Science of Computer Programming 16 (1991) 19-48
Elsevier

An exercise in transformational programming: Backtracking and Branch-and-Bound

Maarten M. Fokkinga
CWI, P.O. Box 4097, 1009 AB Amsterdam, Netherlands

Communicated by J. Darlington
Received May 1988
Revised September 1990

Abstract

Fokkinga, M.M., An exercise in transformational programming: Backtracking and Branch-and-Bound, Science of Computer Programming 16 (1991) 19-48.

We present a formal derivation of program schemes that are usually called Backtracking programs and Branch-and-Bound programs. The derivation consists of a series of transformation steps, specifically *algebraic manipulations*, on the initial specification until the desired programs are obtained. The well-known notions of linear recursion and tail recursion are extended, for structures, to *elementwise* linear recursion and *elementwise* tail recursion; and a transformation between them is derived too.

1. Introduction

Methodologies for the construction of correct programs have attracted wide interest in the past, and in the present. Well known is the assertion method of Floyd for the verification of programs, and the axiomatic basis for computer programming that Hoare [13] founded on this idea. Subsequently, Dijkstra [9, 10] refined the method to a calculus for the construction of so-called totally correct programs. The influence of the work of these three persons is apparent in almost every textbook on programming.

More recently, quite another method for the construction of correct programs has attracted attention: the method of transformational programming; see e.g. Feather [11] and Partsch [18] and the references cited. Basically, one starts with an obviously correct program, or rather specification, for it doesn't need to be effectively computable; and then one applies a series of transformation steps that preserve the correctness but, hopefully, improve the efficiency. In order that the method is practically feasible, it is necessary that the program notation is suitable

for algebraic manipulation; that is, it must be easy to decompose a program into its (semantically meaningful) constituent parts and to recombine them into an operationally slightly different but semantically equivalent form, very much like the "transformations" of a^2-b^2 into $(a+b)(a-b)$ and of $\sin(x+y)$ into $\sin x \cos y + \cos x \sin y$. (Notice that here the "transformations" are just algebraic identities; the same will be true of the kind of transformations that we shall explore in this paper.) A second necessary property of the program notation is its brevity and terseness, for otherwise it would be practically infeasible to rewrite and transform a program in a series of steps until a satisfactory version has been obtained. Imagine, for instance, how one should do elementary high-school algebra with a fully parenthesized prefix notation, dealing with equations like:

$$minus(exp(a, 2), exp(b, 2)) \;=\; mult(plus(a, b), minus(a, b))$$

For the transformational approach to succeed it is really necessary that several programs of high algorithmic content can be placed in a single line and related by the equals sign, say.

A framework for algorithmic programming that meets the above requirements, and many more, has been developed by Meertens in the paper "Algorithmics: towards programming as a mathematical activity" [16]. It is a mathematically rigorous approach to programming that is highly algebraic in nature. Meertens calls it "algorithmics" and we shall refer to his paper as "the Algorithmics paper". We set out to derive in the framework of Algorithmics (the well-known!) programs for Backtracking and Branch-and-Bound (see the explanation below). Apart from the insight in Backtracking and Branch-and-Bound that the reader may get from our high-level, algorithmic discussion and derivation, we also attempt to satisfy Meertens' request for "the discovery and the formulation of 'algebraic' versions of high-level programming paradigms and strategies" [16].

2. Informal discussion of Backtracking and Branch-and-Bound

"Backtracking" is a problem solving method according to which one systematically searches for one or all solutions to a problem by repeatedly trying to extend an approximate solution in all possible ways. Whenever it turns out that such a solution fails, one "backtracks" to the last point of choice where there are still alternatives available. For most problems it is of the utmost importance to spot early on that an approximate solution cannot be extended to a full solution, so that a huge amount of failing trials can be saved. This is called "cutting down the search space". It may diminish the running time of the algorithm by several orders of magnitude.

Now suppose that it is required to find not just any one or all solutions, but an optimal one. In this case one can apply the same method, but every time a solution is encountered the search space can be reduced even further: from then onwards one need not try to extend approximate solutions if it is sure that their extensions

cannot be as good as the currently optimal one. In this case we speak of "Branch-and-Bound".

The above description of Backtracking and Branch-and-Bound is rather operational. It is indeed a description of the sequence of computation steps evoked by the program text, or taken by a human problem solver. It is not at all necessary that the program text itself clearly shows the "backtracking" steps and the "bounding" of the search space. On the contrary, the program text need only show that the required result is delivered; the way in which the result is computed is a property of the particular evaluation method.

Backtracking and Branch-and-Bound are thoroughly discussed in the literature; see e.g. Wirth [21, 22], Alagic and Arbib [1], and many other textbooks on programming. Many of these also provide some sort of correctness argument in the form of assertions or just informal explanation. On close inspection most of them seem incomplete: either the assertions are too weak to carry the proof through, or the implication between assertions (and the invariance of loop assertions) is not proved with mathematical rigour. Even our own previous attempt [12] is not satisfactory in this respect. It also appears that the method of invariant assertions leads to some overspecification: in order to show that the whole search space, i.e., all possibilities, has been investigated some total ordering is imposed on the search space (often some lexicographic order) and it is shown that the search space is traversed along this order. In the transformational approach such an ordering is not needed at all: the very first program, or rather specification, clearly expresses that no possibility is by passed.

We shall illustrate our high-level, algorithmic discussion of Backtracking and Branch-and-Bound by the following simple, but typical, examples: the Problem of an Optimal Selection and a simplification of it, the Problem of a Legal Selection.

The Problem of a Legal Selection (PLS). There is given a collection of N objects, say object $1, 2, \ldots, N$. Each object x has its own weight $w(x)$ and value $v(x)$. The task is to find a selection of the objects, i.e., a subset of $\{1, \ldots, N\}$, whose aggregate weight does not exceed a given limit W. (Slightly more general, the task may be to find all such selections.)

The Problem of an Optimal Selection (POS). With the same assumptions as in PLS, the task is to find a selection of the objects whose weight does not exceed the given limit W and, in addition, whose aggregate value is maximal.

Wirth [22] also discusses POS and we shall arrive at essentially the same algorithm.

3. Preliminaries

In this section we explain the notation and we recall the well-known transformation of linear to iterative recursion.

3.1. About the notation

We use the notation suggested in the Algorithmics paper. In order to be self-contained we list here briefly the conventions, operations, and algebraic laws that we need in the sequel. Some names and symbols have been taken from Bird [4]. The reader is recommended to consult the Algorithmics paper for a thorough motivation and discussion of these topics.

The overall aim of the notational conventions is to make an algebraic manipulation of programs possible and easy, the ideal being that one *calculates* with programs (terms) without a necessity to interprete them. To this end one should allow syntactic ambiguity whenever it does not result in semantic ambiguity, for in this way many trivial transformation steps become superfluous. Imagine for instance what would happen if all parentheses were required in $x+y+\cdots+z$ even when $+$ is associative. The notation below is designed such that reasoning on the function level becomes as easy as reasoning on the point level, cf. "the message" of Backus [2].

Functions and operations

There are binary operations and functions; all functions have a single argument. There is no loss of generality here, because arguments may be structured or tuples, and a function or operation result may itself be a function. The argument of a function and the right argument of an operation must be chosen as large as possible. Function composition (associative!) is the most frequently occurring operation, and is therefore written by juxtaposition, in this paper: a wide space. Meertens [16] proposes to denote application by juxtaposition too, since the resulting syntactic ambiguity is (mostly) not semantically ambiguous: one would have $f(g\,x) = (f\,g)\,x = f\,g\,x$. However, to ease the interpretation of the formulas we will indicate application explicitly by a tiny semicolon, with the convention that its left argument (the function expression) must be chosen as large as possible. (Meertens uses the semicolon merely as a closing parenthesis for which the opening parenthesis must be placed as far as possible to the left.) Thus

$$f\,g\,h\,{:}\,x+y \;=\; \text{“}(f\,g\,h) \text{ applied to } (x+y)\text{”}$$

A binary operation with only one argument (in this paper: the left argument) provided is considered to be a function of its missing argument; it is called a *section.* We shall always enclose a section in parentheses, except for the special operations discussed below. Thus $(x+)\ (y\times){:}\ z \;=\; x+(y\times z)$.

We use symbols like $\oplus$ and $\otimes$ as variables ranging over binary operations, in the same way as f and g are used as variables ranging over functions.

Structures

We use four kinds of structured data, namely *trees, lists, bags* and *sets*; these are generically called *structures.* The type of a structure is denoted $\alpha\bigstar$, where α is the type of the values (elements) contained in the structure; specifically we sometimes write -tree, -list, -bag, or -set instead of $\bigstar$. Operation $\hat{}: \alpha \to \alpha\bigstar$ (written $\hat{}x$ or $\hat{x}$)

forms a singleton structure containing only x. Operation $+\!\!+ : \alpha\bigstar \times \alpha\bigstar \to \alpha\bigstar$ composes two structures of the same kind; in particular, for lists $+\!\!+$ is the append (or concatenation) operation and for sets $+\!\!+$ is the union $\cup$. The difference between the four kinds of structures and between the four $+\!\!+$ operations is algebraically expressed by the laws that hold for $+\!\!+$: for trees $+\!\!+$ satisfies no laws, for lists $+\!\!+$ is associative, for bags $+\!\!+$ is associative and commutative, and for sets $+\!\!+$ is associative, commutative and idempotent (or absorptive):

$$\text{associative} \quad (x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z)$$

$$\text{commutative} \quad x +\!\!+ y = y +\!\!+ x$$

$$\text{idempotent} \quad x +\!\!+ x = x$$

The constant $\emptyset : \alpha\bigstar$ denotes the empty structure; this is formalized by the law

$$\emptyset +\!\!+ x = x = x +\!\!+ \emptyset$$

Thus any tree is a list as well, any list is a bag as well, and any bag is a set as well. This hierarchy of structures is sometimes called the Boom hierarchy, after Boom [7].

