3,1,4,2]$ are (the only) two permutations of $[1,2,3,4]$ that are not K-permutations of $[1,2,3,4]$.

An answer to this question appears in [vL90]: it is shown there how the number of K-permutations of a string consisting of n distinct elements is bounded by 9.9179^n and therefore much less than $n!$, using formal power series and estimates of integrals. A more recent analysis [vL91] of these results leads to an upper bound of $\approx 7.1^n$. We will present a very simple proof that 8^n is an upper bound, and a reduction to normal form of K-permutations by way of abstract data type transformations. From this, an exact formula counting the number of K-permutations is derived.

2 Preliminaries

The usual BMF (Bird-Meertens Formalism) notation for sets and lists is used [Bir87, Mee89a]. The reader is referred to [Bir87] for formal definitions of the BMF operators used in this paper. Informally, they can be characterized by:

$$\begin{aligned}
[a_1,\ldots,a_n]+\!\!+[b_1,\ldots,b_m] &= [a_1,\ldots,a_n,b_1,\ldots,b_m] \\
[\,]+\!\!+x = x+\!\!+[\,] &= x \\
f*[a_1,\ldots,a_n] &= [f\ a_1,\ldots,f\ a_n] \\
\oplus/[a_1,\ldots,a_n] &= a_1\oplus\ldots\oplus a_n \\
(a\oplus)x &= a\oplus x \\
(\oplus a)x &= x\oplus a \\
a\tilde{\oplus}b &= b\oplus a \\
[a_1,\ldots,a_n]\,\mathsf{X}_{\oplus}\,[b_1,\ldots,b_m] &= [a_1\oplus b_1,\ldots,a_n\oplus b_1,\ldots,a_1\oplus b_m,\ldots,a_n\oplus b_m] \\
(f\hat{\oplus}g)x &= (f\ x)\oplus(g\ x).
\end{aligned}$$

Using this notation, the set of permutations of a list is given by:

$$\begin{aligned}
\mathit{perms}\,[\,] &= \{[\,]\} \\
\mathit{perms}\,[a] &= \{[a]\} \\
\mathit{perms}\,(l+\!\!+m) &= \cup/((\mathit{perms}\ l)\,\mathsf{X}_{\odot}\,(\mathit{perms}\ m)) \\
&\quad \text{where} \\
[\,]\odot m &= \{m\} \\
m\odot[\,] &= \{m\}
\end{aligned}$$

[1]Or alternatively $\lfloor\,|\,\rceil$

$$
\begin{array}{rcl}
([a]{+\!\!+}l) \odot ([b]{+\!\!+}m) & = & ([a]{+\!\!+}) * (l \odot ([b]{+\!\!+}m)) \cup \\
 & & ([b]{+\!\!+}) * (([a]{+\!\!+}l) \odot m)
\end{array}
$$

(the operator $\odot$ takes two lists and merges them in all possible ways).

We also use the inverse operator:

$$f^{-1}\ x = \{y \mid f\ y = x\}.$$

3 K-permutation patterns

First we present an informal specification, more or less as given by Koster:

A K-permutation of a string can be constructed as follows:

- *choose two arbitrary adjacent nonempty substrings;*
- *interchange these.*
- *Repeat this as necessary within the chosen substrings, or in the unchanged part of the string.*

This can be straightforwardly translated into a formal specification of the type *Kperm*, which is an extension of the type *List* of *nonempty* lists. The element type is denoted by α. We will write $a \sqcap\!\sqcup b$ for $\lceil a | b \rfloor$, occasionally using brackets for disambiguation.

$$\frac{x : \alpha}{[x] : \mathit{Kperm}(\alpha)} \qquad \frac{a, b : \mathit{Kperm}(\alpha)}{a{+\!\!+}b : \mathit{Kperm}(\alpha)} \qquad \frac{a, b : \mathit{Kperm}(\alpha)}{a \sqcap\!\sqcup b : \mathit{Kperm}(\alpha)}$$

Because we consider *Kperm* as a extension of *List*, the $+\!\!+$ operator of *List* is used here as well, and its associativity is also assumed. So one law certainly holds for *Kperm*:

$$(a{+\!\!+}b){+\!\!+}c = a{+\!\!+}(b{+\!\!+}c). \tag{1}$$

This allows to leave out brackets in expressions with multiple occurrences of $+\!\!+$.