Special operations

We need four operations that act on functions and operations rather than on "elements": reduce or insert $(/)$, map $(*)$, filter $(\lhd)$ and left-reduce or left-insert $(\not\rightarrow)$. The first three are special only in that we write them as postfix operations, hence having the highest priority (exactly like primes). Thus

$$\oplus/\ f*\ p\lhd = (\oplus/)\ (f*)\ (p\lhd) \quad \text{and} \quad \oplus/* = (\oplus/)*$$

In other words, one may consider $/, *, \lhd$ as normal binary operations for which the sections $(\oplus/), (f*)$, and $(p\lhd)$ are written without parentheses. The four operations are completely characterized by means of the laws below. (Actually, a theory is being developed in which one can *derive* these laws from the data type definition for the structures; see e.g., Malcolm [15]. It is outside the scope of this paper to do so here.) In the accompanying examples we assume that $+\!\!+$ is associative so that we need not give the parentheses.

map $f*; x$ is the result of applying f to every element of x. Example:

$$f*; \hat{x}_1 +\!\!+ \cdots +\!\!+ \hat{x}_n = \hat{}(f; x_1) +\!\!+ \cdots +\!\!+ \hat{}(f; x_n)$$

The laws are:

$$\text{(map.0)} \quad f*; \emptyset = \emptyset$$

$$\text{(map.1)} \quad f*; \hat{x} = \hat{}(f; x)$$

$$\text{(map.2)} \quad f*; x +\!\!+ y = (f*; x) +\!\!+ (f*; y)$$

reduce $\oplus/\!:\, x$ is the result of inserting $\oplus$ at every construction node of x. Example:

$$\oplus/\!:\, \hat{x}_1 +\!\!+ \cdots +\!\!+ \hat{x}_n \;=\; x_1 \oplus \cdots \oplus x_n$$

Operation $\oplus$ should satisfy at least the same laws as $+\!\!+$ does; otherwise there would arise inconsistencies from the laws below, since they allow us to *prove* (by induction) that $\oplus$ satisfies the laws of $+\!\!+$, cf. Lemma (4). In the same way, $\oplus/\!:\, \emptyset$ has to be the unit of $\oplus$; if operation $\oplus$ has no unit, then we adjoin a fictitious value ω to the domain of $\oplus$ and *define* $\omega \oplus x = \omega = x \oplus \omega$ for all x (like the introduction of ∞ as the unit of the "minimum" operation). The laws are:

(reduce.0) $\oplus/\!:\, \emptyset$ = the (possibly fictitious) unit of $\oplus$

(reduce.1) $\oplus/\!:\, \hat{x} = x$

(reduce.2) $\oplus/\!:\, x +\!\!+ y = (\oplus/\!:\, x) \oplus (\oplus/\!:\, y)$

filter $p\lhd\!:\, x$ is the result of filtering out those elements of x for which predicate p doesn't hold. Example:

$$odd\lhd\!:\, \hat{7} +\!\!+ \hat{2} +\!\!+ \hat{6} +\!\!+ \hat{5} +\!\!+ \hat{4} \;=\; \hat{7} +\!\!+ \hat{5}$$

The laws are:

(filter.0) $p\lhd\!:\, \emptyset = \emptyset$

(filter.1) $p\lhd\!:\, \hat{x} = \hat{x}$ **if** $p\!:\, x$ **else** $\emptyset$

(filter.2) $p\lhd\!:\, x +\!\!+ y = (p\lhd\!:\, x) +\!\!+ (p\lhd\!:\, y)$

left-reduce $(\oplus \not\to e)\!:\, x$ is the result of a left to right traversal over x, taking $\oplus$ at every construction node and starting with initial left argument e. Example:

$$(\oplus \not\to e)\!:\, \hat{x}_1 +\!\!+ \cdots +\!\!+ \hat{x}_n \;=\; (\cdots (e \oplus x_1) \oplus \cdots) \oplus x_n$$

The laws are:

(lreduce.0) $(\oplus \not\to e)\!:\, \emptyset = e$

(lreduce.1) $(\oplus \not\to e)\!:\, \hat{x} = e \oplus x$

(lreduce.2) $(\oplus \not\to e)\!:\, x +\!\!+ y = (\oplus \not\to ((\oplus \not\to e)\!:\, x))\!:\, y$

Here again operation $\oplus$ must be as rich (with respect to commutativity and idempotency) as $+\!\!+$ in order to avoid inconsistencies.

Thus, for $s : \alpha$-bag, p a predicate on α, and $f : \alpha \to \mathbb{N}$, we have

$$+/\ f*\ p\lhd\!:\, s \;=\; \sum_{x \text{ in } s \mid p(x)} f(x)$$

Similarly, for $s : \alpha\bigstar$, $p : \alpha \to \mathbb{B}$, $f : \alpha \to \beta$-set (mapping each element onto a set), if $+\!\!+$ is set-union (i.e., $+\!\!+$ is associative, commutative and idempotent), then

$$+\!\!+/\ f*\ p\lhd\!:\, s \;=\; \bigcup \{ f(x) \mid x \text{ in } s \wedge p(x) \}$$

In the sequel the term $++/\ f*\ p\triangleleft$ will occur over and over again. The new notation is better suited for algebraic calculation than the conventional set-theoretic notation, since there are no bound variables and each "semantic action" is denoted by a distinct syntactic operation for which algebraic laws have been stated above.

Definitions

In order to distinguish between equalities and definitions, we use the symbol $:=$ for the latter and $=$ for the former. By definition, the left-hand side and right-hand side of a definition are equal, so that $:=$ may always be replaced by $=$. Conversely this is not true; e.g., for any object x we have $x = x$, but the definition $x := x$ will in general not define that object called x.

Some more laws

Here we list some laws that we need in the sequel and have already been given in the Algorithmics paper and also by Bird [4]. The promotion and distribution laws may be proved by structural induction; the other ones are immediate by the laws above.

(filter promotion)	$p\triangleleft\ ++/\ =\ ++/\ p\triangleleft*$
(map promotion)	$f*\ ++/\ =\ ++/\ f**$
(reduce promotion)	$\oplus/\ ++/\ =\ \oplus/\ \oplus/*$
in particular	$++/\ ++/\ =\ ++/\ ++/*$
(map distribution)	$f*\ g*\ =\ (f\ g)*$
	$++/\ \hat{}\ =$ id of type $\alpha \to \alpha$
	$++/\ (\hat{})*\ =$ id of type $\alpha\bigstar \to \alpha\bigstar$
	$f*\ \hat{}\ =\ \hat{}\ f$
	$p\triangleleft\ q\triangleleft\ =\ (p \wedge q)\triangleleft$
	$(e\oplus)\ \oplus/\ =\ (\oplus \not\rightarrow e)$ for associative $\oplus$

The derivation of the Branch-and-Bound algorithm in Section 5 triggers the formulation of some specific laws. However, they may be generalized and then turn out to be of a very general nature, comparable to the laws given above. Here we formulate them in the form of a lemma.

(1) Lemma.

(lreduce-join fusion)	$(\oplus \not\rightarrow e)\ ++/\ =\ (\otimes \not\rightarrow e)$
	where $e \otimes x = (\oplus \not\rightarrow e)\colon x$
(lreduce-map fusion)	$(\oplus \not\rightarrow e)\ f*\ =\ (\otimes \not\rightarrow e)$
	where $e \otimes x = e \oplus (f\colon x)$
(lreduce-filter fusion)	$(\oplus \not\rightarrow e)\ p\triangleleft\ =\ (\otimes \not\rightarrow e)$
	where $e \otimes x = e \oplus x$ **if** $p\colon x$ **else** e

Proof. By induction on the structure of the argument. For (lreduce-join fusion):

Basis 1.

$$(\oplus \not\to e)\ {+\!\!+}/ : \emptyset \ = \ (\oplus \not\to e) : \emptyset \ = \ e \ = \ (\otimes \not\to e) : \emptyset$$

Basis 2.

$$(\oplus \not\to e)\ {+\!\!+}/ : \hat{x} \ = \ (\oplus \not\to e) : x \ = \ e \otimes x \ = \ (\otimes \not\to e) : \hat{x}$$

Induction step.

$$(\oplus \not\to e)\ {+\!\!+}/ : s {+\!\!+} t$$

$=$ law (reduce.2) in which $\oplus := {+\!\!+}$

$$(\oplus \not\to e) : ({+\!\!+}/ : s) {+\!\!+} ({+\!\!+}/ : t)$$

$=$ law (lreduce.2)

$$(\oplus \not\to ((\oplus \not\to e)\ {+\!\!+}/ : s))\ {+\!\!+}/ : t$$

$=$ induction hypothesis

$$(\otimes \not\to ((\otimes \not\to e) : s)) : t$$

$=$ law (lreduce.2)

$$(\otimes \not\to e) : s {+\!\!+} t.$$

The other parts are proved similarly. □

Here follow two corollaries. Neither of these corollaries is used in the sequel; however, Corollary (3) is a simplified form of Theorem (21) in Section 5. In that theorem the predicates $p_N, \ldots, p_0$ "change dynamically, during the computation".

(2) Corollary.

$$(\oplus \not\to e)\ {+\!\!+}/\ f* \ p\lhd \ = \ (\otimes \not\to e)$$

where $e \otimes x := (\oplus \not\to e)\, f : x$ **if** $p : x$ **else** e.

(3) Corollary.

$$(\oplus \not\to e)\ {+\!\!+}/\ f_N* \ p_{N-1}\lhd \ \cdots \ {+\!\!+}/\ f_1* \ p_0\lhd \ = \ (\otimes_0 \not\to e)$$

where

$$e \otimes_N x := e \oplus x$$

$$e \otimes_n x := (\otimes_{n+1} \not\to e)\ f_{n+1} : x \ \textbf{if}\ p_n : x \ \textbf{else}\ e$$

(for $n = N-1, \ldots, 0$).

Proof. By induction on $N - n$ it is easy to prove that

$$= \begin{array}{l} (\oplus \not\rightarrow e) \mathbin{+\!\!+}/\ f_N*\ p_{N-1}\triangleleft \cdots \mathbin{+\!\!+}/\ f_1*\ p_0\triangleleft \\ (\otimes_n \not\rightarrow e) \mathbin{+\!\!+}/\ f_n*\ p_{n-1}\triangleleft \cdots \mathbin{+\!\!+}/\ f_1*\ p_0\triangleleft \end{array}$$

using Corollary (2). □

Here are two other useful lemmas.

(4) Lemma. *Let $\oplus$ be associative, commutative, and idempotent, and let m be in s. Then*

$$\oplus/; s \ = \ (m\oplus)\ \oplus/\ ; s$$

Proof. Let $\otimes$ be any operation and consider $\otimes/; s$. Let $\mathbin{+\!\!+}$ be the construction operation of s. Then, *within the argument of* $\otimes/$, operation $\mathbin{+\!\!+}$ may be considered to be as rich as $\otimes$ with respect to associativity, commutativity and idempotency. More precisely,

$$\otimes \text{ associative} \quad \Rightarrow \otimes/; x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) \ = \ \otimes/; (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z$$

$$\otimes \text{ commutative} \ \Rightarrow \otimes/; x \mathbin{+\!\!+} y \ = \ \otimes/; y \mathbin{+\!\!+} x$$

$$\otimes \text{ idempotent} \quad \Rightarrow \otimes/; x \mathbin{+\!\!+} x \ = \ \otimes/; x$$

This is easily proved; e.g., for commutativity we argue

$$= \begin{array}{l} \otimes/; x \mathbin{+\!\!+} y \\ (\otimes/; x)\otimes(\otimes/; y) \end{array}$$

$$= \quad \text{commutativity of } \otimes$$

$$= \begin{array}{l} (\otimes/; y)\otimes(\otimes/; x) \\ \otimes/; y \mathbin{+\!\!+} x. \end{array}$$

Hence, for associative, commutative and idempotent $\oplus$ we have, when m is in s,

$$\oplus/; s \ = \ \oplus/; m \mathbin{+\!\!+} s \ = \ (m\oplus)\ \oplus/; s. \qquad \square$$

The next lemma is formulated for a specific operation $\uparrow$. We suppose that the domain of $\uparrow$ is linearly ordered, say by $\leqslant$; then $x \uparrow y$ is the maximum (with respect to $\leqslant$) of x and y. (One might generalize the lemma by just looking at what properties are used, but we refrain from doing so here.)

(5) Lemma. *Let s be an arbitrary structure, linearly ordered by $\leqslant$, and let m be arbitrary (not necessarily in s). Then*

$$(m\uparrow)\ \uparrow/; s \ = \ (m\uparrow)\ \uparrow/\ (m\leqslant)\triangleleft; s$$

Proof. By induction on the structure of s.

Case $s = \emptyset$. Trivial.

Case $s = \hat{x}$. Immediate from the meaning of $\uparrow$ and $(m\leqslant)\triangleleft$.

Case $s = r \leftrightarrow t$. For brevity define $p := (m\leq)$. Then

$$
\begin{array}{ll}
 & (m\uparrow)\ \uparrow/\cdot\ s \\
= & \\
 & (m\uparrow)\ \uparrow/\cdot\ r \leftrightarrow t \\
= & \\
 & m\uparrow(\uparrow/\cdot r)\uparrow(\uparrow/\cdot t) \\
= & \quad \text{associativity, commutativity, and idempotence of } \uparrow \\
 & m\uparrow((m\uparrow)\ \uparrow/\cdot\ r)\uparrow((m\uparrow)\ \uparrow/\cdot\ t) \\
= & \quad \text{induction hypothesis twice} \\
 & m\uparrow((m\uparrow)\ \uparrow/\ p\lhd\cdot\ r)\uparrow((m\uparrow)\ \uparrow/\ p\lhd\cdot\ t) \\
= & \\
 & m\uparrow(\uparrow/\ p\lhd\cdot\ r)\uparrow(\uparrow/\ p\lhd\cdot\ t) \\
= & \\
 & (m\uparrow)\ \uparrow/\cdot\ ((p\lhd\cdot r)\leftrightarrow(p\lhd\cdot t)) \\
= & \\
 & (m\uparrow)\ \uparrow/\ p\lhd\cdot\ s
\end{array}
$$

This completes the proof. □

3.2. Linear and iterative recursion

In Section 4.1 we shall introduce the notions of "elementwise linear recursive" and "elementwise iterative" and the transformation between them. These concepts are analogous to the well-known notions of "linear recursion" and "iteration" and the corresponding transformation. As an aid to the reader we recall these well-known concepts here, formulated in the current notation.

Consider f_n $(n = 0, 1, \ldots)$ defined by

$$
\begin{array}{lll}
f_0 & := & \text{some given value} \\
f_n & := & h_n\cdot f_{n-1} \quad \text{for } n>0
\end{array}
$$

This definition has a *linear recursive* form (meaning that there is only one occurrence of f in the right-hand side). For example, for the factorial function $f_n = n!$ we have $f_0 = 0! = 1$ and $h_n\cdot x = n\times x$. A definition in *iterative* form (or *tail recursive* form) of g_n such that $f_N = g_0\cdot f_0$, may be derived by aiming at

$$(\bigstar) \qquad g_n\cdot f_n = f_N.$$

In other words, g_n captures the future "extension" of f_n to f_N. For $n = N$ we find from the aim $(\bigstar)$ that $f_N = g_N\cdot f_N$; hence we may define

$$g_N := \mathrm{id}$$

Now we proceed by induction; for $n < N$ we try to establish $(\bigstar)$ from right to left:

$$
\begin{array}{ll}
 & f_N \\
= & \quad \text{induction hypothesis} \\
 & g_{n+1}\cdot f_{n+1} \\
= & \quad \text{definition of } f_{n+1} \\
 & g_{n+1}\ h_{n+1}\cdot f_n
\end{array}
$$

which we want to be equal to $g_n ; f_n$. Hence we may define

$$g_n \;:=\; g_{n+1}\ h_{n+1}$$

and by construction the aim (★) has been achieved. All of the above may be clarified further by noticing that $f_N = h_N\ h_{N-1} \cdots h_1 ; f_0$ (by repeatedly unfolding the definition of f_n), $f_n = h_n \cdots h_1 ; f_0$ and therefore, immediately, $g_n = h_N \cdots h_{n+1}$. For the factorial example we find

$$\begin{aligned} g_N ; x &= x \\ g_n ; x &= g_{n+1} ; (n+1) \times x \\ &= N \times \cdots \times (n+1) \times x \end{aligned}$$

Notice also that g_n has one parameter more than f_n. This parameter is sometimes called the *accumulating parameter*, and the transformation of the linear recursive definition to the iterative definition may be called parameter accumulation: the final result f_N is *accumulated* in this parameter.

The importance of the iterative definition is two-fold. First, it allows us to express precisely "what is to be computed further to obtain f_N when given some f_n". This is a concept that might be useful in an algorithmic analysis; we shall make heavy use of it in the sequel. Secondly, the iterative definition allows for a more efficient implementation, in particular with respect to the storage space. For example, the canonical imperative implementations of linear recursive and iterative definitions read:

fct $f(n : int)$ = **if** $n = 0$ **then** $f0$ **else** $h(n, f(n-1))$

fct $f(N : int)$ = **begin var** $x := f0,\ n := 0;$
while $n < N$ **do** $n, x := n+1,\ h(n+1, x);$
$f := x$
end

4. Backtracking

In this section we discuss Backtracking at a high level of abstraction. We present a definition (or specification) of the problem in Section 4.1, and derive well-known algorithms in Section 4.2. (In Section 6 the algorithms are implemented in a Pascal-like language.)