The intended interpretation of $a \sqcap\!\sqcup b$ is, of course, $b{+\!\!+}a$. It is possible to add this as an equivalence on the type *Kperm*:

$$a \sqcap\!\sqcup b = b{+\!\!+}a. \tag{2}$$

However, doing so would result in the loss of important structure from the type *Kperm*, viz. what string a *Kperm* term "is a permutation of". This information is retained by including *no* laws that allow changing the order of the basic elements in *Kperm* terms.

Formally, this can be described as follows.

Definition 3.1 Given two[2] functions $F : \alpha \to \beta$ and $G : \alpha \to \gamma$, the (F, G)-induced equivalence $=_{F,G}$ (on α) is defined by

$$x =_{F,G} y \Leftrightarrow ((F\ x) = (F\ y) \wedge (G\ x) = (G\ y)).$$

[2]The generalization to different numbers of functions is obvious.

We add laws L to *Kperm* such that $x =_{I,O} y \Leftrightarrow x =_L y$, where I and O are functions that give for a *Kperm* term the "original" list and the "interpretation", i.e. the permutation that is represented. The "original" list is obtained by replacing all occurrences of $\sqcap\!\!\sqcup$ by $+\!\!+$, and the string that is actually represented by the K-permutation is obtained by replacing all occurrences of $\sqcap\!\!\sqcup$ by $\widetilde{+\!\!+}$. The homomorphisms O (for *O*riginal) and I (for *I*nterpretation) are given by:

$$\begin{array}{rcl} O[a] & = & [a] \\ O(l +\!\!+ m) & = & l +\!\!+ m \\ O(l \sqcap\!\!\sqcup m) & = & l +\!\!+ m \\ I[a] & = & [a] \\ I(l +\!\!+ m) & = & l +\!\!+ m \\ I(l \sqcap\!\!\sqcup m) & = & m +\!\!+ l. \end{array}$$

As mentioned before, associativity of $+\!\!+$ is assumed. Another equivalence that must be added to have (I, O)-induced equivalence on *Kperm* is associativity of $\sqcap\!\!\sqcup$:

$$(a \sqcap\!\!\sqcup b) \sqcap\!\!\sqcup c = a \sqcap\!\!\sqcup (b \sqcap\!\!\sqcup c) \tag{3}$$

(since both have the same elements, read from left to right, and both denote the string $c +\!\!+ b +\!\!+ a$). This is the point where van Leijenhorst's analysis [vL90] is non-optimal: in the grammar he uses, associativity of $+\!\!+$ is (implicitly) used, whereas associativity of $\sqcap\!\!\sqcup$ is not.

Note that the definitions of O and I are sound for *Kperm* with the two associativity laws 1 and 3, since $+\!\!+$ is associative; if equation 2 were added as an equivalence, soundness of O would imply that $l +\!\!+ m = m +\!\!+ l$ for all l and m (since $O(l \sqcap\!\!\sqcup m) = l +\!\!+ m$, $O(m +\!\!+ l) = m +\!\!+ l$). This has the undesired effect of reducing the result type of O from lists to bags.

4 All K-permutations of a list

Now that functions O and I have been defined, the problem posed by Koster can be formally specified. The set of all K-permutations of a list l is

$$\{m \mid \exists n : (O\ n) = l \wedge (I\ n) = m\},$$

or, more concisely

$$I * (O^{-1} l).$$

Then the problem of determining whether there exist a sizable number of permutations that cannot be generated using $\sqcap\!\!\sqcup$ can be specified as follows:
Investigate $f[1..n]$ where

$$f = (\# \cdot \mathit{perms}) \hat{-} (\# \cdot I * \cdot O^{-1}).$$

One way to continue the analysis would be by finding a more efficient program for f. In this case, however, a further analysis on the *data structure* side will prove to be more useful.

5 Determining the number of K-permutation patterns

Using the formal specification given above, we can now give an upper bound for the number of K-permutations by considering K-permutation patterns.