4.1. Definition and initial exploration

By definition we say that the following kind of problems may be called *Backtracking problems*: the task is to yield any or all of $p \lhd ; s_N$ where s_N is inductively defined

by

$$s_0 \;:=\; \text{some given structure}$$

$$s_n \;:=\; {+\!\!+}/\ f_n{*};\ s_{n-1} \quad \text{for } n>0 \tag{6}$$

Here, f_n is a function that constructs substructures of s_n out of elements of s_{n-1}, and p is some given predicate called the *legality constraint*. Thus we have the typing: $s_n : \alpha\bigstar$, $f_n : \alpha \to \alpha\bigstar$ and $p : \alpha \to \mathbb{B}$, for some α. Mostly α is β-bag or β-set, and then in imperative implementations the members of s_n are represented by an **array of** β. For the example problem PLS we have

$$s_n \;=\; \text{all selections (subsets) of } \{1,\ldots,n\} \;:\mathbb{N}\text{-set-set}$$

so that we may define

$$\begin{aligned} s_0 &:= \hat{\emptyset} && :\mathbb{N}\text{-set-set}\\ f_n;\, x &:= \hat{x} {+\!\!+} \hat{}(x {+\!\!+} \hat{n}) && :\mathbb{N}\text{-set} \to \mathbb{N}\text{-set-set}\\ p &:= (W\geqslant)\ +/\ w* && :\mathbb{N}\text{-set} \to \mathbb{B} \end{aligned}$$

We shall explain the adjective "backtracking" at the end of Section 4.2.

Before attacking the problem of finding an efficient way to compute any or all of $p\lhd; s_N$, we play somewhat with definition (6) and derive alternative but semantically equivalent (i.e., equal) formulations. The reader may notice that the following manipulations would have been practically impossible had we chosen Pascal as the program notation.

First, we repeatedly unfold the definition of s_n:

$$\begin{aligned} & s_N \\ =\ & {+\!\!+}/\ f_N{*};\ s_{N-1} \\ =\ & {+\!\!+}/\ f_N{*}\ {+\!\!+}/\ f_{N-1}{*};\ s_{N-2} \\ & \vdots \\ =\ & \end{aligned}$$

$${+\!\!+}/\ f_N{*}\ \cdots\ {+\!\!+}/\ f_1{*};\ s_0 \tag{7}$$

Next, we apply map promotion ($f_n{*}\ {+\!\!+}/\ =\ {+\!\!+}/\ f_n{**}$) repeatedly, and obtain

$$\begin{aligned} & s_N \\ =\ & \quad \text{by equation (7)} \\ & {+\!\!+}/\ f_N{*}\ {+\!\!+}/\ f_{N-1}{*}\ \cdots\ {+\!\!+}/\ f_1{*};\ s_0 \\ =\ & \quad \text{by (map promotion) on the subterm } f_N{*}\ {+\!\!+}/ \\ & {+\!\!+}/\ {+\!\!+}/\ f_N{**}\ f_{N-1}{*}\ \cdots\ {+\!\!+}/\ f_1{*};\ s_0 \\ & \vdots \\ =\ & \end{aligned}$$

$$({+\!\!+}/)^N\ f_N{*}^N\ f_{N-1}{*}^{N-1}\ \cdots\ f_1{*}^1;\ s_0 \tag{8}$$

Here a superscript n means n-fold repetition (n occurrences after each other). By repeatedly applying map distribution ($f* \; g* = (f \; g)*$) we find from equation (8)

(9) $\quad s_N = (+\!+/)^N \; (\cdots (f_N* \; f_{N-1})* \cdots f_1)*; \; s_0$

Consider once more equation (7):

$$s_N = \underbrace{+\!+/ \; f_N* \cdots +\!+/ \; f_{n+1}*} \; \underbrace{+\!+/ \; f_n* \cdots +\!+/ \; f_1*}; \; s_0$$

The part $+\!+/ \; f_n* \cdots +\!+/ \; f_1*; \; s_0$ clearly equals s_n. Let us give $+\!+/ \; f_N* \cdots +\!+/ \; f_{n+1}*$ the name r_n; so r_n maps s_n onto s_N and has type $\alpha\bigstar \to \alpha\bigstar$:

(10) $\quad s_N = r_n; \; s_n$

(11) $\quad r_n = +\!+/ \; f_N* \cdots +\!+/ \; f_{n+1}*$

It is easy to give an inductive, even iterative, definition of r_n. However, in the following section it turns out to be more helpful to have a name for the contribution to s_N of each element of s_n separately. That is, we are looking for $t_n : \alpha \to \alpha\bigstar$ that satisfy

(12) $\quad s_N = +\!+/ \; t_n*; \; s_n.$

In words, for x from s_n, $t_n; x$ is the contribution of x to s_N. Now we derive an explicit definition for t_n from the desired equation (12). First, for $n = N$ we desire $s_N = +\!+/ \; t_N*; \; s_N$ so that we may define $t_N := \hat{}$. Next, proceeding by induction and therefore assuming that $s_N = +\!+/ \; t_{n+1}*; \; s_{n+1}$, we aim at t_n such that $+\!+/ \; t_n*; \; s_n = s_N$:

s_N

$=$ induction hypothesis

$+\!+/ \; t_{n+1}*; \; s_{n+1}$

$=$ definition of s_{n+1}

$+\!+/ \; t_{n+1}* \; +\!+/ \; f_{n+1}*; \; s_n$

$=$ (map promotion)

$+\!+/ \; +\!+/ \; t_{n+1}** \; f_{n+1}*; \; s_n$

$=$ (reduce promotion)

$+\!+/ \; +\!+/* \; t_{n+1}** \; f_{n+1}*; \; s_n$

$=$ (map distribution)

$+\!+/ \; (+\!+/ \; t_{n+1}* \; f_{n+1})*; \; s_n,$

which we want to be equal to $+\!+/ \; t_n*; \; s_n$. So we may define

$$t_n := +\!+/ \; t_{n+1}* \; f_{n+1}$$

and aim (12) has been achieved. Together:

$$t_N := \hat{}$$

$$t_n := +\!\!+/\ t_{n+1}*\ f_{n+1}. \tag{13}$$

We conclude this exploration by an important observation. In analogy with the notions of linear recursion and iteration (or tail recursion), we call definitions of the form (6) *elementwise linear recursive* and those of the form (13) *elementwise iterative* (or *elementwise tail recursive*). The derivation above of the elementwise iterative definition (13) from the original elementwise linear recursive definition (6) is exactly analogous to the transformation of linear recursion into iteration; see Section 3.

The importance of the elementwise iterative definition is two-fold, as explained in Section 3 for iterative definitions in general. Firstly, t_n is the precise formulation of "the contribution to s_N for x drawn from s_n"; we'll need this concept in the algorithmic analysis below. Secondly, the direct imperative implementations based on the t_n are simpler and more efficient than those based on the s_n; see Section 6.

4.2. Improving the efficiency of the algorithm

The specification of the task, namely to yield any or all of $p\lhd; s_N$ with s_N defined by (6), happens to be executable. Without further knowledge about the f_n and in particular p we cannot give a more efficient program. But note that a direct execution will in many cases take too much time due to exponential growth of the sizes of structures s_n. For example, for PLS structure s_n has 2^n elements. Even if only one element of $p\lhd; s_N$ is requested, and in principle only a small portion of the 2^N elements needs to be inspected in search for one that satisfies p, this will take too much computational time.

One way to reduce the computational time is to reduce the structures s_n without omitting elements that would eventually contribute something to s_N and would pass the filter $p\lhd$. In other words, one should try to promote (parts of) the filter $p\lhd$ as far as possible into the generation of the structures s_n. Darlington [8] has coined the name *filter promotion* for this technique (see also Bird [3]), and Wirth [21, 22] calls it *pruning the search space* and *preselection.* For example, for PLS each element of s_n gives rise to 2^{N-n} elements in s_N, so that omitting it may save quite a lot.

More precisely, one should find predicates p_n that are a necessary condition on elements x of s_n in order that their contribution $t_n; x$ to s_N may satisfy p, i.e.,

$$\emptyset = p\lhd\ +\!\!+/\ t_n*\ (\neg p_n)\lhd; s_n$$

where $\neg$ is the negation operation. For then we have

$$\begin{aligned} & p\lhd; s_N \\ = \quad & p\lhd +\!\!+/\ f_N* \cdots +\!\!+/\ f_n* \cdots +\!\!+/\ f_1*; s_0 \\ = \quad & \text{proved in detail in the appendix, Theorem (27)} \\ & p\lhd\ p_N\lhd\ +\!\!+/\ f_N* \cdots p_n\lhd\ +\!\!+/\ f_n* \cdots p_1\lhd\ +\!\!+/\ f_1*; s_0. \end{aligned} \tag{14}$$

Now notice that

$$
\begin{aligned}
& p_n \lhd \mathbin{+\!\!+}/\ f_n* \\
= \quad & \text{filter promotion} \\
& \mathbin{+\!\!+}/\ p_n \lhd* f_n* \\
= \quad & \text{map distribution} \\
& \mathbin{+\!\!+}/\ (p_n \lhd f_n)*
\end{aligned}
$$

so that by defining $f'_n := p_n \lhd f_n$ we find from (14)

$$(15) \qquad p \lhd\!: s_N \ = \ p \lhd \mathbin{+\!\!+}/\ f'_N* \cdots \mathbin{+\!\!+}/\ f'_1*\!: s_0 .$$

Equation (15) has the same form as equation (7), so that we immediately know an elementwise linear recursive and an elementwise iterative algorithm for computing $p \lhd\!: s_N$; cf. (6) and (13):

$$
\begin{aligned}
& s'_0 := p_0 \lhd\!: s_0 \\
(16) \qquad & s'_n := \mathbin{+\!\!+}/\ f'_n*\!: s'_{n-1} = \mathbin{+\!\!+}/\ (p_n \lhd f_n)*\!: s'_{n-1} = p_n \lhd \mathbin{+\!\!+}/\ f_n*\!: s'_{n-1} \\
& p \lhd\!: s_N = p \lhd\!: s'_N
\end{aligned}
$$

and

$$
\begin{aligned}
& t'_N := \hat{} \\
(17) \qquad & t'_n := \mathbin{+\!\!+}/\ t'_{n+1}* f'_{n+1} = \mathbin{+\!\!+}/\ t'_{n+1}* p_{n+1} \lhd f_{n+1} \\
& p \lhd\!: s_N = p \lhd \mathbin{+\!\!+}/\ t'_0*\!: s_0 .
\end{aligned}
$$

We observe that a further, sometimes important but far less drastic, efficiency improvement is possible. For p_n was supposed to be a condition on elements of $s_n = \mathbin{+\!\!+}/\ f_n*\!: s_{n-1}$, but it is actually used in a filter on $\mathbin{+\!\!+}/\ f_n*\!: s'_{n-1}$ and by construction we know that elements of s'_{n-1} already satisfy p_{n-1}. Therefore, the actual test may sometimes be simplified to, say, q_n; formally q_n should satisfy

$$p_n \lhd \mathbin{+\!\!+}/\ f_n*\!: s_{n-1} \ = \ q_n \lhd \mathbin{+\!\!+}/\ f_n*\ p_{n-1} \lhd\!: s_{n-1} .$$

For our PLS example we have the following. Clearly, a selection out of n objects that already exceeds the limit weight cannot become legal by putting more objects into it. So p_n is the predicate that, exactly like p, says whether the aggregate weight does not exceed the limit. Further, q_n need only check whether the newly added object, if any, does not raise the aggregate weight too much. So for PLS we find $q_n = p_n$. (For the well-known Eight Queens Problem, p_n is the legality constraint that no queen is attacked by any other, whereas q_n only says whether the newly added queen does not attack the others. Here we find $p_n \Rightarrow q_n$ but $q_n \neq p_n$.)

Once one has succeeded in performing a filter promotion along the lines just sketched, one may try to do so a second time, with predicates p'_n say, and find

definitions analogous to (6), (16) and (13), (17) for s''_n, t''_n and f''_n. It turns out that

$$f''_n := p'_n \lhd f'_n = p'_n \lhd p_n \lhd f_n = (p'_n \wedge p_n) \lhd f_n$$

and therefore we conclude that repeated filter promotions may be done at once, taking $p'_n \wedge p_n$ as the filter on s_n. (Here, $p \wedge q$ is a notation of the predicate r defined by $r\, x := (p\, x) \wedge (q\, x)$.) This observation might be formulated as an Algorithmics theorem.

We conclude the discussion by a remark on the mechanical evaluation of "programs" (16) and (17), or, completely unfolded, (15). First of all notice that they just express, mathematically, the result to be computed. There are many ways to evaluate the expressions and thus compute the result. One of them is the full computation of s'_0, followed by the full computation of s'_1, and so on. Another method is as follows. The evaluator tries to output the requested result and therefore computes s'_N only as far as is needed—and this in turn may trigger the computation of s'_{N-1} (only as far as is needed to proceed with the main computation), and so on. This method of evaluation is called *lazy* or *demand driven* evaluation and is more or less the same as normal order reduction in the Lambda Calculus. Under lazy evaluation the computations according to (16), (17) and (15) behave as a backtracking process. In effect, the process repeatedly extends (by f_n) an already found partial solution (elements of s'_{n-1}) and checks whether the extensions pass the filter p_n. This is done in a depth-first way, so that upon a failure of an extension to pass the filter, the process "backtracks" to the last passed point where further alternatives are still available.

5. Branch-and-Bound

In the previous section we discussed the problem of delivering any or all of $p \lhd s_N$. Now we consider the task of computing the optimal element of $p \lhd s_N$. To this end we assume that there exists a linear order $\leqslant$ on the element of s_N and that $\uparrow / \; p \lhd s_N$ is requested; operation $\uparrow$ is defined by

$$x \uparrow y = \text{the maximum of } x \text{ and } y \text{ with respect to } \leqslant$$

Without further knowledge we cannot, of course, give a more efficient algorithm than the specification $\uparrow / \; p \lhd s_N$ itself. So let us assume that we know something more. First of all, as in the previous section there may exist predicates p_n that are a necessary condition for elements of s_n in order that their contribution to s_N may satisfy p. Then we can apply the technique of filter promotion or preselection. The improved algorithm, however, has still exactly the same structure as the original one: the functions f_n are simply replaced by $f'_n = p_n \lhd f_n$. We shall not deal with this aspect any further. Secondly, there may exist predicates $p_{n,m}$ that are a necessary condition on elements of s_n in order that their contribution to s_N may dominate m; here m is some element that plays the role of "the currently found maximum of s_N" and informally $p_{n,m}$ says whether an element of s_n "looks promising" with

respect to m. It is this knowledge that we are going to exploit in the sequel.

At first sight it seems that we still can apply the technique of filter promotion. For, when given m in $p\triangleleft; s_N$, we have

$$\uparrow/\ p\triangleleft; s_N$$

$=$ Lemma (4)

$$(m\uparrow)\ \uparrow/\ p\triangleleft; s_N$$

$=$ Lemma (5) in which $s := p\triangleleft; s_N$

$$(m\uparrow)\ \uparrow/\ (m\leqslant)\triangleleft\ p\triangleleft; s_N$$

$=$ "filter promotion" as in Section 4

$$(m\uparrow)\ \uparrow/\ (m\leqslant)\triangleleft\ p\triangleleft\ p_{N,m}\triangleleft\ +\!+/\ f_N* \cdots p_{1,m}\triangleleft\ f_1*\ p_{0,m}\triangleleft; s_0.$$

However, the problem is that we want the argument m in $p_{n,m}$ to change dynamically as the computation proceeds: it should be updated as soon as a new currently maximal element is found. Had we had dynamically assignable variables at our disposal, we could have written:

var $m :=$ some (fictitious) element of $p\triangleleft; s_N$;
fct $test(x) :=$ **if** $m \leqslant x$ **then** $m := x$; *true* **else** *false* **fi**;
result-is $(m\uparrow)\ \uparrow/\ test\triangleleft\ p\triangleleft\ p_{N,m}\triangleleft\ +\!+/\ f_N* \cdots p_{1,m}\triangleleft\ +\!+/\ f_1*\ p_{0,m}\triangleleft; s_0.$

Under lazy evaluation of the **result-is** expression, this program specifies the desired computation. Our aim, now, is to express and formally derive in a functional, algorithmic setting what is intended by the above imperative program.

The assumed property of $p_{n,m}$ is formalized as:

(18) $$\emptyset\ =\ (m\leqslant)\triangleleft\ p\triangleleft\ +\!+/\ t_n*\ (\neg p_{n,m})\triangleleft; s_n$$

where $\neg$ is the negation operation. As in the previous section, and in detail shown in Lemma (26) in Appendix A, we find

(19) $$(m\leqslant)\triangleleft\ +\!+/\ t_n*; s\ =\ (m\leqslant)\triangleleft\ +\!+/\ t_n*\ p_{n,m}\triangleleft; s \quad \text{for } s \subseteq s_n$$

where $x \subseteq y$ means that x is a (possibly noncontiguous) substructure (i.e. subset, subbag, subsequence) of y. As a preparatory step we derive from this, for $\hat{x} \subseteq s_n$:

$$(m\uparrow)\ \uparrow/\ p\triangleleft\ t_n; x$$

$=$ Lemma (5) at the left part

$$(m\uparrow)\ \uparrow/\ (m\leqslant)\triangleleft\ p\triangleleft\ +\!+/\ t_n*; \hat{x}$$

$=$ equation (19) together with the law $p\triangleleft\ q\triangleleft\ =\ q\triangleleft\ p\triangleleft$

$$(m\uparrow)\ \uparrow/\ (m\leqslant)\triangleleft\ p\triangleleft\ +\!+/\ t_n*\ p_{n,m}\triangleleft; \hat{x}$$

$=$ Lemma (5)

$$(m\uparrow)\ \uparrow/\ p\triangleleft\ +\!+/\ t_n*\ p_{n,m}\triangleleft; \hat{x}$$

$=$

(20) $$(m\uparrow)\ \uparrow/\ p\triangleleft\ t_n; x\ \textbf{if}\ p_{n,m}; x\ \textbf{else}\ m$$

This equation will allow us to skip elements x of s_n that do not look promising with respect to m. We call these elements *bad*.