Consider a string l of length n. Between each two successive elements of that string one can imagine a concatenation operator (so, $n-1$ in total). The O operation above consists of two steps: first, all $\sqcap\!\sqcup$ operators are replaced by $+\!\!+$. The second step is more implicit: because $+\!\!+$ is associative, all bracketing in the result is irrelevant and can be eliminated. So, arbitrary elements from $O^{-1}l$ can be constructed by reverting this process: first, an arbitrary complete bracketing of l is chosen, and then some $+\!\!+$ operators are replaced by $\sqcap\!\sqcup$.

Now it is easy to count the number of K-permutation *patterns* (i.e., the ways of placing brackets and $+\!\!+$ or $\sqcap\!\sqcup$ operators, without considering equivalences among those):

- As mentioned by van Leijenhorst [vL90], the number of complete binary bracketings of a string of length n is given by the Catalan number [Cat38]
$$C_n = \frac{1}{n}\binom{2n-2}{n-1}.$$
- Of $(n-1)$ $+\!\!+$-operators, an arbitrary number is replaced by $\sqcap\!\sqcup$; this can be done in 2^{n-1} ways.
- Thus, the number of K-permutation patterns of a string of length n is given by
$$K_n = 2^{n-1}C_n = \frac{2^{n-1}}{n}\binom{2n-2}{n-1}.$$

Since the Catalan numbers are known to be bounded by $C_n < 4^n$, this gives an upper bound of 8^n on the number of K-permutation patterns, and thus also on the number of K-permutations.

6 Lists and rose trees with append and reverse

An upper bound for the number of K-permutations has been given, by considering an abstract data type, disregarding equivalences on that type. In order to investigate more precisely the exact number of K-permutations, the data type involved must be as simple as possible. The description using *Kperm* is highly symmetric: there are two "concatenation" operators, both associative. This can be corrected by a translation from *Kperm* to a data type with one binary operator (concatenation) and one unary operator (reverse). This is the type *Revlist*, with introduction rules:

$$\frac{x : \alpha}{[x] : \mathit{Revlist}(\alpha)} \qquad \frac{a, b : \mathit{Revlist}(\alpha)}{a +\!\!+ b : \mathit{Revlist}(\alpha)} \qquad \frac{a : \mathit{Revlist}(\alpha)}{\overline{a} : \mathit{Revlist}(\alpha)}$$

The transition from *Kperm* to *Revlist* is given by the translation function T, defined by:

$$\begin{array}{rcl} T[a] & = & [a] \\ T(l +\!\!+ m) & = & (T\ l) +\!\!+ (T\ m) \\ T(l \sqcap\!\sqcup m) & = & \overline{\overline{(T\ l)} +\!\!+ (T\ m)}. \end{array}$$

For *Revlist*, the "interpretation" and "original" homomorphisms I' and O' are given by:

$$
\begin{array}{rcl}
I'\,[a] & = & [a] \\
I'(l \mathbin{+\!\!+} m) & = & (I'\,l) \mathbin{+\!\!+} (I'\,m) \\
I'\,\overline{m} & = & \mathit{rev}(I'\,m) \\
O'[a] & = & [a] \\
O'(l \mathbin{+\!\!+} m) & = & (O'\,l) \mathbin{+\!\!+} (O'\,m) \\
O'\,\overline{m} & = & O'\,m
\end{array}
$$

where *rev* is a suitably defined reverse-function on *List*.

The following laws are assumed:

$$
\begin{array}{rcl}
\overline{\overline{m}} & = & m \\
\overline{[a]} & = & [a] \\
(l \mathbin{+\!\!+} m) \mathbin{+\!\!+} n & = & l \mathbin{+\!\!+} (m \mathbin{+\!\!+} n),
\end{array}
$$

where the first two laws can be used as directed equivalences, for normalization of *Revlist* terms. I.e., except for the associativity of append, we can assume a unique normal form for *Revlist*. Note that the equivalence induced by these laws is exactly (I', O')-induced equivalence.

Theorem 6.1
If $x = y$ according to the laws of *Kperm*, then $(T\ x) = (T\ y)$ according to the laws of *Revlist*.