Now we tackle the problem of deriving an efficient algorithm for the computation of $\uparrow/\ p\lhd;\ s_N$, i.e., $(m\uparrow)\ \uparrow/\ p\lhd;\ s_N$ where m is some element of $p\lhd;\ s_N$, or, slightly more generally, where m is some (fictitious) element satisfying

$$(m\uparrow)\ \uparrow/\ p\lhd;\ s_N\ =\ \uparrow/\ p\lhd;\ s_N$$

The key to the solution is to sequentialize the computation so as to be able to control future computations by "the currently found maximum m' of s_N". The sequentialization of $(m\uparrow)\ \uparrow/$ is $(\uparrow \not\to m)$; see Section 3. Here follow the initial steps of a derivation of the desired algorithm. These steps motivate the formulation and proof of Theorem (21) below.

$$\begin{array}{ll} & \text{requested value} \\ = & \\ & (m\uparrow)\ \uparrow/\ p\lhd;\ s_N \\ = & \\ & (\uparrow \not\to m)\ p\lhd;\ s_N \\ = & \\ & (\uparrow \not\to m)\ p\lhd\ {+\!\!+}/\ f_N*;\ s_{N-1} \\ = & \\ & \ldots \end{array}$$

At this point we wish to promote $(\uparrow \not\to m)$ to s_{N-1} in order to skip bad elements of s_{N-1} and not subject them to $p\lhd\ {+\!\!+}/\ f_N*$. The promotion laws for $\not\to$ in Section 3 were "invented" for this very purpose here. Applying Lemma (1) we get:

$$\begin{array}{ll} & \vdots \\ = & \\ & (\oplus \not\to m);\ s_{N-1} \\ & \text{with } m'\oplus x\ :=\ (\uparrow \not\to m')\ p\lhd\ f_N;\ x \\ = & \quad \text{aiming at the use of (20), rewrite the rhs of } := \text{ of the previous line,} \\ & \quad \text{using } (e\uparrow)\ \uparrow/\ =\ (\uparrow \not\to e) \text{ and } t_{N-1} = f_N \\ & (\oplus \not\to m);\ s_{N-1} \\ & \text{with } m'\oplus x\ =\ (m'\uparrow)\ \uparrow/\ p\lhd\ t_{N-1};\ x \\ & \qquad\qquad = \quad \text{equation (20), noting that } \hat{x} \subseteq s_{N-1} \\ & \qquad\qquad\quad (m'\uparrow)\ \uparrow/\ p\lhd\ t_{N-1};\ x \textbf{ if } p_{N-1,m'};\ x \textbf{ else } m' \\ & \qquad\qquad = \ (\uparrow \not\to m')\ p\lhd\ f_N;\ x \textbf{ if } p_{N-1,m'};\ x \textbf{ else } m' \\ = & \\ & \ldots \end{array}$$

and we see that bad elements of s_{N-1} are skipped; the search space is bounded more and more during the search $(\oplus \not\to m)$. Of course, we wish to do the same with

bad elements of s_{N-2} and therefore we continue the derivation:

$$
\begin{array}{ll}
 & \vdots \\
= & \\
 & (\oplus \not\triangleright m) \mathbin{+\!\!+\!/} f_{N-1}*;\ s_{N-2} \quad \text{with } \oplus \text{ as before} \\
= & \quad \text{fusion of } (\oplus \not\triangleright m) \text{ with } \mathbin{+\!\!+\!/} f_{N-1}* \text{ using Lemma (1)} \\
 & (\otimes \not\triangleright m);\ s_{N-2} \\
 & \text{with } m' \otimes x := (\oplus \not\triangleright m')\ f_{N-1};\ x \\
 & \qquad = \quad \text{slight generalization of the derivation so far} \\
 & \qquad\quad (m'\uparrow)\ \uparrow\!/\ p\triangleleft\ t_{N-2};\ x \\
 & \qquad = \quad \text{equation (20), noting that } \hat{x} \subseteq s_{N-2} \\
 & \qquad\quad (m'\uparrow)\ \uparrow\!/\ p\triangleleft\ t_{N-2};\ x \textbf{ if } p_{N-2,m'};\ x \textbf{ else } m' \\
 & \qquad = (\oplus \not\triangleright m')\ f_{N-1};\ x \textbf{ if } p_{N-2,m'};\ x \textbf{ else } m' \\
= & \\
 & \ldots
\end{array}
$$

and it should be clear that we can continue in this way. We shall now do it all at once: we generalize operations $\oplus, \otimes, \ldots$ to an inductively defined sequence $\oplus_N, \oplus_{N-1}, \oplus_{N-2}, \ldots$ and formulate (the required slight generalization of) the transformation in a theorem.

Define operations $\oplus_n$ as follows:

$$
\begin{aligned}
m \oplus_N x &:= m \uparrow x \textbf{ if } (p \wedge p_{N,m});\ x \textbf{ else } m \\
m \oplus_n x &:= (\oplus_{n+1} \not\triangleright m)\ f_{n+1};\ \textbf{if } p_{n,m};\ x \textbf{ else } m \quad (\text{for } n = N-1, \ldots, 0).
\end{aligned}
$$

(21) Theorem. *For all n and all s with $0 \leq n \leq N$ and $s \subseteq s_n$:*

$$(m\uparrow)\ \uparrow\!/\ p\triangleleft\ \mathbin{+\!\!+\!/}\ t_n*;\ s\ =\ (\oplus_n \not\triangleright m);\ s.$$

Proof. By induction on $N - n$.

Basis. For $s \subseteq s_N$:

$$
\begin{array}{ll}
 & (m\uparrow)\ \uparrow\!/\ p\triangleleft\ \mathbin{+\!\!+\!/}\ t_N*;\ s \\
= & \\
 & (m\uparrow)\ \uparrow\!/\ p\triangleleft;\ s \\
= & \\
 & (\uparrow \not\triangleright m)\ p\triangleleft;\ s \\
= & \quad \text{(lreduce-filter fusion), i.e., Lemma (1)} \\
 & (\oplus \not\triangleright m);\ s \\
 & \text{with } m' \oplus x := m' \uparrow x \textbf{ if } p;\ x \textbf{ else } m'
\end{array}
$$

$$= (m'\uparrow)\ \uparrow/\ p\lhd;\ \hat{x}$$

$$= \quad \text{equation (20), noting that } \hat{x} \subseteq s \subseteq s_N$$

$$(m'\uparrow)\ \uparrow/\ p\lhd;\ \hat{x}\ \textbf{if}\ p_{N,m'};\ x\ \textbf{else}\ m'$$

$$= m' \uparrow x\ \textbf{if}\ (p \wedge p_{N,m'});\ x\ \textbf{else}\ m'$$

$$= m' \oplus_N x$$

$$= (\oplus_N \not\to m);\ s.$$

Induction step (from n to $n-1$). For $s \subseteq s_{n-1}$:

$$(m\uparrow)\ \uparrow/\ p\lhd\ +\!\!+/\ t_{n-1}*;\ s$$

$$= (m\uparrow)\ \uparrow/\ p\lhd\ +\!\!+/\ t_n*\ +\!\!+/\ f_n*;\ s$$

$$= \quad \text{induction hypothesis for } n$$

$$(\oplus_n \not\to m)\ +\!\!+/\ f_n*;\ s$$

$$= \quad \text{(lreduce join, lreduce-map fusion), i.e., Lemma (1)}$$

$$(\otimes \not\to m);\ s$$

$$\text{with } m' \otimes x := (\oplus_n \not\to m')\ f_n;\ x$$

$$= \quad \text{induction hypothesis}$$

$$(m'\uparrow)\ \uparrow/\ p\lhd\ +\!\!+/\ t_n*\ f_n;\ x$$

$$= (m'\uparrow)\ \uparrow/\ p\lhd\ t_{n-1};\ x$$

$$= \quad \text{equation (20), noting that } \hat{x} \subseteq s \subseteq s_{n-1}$$

$$(m'\uparrow)\ \uparrow/\ p\lhd\ t_{n-1};\ x\ \textbf{if}\ p_{n-1,m'};\ x\ \textbf{else}\ m'$$

$$= \quad \text{back again}$$

$$(\oplus_n \not\to m')\ f_n;\ x\ \textbf{if}\ p_{n-1,m'};\ x\ \textbf{else}\ m'$$

$$= m' \oplus_{n-1} x$$

$$= (\oplus_{n-1} \not\to m);\ s$$

This completes the proof. □

As an immediate corollary we have that, when m is the smallest with respect to $\leqslant$ or when m is in $p\lhd;\ s_N$,

$$(22) \qquad p\lhd;\ s_N = (m\uparrow)\ \uparrow/\ p\lhd;\ s_N = (\oplus_0 \not\to m);\ s_0$$

Algorithm $(\oplus_0 \not\to m)$ describes precisely the desired computation: each operation $\oplus_n$ carries in its left argument the current maximum and skips those elements (i.e., does not subject them to further computation) that do not look promising with respect to the current maximum.

6. Imperative implementations

In this section we give some imperative implementations of the algorithms derived in the previous two sections. It turns out that the elementwise iterative version has a conventional implementation, whereas the elementwise linear recursive version looks unconventional. We also provide assertions needed for the correctness proofs, and it appears that the invariance of the assertions can be verified by precisely the derivations of the previous sections.

For reasons of time efficiency we want to describe the computation that corresponds to the demand driven (or lazy) evaluation. Also, for reasons of storage efficiency (and again to simulate the demand driven evaluation as far as possible), we shall use one global variable x in which the elements of s_n are built in succession (so actually we assume that each s_n is a list, bag or set, and not a tree); the structures s_n are not stored in any other way.

We consider programs (16), (17) and (22). In the imperative programs $f(n), p(n), s'(n)$ correspond to f_n, p_n and s'_n from the algorithmic expressions. For simplicity we assume that $p_0 \lhd s_0 = \hat{x}_0$ (a singleton).