Proof 6.1
Trivially, using associativity of $\mathbin{+\!\!+}$, one can prove that $T((l \mathbin{+\!\!+} m) \mathbin{+\!\!+} n) = T(l \mathbin{+\!\!+} (m \mathbin{+\!\!+} n))$. We tacitly use associativity in the other half of the proof:

$$
\begin{array}{rll}
T(l \sqcap\!\sqcup (m \sqcap\!\sqcup n)) & =_{\{\text{definition } T\}} & \overline{\overline{(T\ l)} \mathbin{+\!\!+} \overline{T(m \sqcap\!\sqcup n)}} \\
 & =_{\{\text{definition } T\}} & \overline{\overline{(T\ l)} \mathbin{+\!\!+} \overline{\overline{\overline{(T\ m)} \mathbin{+\!\!+} \overline{(T\ n)}}}} \\
 & =_{\{\overline{\overline{m}} = m\}} & \overline{\overline{(T\ l)} \mathbin{+\!\!+} \overline{(T\ m)} \mathbin{+\!\!+} \overline{(T\ n)}} \\
 & =_{\{\overline{\overline{m}} = m\}} & \overline{\overline{\overline{\overline{(T\ l)} \mathbin{+\!\!+} \overline{(T\ m)}}} \mathbin{+\!\!+} \overline{(T\ n)}} \\
 & =_{\{\text{definition } T\}} & \overline{\overline{T(l \sqcap\!\sqcup m)} \mathbin{+\!\!+} \overline{(T\ n)}} \\
 & =_{\{\text{definition } T\}} & T((l \sqcap\!\sqcup m) \sqcap\!\sqcup n).\Box
\end{array}
$$

Theorem 6.2
The interpretation and original homomorphisms I' and O' on *Revlist* are equivalent to I and O on *Kperm*, respectively:

1. $I = I' \cdot T$
2. $O = O' \cdot T$

Proof 6.2

1. By structural induction on *Kperm*, using the property of *rev*:

$$\mathit{rev}((\mathit{rev}\ x) \mathbin{+\!\!+} (\mathit{rev}\ y)) = y \mathbin{+\!\!+} x.$$

$$\begin{aligned}
I'(T[a]) &=_{\{\text{def. } T\}} && I'[a] \\
&=_{\{\text{def. } I'\}} && [a] \\
&=_{\{\text{def. } I\}} && I[a] \\
I'(T(l \mathbin{+\!\!+} m)) &=_{\{\text{def. } T\}} && I'(T\ l) \mathbin{+\!\!+} I'(T\ m) \\
&=_{\{\text{induction}\}} && (I\ l) \mathbin{+\!\!+} (I\ m) \\
&=_{\{\text{def. } I\}} && I(l \mathbin{+\!\!+} m) \\
I'(T(l \sqcap\!\sqcup m)) &=_{\{\text{def. } T\}} && I'\overline{\overline{(T\ l)} \mathbin{+\!\!+} \overline{(T\ m)}} \\
&=_{\{\text{def. } I'\}} && \mathit{rev}(I'(\overline{(T\ l)} \mathbin{+\!\!+} \overline{(T\ m)})) \\
&=_{\{\text{def. } I'\}} && \mathit{rev}(\mathit{rev}(I'(T\ l)) \mathbin{+\!\!+} \mathit{rev}(I'(T\ m))) \\
&=_{\{\text{property } \mathit{rev}\}} && (I'(T\ m)) \mathbin{+\!\!+} (I'(T\ l)) \\
&=_{\{\text{induction}\}} && (I\ m) \mathbin{+\!\!+} (I\ l) \\
&=_{\{\text{def. } I\}} && I(l \sqcap\!\sqcup m)\square
\end{aligned}$$

2. Analogously.

Theorem 6.3
T is total, injective and surjective, and thus an isomorphism between *Kperm* and *Revlist* (with laws).

Proof 6.3

- Obviously, T is total, since it is defined inductively over all the constructors of *Kperm*.
- Injectivity of T follows from the fact that the equivalences induced by the introduced laws are the (I, O)-induced and the (I', O')-induced equivalence, respectively.
- Surjectivity of T can be proved by induction on the length (i.e., the number of basic elements) of the *Revlist*, considering normalized terms only.

  Hypothesis $\forall \mathit{Revlist}\ x : \mathit{length}(x) \leq n \Rightarrow (\exists \mathit{Kperm}\ y : (T\ y) = x)$. This will be verbalized as "for every x a T-original exists".