6.1. Implementation of (16)

Coroutines make an imperative description of demand driven evaluation easy. A coroutine differs from a subroutine only in that it may "return" several times during the execution of its body; whenever it is re-invoked it continues the execution at the last point of return. The notation below is ad-hoc but self explanatory.

```
var x;
fct p():bool = {yields p: x};
fct p(n:int):bool = {yields p_n: x};
coroutine f(n:int) =
  {returns each element of f_n: x in succession in var x};
coroutine s'(n:int) =
  {returns each element of s'_n in succession in var x}
  if n = 0
  then begin x := x0; return end
  else for each return of s'(n-1) do
         for each return of f(n) do
           if p(n) then return;
...
for each return of s'(N) do if p() then print
(or: for the first return of s'(N) do if p() then print)
...
```

Thus an expression like $++/\ f*:\ s$ is transcribed as

```
for each return of s̃ do
  for each return of f̃ do ...
```

where $\tilde{s}$ and $\tilde{f}$ are coroutines implementing s and f.

For the PLS example we may choose to represent elements x from s_n by an array a such that $a[i] = (i \text{ belongs to } x)$, together with a variable wgt that equals the aggregate weight of x. For the representation of elements from s_n only $a[1], \ldots, a[n]$ and wgt are significant; $a[n+1], \ldots, a[N]$ are meaningless. (Hence, in the context of $n = 0$ the initialization $x := x0$ is implemented by **skip**.) The problem dependent definitions now read as follows.

```
var x: record a: array [1..N] of bool;
               wgt: real
      end;
fct p() :- superfluous, or identically true;
fct p(n: int): bool = (x.wgt ≤ W);
coroutine f(n: int) =
  begin x.a[n] := true; wgt := wgt + w(n); return;
        wgt := wgt − w(n);
        x.a[n] := false; return
  end;
proc print = write(x.a[1..N]).
```

6.2. *Another implementation of* (16)

Coroutines are not readily available. Therefore we present here an implementation not using them. At first sight this seems very problematic, for the imperative program should describe that the computation corresponding to $++/\ f'_n*$ is to be performed for each result (element) of s'_{n-1}. The results of s'_{n-1}, however, are stored one after the other in **var** x. Nevertheless this can be done satisfactorily. The idea is to pass $++/\ f'_n*$ as a "continuation parameter" to the procedure that implements s'_{n-1}. Whenever this procedure is about to yield a result (one element of s'_{n-1}), it should now invoke the continuation parameter. To explain this more precisely, we express this transformation first in the algorithmic notation.

From equation (10), $s_N = r'_n ; s'_n$, we see that the continuation of s'_n in the computation of s_N is r'_n. (The primes intend to indicate that the p_n are taken into account; cf. (16) versus (6), and (17) versus (13).) We wish to define some s''_n that, given r'_n as continuation parameter, produces s_N. So we aim at

$$s''_n ; r_n = s_N .$$

From this aim one derives quite easily the definition

$$s''_0 ; c := c ; s_0 = c ; \hat{x}_0$$

$$s''_n ; c := s''_{n-1} ; (c\ ++/\ f'_n*)$$

$$p\lhd ; s_N = p\lhd\ s''_N ; r'_N = p\lhd\ s''_N ; id = s''_N ; p\lhd .$$

(The very last equation is justified by an inductive proof of $f\ s''_n;\ c\ =\ s''_n;\ (f\ c)$ for all f and c.) Similarly we assume that also $c\ +\!+/\ f'_n*$ can be turned inside-out: that is there exists some f''_n for which $f''_n;\ c\ =\ c\ +\!+/\ f'_n*$. The imperative implementation now suggests itself:

```
var x;
proc f''(n: int; proc c) =
  {yields in succession in var x each element of (f''_n; c); x};
proc s''(n: int; proc c) =
  {yields in succession in var x each element of s''_n; c}
  if n = 0 then x := x0; c else s''(n − 1, proc: f''(n, c));
...
s''(N, proc: if p( ) then print)
...
```

Specifically for PLS the problem dependent definitions read:

```
proc f''(n: int; proc c) =
  begin x.a[n] := true; wgt := wgt + w(n);
        if p(n) then c;
        wgt := wgt − w(n);
        x.a[n] := false; c
end;
```

and everything else (namely $x, p(), p(n)$ and *print*) is the same as for the coroutine implementation.

6.3. Implementation of (17)

The elementwise iterative definition of t'_n allows for a straightforward implementation. In the absence of further knowledge or assumptions about the f_n, we still use the coroutine implementation for these. Note however that very often the iteration "**for** each return of $f(n)$ **do**" can be formulated as a proper iteration in which x is assigned successively each element of $f_n;\ x$.

```
x, p(), p(n: int), f(n: int) :- as in Section 6.1
proc t'(n: int) =
  {stores each element of t'_N; x in succession in var x;
  or rather, prints the elements of p◁ t'_N; x in succession}
  if n = N
  then {ready; or rather:} if p() then print
  else for each return of f(n+1) do
         if p(n+1) then t'(n+1);
...
x := x0; t'(0)
...
```

Specifically for PLS, each $f_n; x$ consists of two elements so that "**for** each return of $f(n+1)$ **do**" can be unfolded in place, giving:

```
proc t'(n:int) =
  if n = N
  then print
  else begin x.a[n] := true; wgt := wgt + w(n);
             if p(n+1) then t'(n+1);
             wgt := wgt - w(n);
             x.a[n] := false; t'(n+1)
       end;
```

6.4. Implementation of (22)

The implementation of

$$(\oplus \not\to m);\ \hat{x}_1 +\!\!+ \cdots +\!\!+ \hat{x}_n \ = \ (\cdots(m \oplus x_1) \oplus \cdots) \oplus x_n$$

suggests itself: an iteration of $\oplus$ over $x_1, \ldots, x_n$ with one global variable **var** m in which $\oplus$ finds its left argument stored, and consequently should leave its result. We choose $op(n)$ as the Pascal-like name of operation $\oplus_n$.

```
x, p(), f'(n) :- as before
fct p(n:int, m:elt):bool = {yields p_{n,m}; x};
proc op(n:int) =
  {yields the result of (⊕_n ↛ m); x in var m}
  if n = N
  then if p() and p(N, m) then m := m ↑ x else m := m
  else if p(n, m)
       then for each return of f'(n+1) do op(n+1)
       else m := m;
...
x := x0; m := some (fictitious) value such that (m↑) p◁; s_N = p◁; s_N ;
op(0); write(m)
```

Specifically for POS we instantiate the above to:

```
var x, m : record a:array [1..N] of bool;
                  wgt:real
           end;
fct p1(n:int):bool = x.wgt ≤ W;
fct p2(n:int, m:...) = x.wgt + Σ_{i=n+1}^{N} w(i) ≥ m.wgt;
proc op(n:int) =
  if n = N
  then if {p2(N, m) and} m.wgt ≤ x.wgt
       then m := x else skip
  else if p2(n, m) then
```

```
        begin x.a[n+1] := true; x.wgt := x.wgt + w(n+1);
              if p1(n+1) then op(n+1);
              x.wgt := x.wgt - w(n+1);
              x.a[n+1] := false; op(n+1)
        end;
...
skip {i.e., x := x0}; m.wgt := 0;
op(0); write(m.a[1..N])
```

7. Concluding remarks

By means of the examples of Backtracking and Branch-and-Bound, we have shown how program derivations may proceed in an algebraic way. It was quite essential, from a practical point of view, that the program texts didn't grow too long. Moreover, and at least as importantly, it turned out that the concepts formalized by the squiggles /, *, $\lhd$, $\not\rhd$, ++ were rightly chosen in the sense that they appear to be generally applicable and have easy-to-apply laws. A derivation of the programs of Section 6 would have been impossible if a Pascal-like notation was used from the very beginning.

Since the writing of this paper (beginning of 1988) much work has been done in order to make the Algorithmics style of programming a worthwhile alternative to various, more traditional styles of programming. Bird [5] has developed a series of high-level theorems that may be successfully applied in the derivation of algorithms on lists and even arrays. Malcolm [14, 15] has given a categorical foundation, and he has shown that for *any* data type definition ("initial/final algebra") some laws come for free; in particular the (reduce/map/filter promotion) and the (lreduce-reduce/map/filter fusion) laws of Section 3.1. Thus, there is a general pattern in most of the laws that makes them easy to remember (and to discover!). Meertens [17] shows that for "homomorphisms" (and even "paramorphisms") on such data types a lot of identities that used to be proved by induction (as in this paper) can also be justified by more "calculational" steps. Apart from this kind of foundational work, a lot of specialized theories are being developed, each for a particular data type or problem type; see in particular Bird [4–6].

In view of the above achievements the question suggests itself whether there is some more basic theory from which one can obtain our theorems by a few simple calculation steps.