  Base case For $n = 1$, obviously $[a] = T[a]$.

  Induction Any *Revlist* with *length* > 1 is of one of two forms: $l \mathbin{+\!\!+} m$ or $\overline{p \mathbin{+\!\!+} q}$.
  l and m have T-originals l' and m', by induction hypothesis, and thus $l \mathbin{+\!\!+} m$ has a T-original, viz. $l' \mathbin{+\!\!+} m'$.
  We may assume that $p = \overline{p'}$ and $q = \overline{q'}$ because of the law $\overline{\overline{m}} = m$. By induction, p' and q' have T-originals p'' and q'', and thus $\overline{p \mathbin{+\!\!+} q}$ has a T-original, viz. $p'' \sqcap\!\sqcup q''$. $\square$

In order to arrive at a unique representation of K-permutations, we use yet another data type, which may be called a rose tree [Mee89b]. In this type the associativity of $+\!\!\!+$ is factored out. We assume the existence of a type *Plist* of lists with ≥ 2 elements. The types RT and $\overline{RT}$ are defined by:

$$\frac{a : \alpha}{\begin{array}{c}[a] : RT(\alpha) \\ [a] : \overline{RT}(\alpha)\end{array}} \qquad \frac{l : \mathit{Plist}(RT(\alpha))}{\textcircled{\scriptsize $\overline{+\!\!+}/$}\, l : \overline{RT}(\alpha)} \qquad \frac{l : \mathit{Plist}(\overline{RT}(\alpha))}{\textcircled{\scriptsize $+\!\!+/$}\, l : RT(\alpha)}$$

The notations $\textcircled{\scriptsize $+\!\!+/$}$ and $\textcircled{\scriptsize $\overline{+\!\!+}/$}$ have no formal meaning in this context. They do, however, serve the intuition. A node of type $\textcircled{\scriptsize $+\!\!+/$}$ represents all its sons from left to right, a node of type $\textcircled{\scriptsize $\overline{+\!\!+}/$}$ represents all its sons from right to left.

The transition from *Revlist* to RT can best be specified by its inverse T^I:

$$\begin{array}{rcl}
T^I[a] & = & [a] \\
T^I(\textcircled{\scriptsize $+\!\!+/$}\, l) & = & +\!\!\!+/T^I * l \\
T^I(\textcircled{\scriptsize $\overline{+\!\!+}/$}\, l) & = & \overline{+\!\!\!+/(\overline{} \cdot T^I) * L}
\end{array}$$

which is correct since T^I is surjective and injective.

Returning to combinatorics, we can now see that the problem of counting all K-permutations is closely related to Schröder's generalized bracketing problem [Sch70] as presented in [Com74]. There, the problem is to determine the number c_n of different rose trees with ≥ 2 branches at each inner node and n leaves. A recursive formula is given[3] for c_n:

$$\begin{array}{rcl}
(n+1)c_{n+1} & = & 3(2n-1)c_n - (n-2)c_{n-1} \quad \text{for } n \geq 2 \\
, c_1 = c_2 & = & 1.
\end{array}$$

Since the types (viz. $\textcircled{\scriptsize $+\!\!+/$}$ or $\textcircled{\scriptsize $\overline{+\!\!+}/$}$) of the subtrees at all levels of a $RT/\overline{RT}$ tree are completely determined by the type of the root, for each rose tree with a given number of leaves there are exactly two $RT/\overline{RT}$ trees with that number of leaves, viz. one with a $\textcircled{\scriptsize $+\!\!+/$}$ root and one with a $\textcircled{\scriptsize $\overline{+\!\!+}/$}$ root. Thus, we can conclude that the number of K-permutations of a string of length n equals $2c_n$, for $n \geq 2$. This improves the results as given in [vL90, vL91].

7 Concluding remarks

It has been shown how techniques from the area of formal specification may be profitably used in the analysis of combinatorial problems. The formal "game" we played may be relevant for other problems as well. In short, it may be described as follows:

- given an abstract data type A without laws, and two functions I_A and O_A, add laws L_A that construct the equivalence classes w.r.t. I_A and O_A, i.e. for terms x and y, $x =_{L_A} y$ should hold iff $I_A(x) = I_A(y) \wedge O_A(x) = O_A(y)$;

- while the type A has laws, do the following:
define a new data type A', and a function T from the (terms of the) previous data

[3]The formula in [Com74] actually is incorrect: it has $(n-1)c_{n+1}$ as its left hand side. The formula above does conform with the table of c_n values presented in [Com74]. This error has been discovered by Hans Zantema, who refereed this chapter for CSN '91.

type A to A', such that $T(x) = T(y) \Rightarrow x =_{L_A} y$. Define functions $I_{A'}$ and $O_{A'}$, such that $I_A(x) = I_{A'}(T(x))$ and $O_A(x) = O_{A'}(T(x))$. Then add laws for A' that construct equivalence classes w.r.t. $I_{A'}$ and $O_{A'}$.

A suitable choice for such a function T is one such that maps two or more laws from L_A to one and the same law in the new type (like associativity of $+\!\!+$ and of $\sqcap\!\sqcup$ in *Kperm* were both mapped to associativity of $+\!\!+$ in *Revlist*). This may happen if T is not injective on *terms of* A.

- if a data type without laws is obtained, one has a unique normal form for the original data type A (w.r.t. I_A and O_A).

On a more abstract level, this means that for an abstract data type A and functions I and O, we construct the quotient type of A w.r.t. (I, O)-induced equality, by adding laws to that effect, and eventually construct a "free" data type for the quotient.

If this process could be carried out in reverse, this would have interesting applications in the area of implementation of abstract data types, since then abstract data types with laws could be implemented by free types with an explicit equality function.

If one were to consider a data type with $\sqcap\!\sqcup$, $+\!\!+$ and $\overline{}$, one would find that some nice equivalences hold on this data type, resembling the well-known De Morgan-laws in Boolean algebras. D. Turner describes this in [Tur90].

A remaining interesting problem, for which no better than a trivial exponential algorithm has been given, is the *correspondence* problem for K-permutations, i.e. given strings x and y, does a K-permutation z exist, such that $(O\ z) = x \wedge (I\ z) = y$? This may be the subject of further studies.

Acknowledgement

Daniël Tuijnman is thanked for the numerous discussions we had on this subject, his scrutiny of my ideas, and for investigating the relevant literature on combinatorics. Dick van Leijenhorst provided useful comments on the combinatorial aspects of this problem. Kees Koster is thanked for providing this interesting problem; he, Norbert Völker and Helmut Partsch added occasional $\sqcup\!\sqcap$'s and gave useful criticism on drafts of this note. I would like to thank one particular CSN '91 referee for many useful suggestions for improvements.

References

[Bir87] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, pages 5–42. Springer-Verlag, Berlin, 1987.

[Cat38] Catalan. Note sur une équation aux différences finies. *J.M. pures appl.*, 3:508–516, 1838.

[Com74] L. Comtet. *Advanced Combinatorics - The art of finite and infinite expansions.* Reidel, Dordrecht, 1974.

[Kos65] C.H.A. Koster. On the construction of ALGOL-procedures for generating, analysing and translating sentences in natural languages. Technical Report MR 72, Mathematisch Centrum, Amsterdam, February 1965.

[Kos90] C.H.A. Koster, November 1990. Personal communication.

[Mee89a] L.G.L.T. Meertens. Lecture notes on the generic theory of binary structures. In *STOP International Summer School on Constructive Algorithmics, Ameland* [STO89]. Lecture notes.

[Mee89b] L.G.L.T. Meertens. Variations on trees. In *STOP International Summer School on Constructive Algorithmics, Ameland* [STO89]. Lecture notes.

[Sch70] Schröder. Vier combinatorische Probleme. *Z. für M. Phys.*, 15:361–376, 1870.

[STO89] STOP. *STOP International Summer School on Constructive Algorithmics, Ameland*, September 1989. Lecture notes.

[Tur90] D.A. Turner. Duality and De Morgan principles for lists. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business - A Birthday Salute to Edsger W. Dijkstra*, chapter 47, pages 390–398. Springer Verlag, Berlin/Heidelberg/New York, 1990.

[vL90] D.C. van Leijenhorst. On a ternary bracketing problem from the theory of formal languages. Technical Report 90-24, KUN, December 1990.

[vL91] D.C. van Leijenhorst, January 1991. Addendum to [vL90].