Although Backtracking and Branch-and-Bound have been chosen only to conduct the experiment of an Algorithmics development, it is interesting to compare the results with other approaches to these problems. We mention some of them. First of all there are the traditional imperative developments, e.g., by Wirth [22] and many others. They arrive at programs that we have given in Section 6.3. The invariance of the assertions that we have given for the programs can be shown

easily, using the equalities derived in Sections 4 and 5; it even seems inescapable to use (or re-derive) these equalities. So it appears that these reasonings need to occur in the traditional program derivations, although in disguised form and sometimes imprecise or incomplete. Next we mention Wadler [20]. He shows how to obtain our ultimate program for Backtracking (not Branch-and-Bound) by a transformation of a program that uses a nondeterministic **choice** operation which has to avoid branches of the computation path that end in **fail**. We have reasoned about the set of *all* solutions in a purely mathematical way; no concept of a choice-making demon has ever been needed. Finally, Smith [19] comes to similar results as ours by an automatable strategy for designing subspace generators. His "generators" correspond to the coroutines of Section 6.1; these are characterized by pre- and post-conditions and have very much the flavor of imperative style programming rather than the flavor of mathematical expressions, like our formulas in Sections 4 and 5.

Appendix A. Some proofs

We shall derive equation (14) of Section 4 formally. We choose to formalize the assumption "p_n is a necessary condition on the elements x of s_n in order that their contribution $t_n \fatsemi x$ to s_N may satisfy p" by

(23) $\quad \emptyset \;=\; p\lhd \mathbin{+\!\!+}/\ t_n* \ (\neg p_n)\lhd \fatsemi\ s_n.$

Note that if we had chosen the formalization as $\emptyset = p\lhd \mathbin{+\!\!+}/\ t_n* \ (\neg p_n)\lhd \fatsemi\ s$ for *all* s, then we would immediately have Lemma (26). We feel, however, that (23) expresses the assumption most clearly and is much weaker, more general, than the alternative.

First we define a relation $\subseteq$ between structures (namely 'inclusion' for sets, 'noncontiguous subsequence' for lists).

(24) Definition. The relation $\subseteq$ is the smallest relation between structures, such that
(1) $\emptyset \subseteq \emptyset$,
(2) $\emptyset \subseteq \hat{x}$ and $\hat{x} \subseteq \hat{x}$,
(3) $s' \mathbin{+\!\!+} t' \subseteq s \mathbin{+\!\!+} t$ whenever $s' \subseteq s$ and $t' \subseteq t$.

(25) Lemma. *Relation* $\subseteq$ *satisfies the following properties.*
(1) $s \subseteq s$,
(2) $r \subseteq s \subseteq t$ *implies* $r \subseteq t$,
(3) $s \subseteq s \mathbin{+\!\!+} t$ *and* $t \subseteq s \mathbin{+\!\!+} t$,
(4) $s \subseteq t \subseteq s$ *implies* $s = t$,
(5) $s \subseteq t$ *implies* $p\lhd \fatsemi\ s \subseteq p\lhd \fatsemi\ t$,
(6) $p\lhd \fatsemi\ s \subseteq s$,
(7) $s \subseteq t$ *implies* $\mathbin{+\!\!+}/\ f* \fatsemi\ s \subseteq \mathbin{+\!\!+}/\ f* \fatsemi\ t$.

Proof. Most proofs are straightforward by induction. By way of illustration we prove (7) by induction on the inference of $s \subseteq t$.

Case $s \subseteq t$ on account of (24.1): $s = \emptyset = t$. Trivial.

Case $s \subseteq t$ on account of (24.2): both for $s = \emptyset$, $t = \hat{x}$ and for $s = \hat{x} = t$ trivial.

Case $s \subseteq t$ on account of (24.3): $s = s_1 +\!\!+ s_2$, $t = t_1 +\!\!+ t_2$ and $s_i \subseteq t_i$ for $i = 1, 2$. Now

$$
\begin{aligned}
& +\!\!+/\ f*;\ s_1 +\!\!+ s_2 \\
= \quad & \text{(map.2) and (reduce.2)} \\
& (+\!\!+/\ f*;\ s_1) +\!\!+ (+\!\!+/\ f*;\ s_2) \\
\subseteq \quad & \text{induction hypothesis and (24.3)} \\
& (+\!\!+/\ f*;\ t_1) +\!\!+ (+\!\!+/\ f*;\ t_2) \\
= \quad & \text{(map.2) and (reduce.2)} \\
& +\!\!+/\ f*;\ t_1 +\!\!+ t_2
\end{aligned}
$$

This completes the proof. □

(26) Lemma. *Under the assumption* (23), *for all* $s \subseteq s_n$:

$$p\triangleleft\ +\!\!+/\ t_n*;\ s\ =\ p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ s$$

Proof. By induction on the structure of s.

Case $s = \emptyset$. Trivial.

Case $s = \hat{x}$. Then

$$
\begin{aligned}
& p\triangleleft\ +\!\!+/\ t_n*;\ \hat{x} \\
= \quad & \\
& p\triangleleft\ +\!\!+/\ t_n*\ (p_n \vee \neg p_n)\triangleleft;\ \hat{x} \\
= \quad & \\
& p\triangleleft\ +\!\!+/\ t_n*;\ ((p_n\triangleleft;\ \hat{x}) +\!\!+ (\neg p_n\triangleleft;\ \hat{x})) \\
= \quad & \\
& (p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ \hat{x}) +\!\!+ (p\triangleleft\ +\!\!+/\ t_n*\ (\neg p_n)\triangleleft;\ \hat{x}) \\
= \quad & \text{assumption (23)} \\
& p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ \hat{x}.
\end{aligned}
$$

Case $s = r +\!\!+ t$. Now

$$
\begin{aligned}
& \text{left-hand side} \\
= \quad & \text{(map.2), (reduce.2) and (filter.2)} \\
& (p\triangleleft\ +\!\!+/\ t_n*;\ r) +\!\!+ (p\triangleleft\ +\!\!+/\ t_n*;\ t) \\
= \quad & \text{induction hypothesis, noticing that } r \subseteq s_n \text{ by (25.3) and (25.2)} \\
& (p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ r) +\!\!+ (p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ t) \\
= \quad & \text{(map.2), (reduce.2) and (filter.2)} \\
& p\triangleleft\ +\!\!+/\ t_n*\ p_n\triangleleft;\ r +\!\!+ t \\
= \quad & \\
& \text{right-hand side.}
\end{aligned}
$$

This completes the proof. (The reasoning for the case $s = \hat{x}$ fails for arbitrary s; otherwise that reasoning would be an induction-less proof of the lemma.) □

(27) Theorem. *Under assumption* (23), *equation* (14) *holds true.*

Proof. Define

$$s'_0 := p_0 \lhd ; s_0$$

$$s'_n := p_n \lhd +\!\!+/ f_n* ; s'_{n-1} = p_n \lhd +\!\!+/ f_n* \cdots p_1 \lhd +\!\!+/ f_1* \; p_0 \lhd ; s_0$$

We show by induction on n that

$$p \lhd ; s_N = p \lhd +\!\!+/ t_n* ; s'_n \text{ and}$$

$$s'_n \subseteq s_n.$$

Basis.

$$p \lhd ; s_N = p \lhd +\!\!+/ t_0* ; s_0 = \{\text{Lemma (26)}\}\ p \lhd +\!\!+/ t_0* ; s'_0.$$

$$s'_0 = p_0 \lhd ; s_0 \subseteq \{\text{Lemma (25.6)}\}\ s_0.$$

Induction step. For $p \lhd ; s_N$ we argue:

$p \lhd ; s_N$

= induction hypothesis

$p \lhd +\!\!+/ t_n* ; s'_n$

= definition of t_n

$p \lhd +\!\!+/ (+\!\!+/ t_{n+1}* \; f_n)* ; s'_n$

= (map distribution), (map promotion)

$p \lhd +\!\!+/ t_{n+1}* +\!\!+/ f_n* ; s'_n$

= induction hypothesis gives $s'_n \subseteq s_n$,
Lemma (25.7) gives $+\!\!+/ f_n* ; s'_n \subseteq +\!\!+/ f_n* ; s_n = s_{n+1}$;
apply Lemma (26)

$p \lhd +\!\!+/ t_{n+1}* \; p_{n+1} \lhd +\!\!+/ f_n* ; s'_n$

= definition of s'_{n+1}

$p \lhd +\!\!+/ t_{n+1}* ; s'_{n+1}.$

And for s'_{n+1} we calculate:

$$
\begin{aligned}
& s'_{n+1} \\
= \quad & \text{definition} \\
& p_{n+1} \lhd +\!\!+/\ f_{n+1}*;\ s'_n \\
\subseteq \quad & \text{Lemma (25.7), induction hypothesis} \\
& p_{n+1} \lhd +\!\!+/\ f_n*;\ s_n \\
\subseteq \quad & \text{Lemma (25.6)} \\
& +\!\!+/\ f_n*;\ s_n \\
= \quad & \text{definition} \\
& s_{n+1}.
\end{aligned}
$$

This completes the proof. □

References

[1] S. Alagic and M.A. Arbib, *The Design of Well-Structured and Correct Programs*, Texts and Monographs in Computer Science (Springer, Berlin, 1978).

[2] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (8) (1978) 613-641.

[3] R.S. Bird, The promotion and accumulation strategies in transformational programming, *ACM Trans. Programming Languages Systems* **6** (4) (1984) 487-504; Addendum: *ACM Trans. Programming Languages Systems* **7** (3) (1985) 490-492.

[4] R.S. Bird, An introduction to the theory of lists, in: M. Broy, ed., *Logic of Programming and Calculi of Discrete Design* (Springer, Berlin, 1987) 3-42; also: Tech. Monograph PRG-56, Oxford University, Computing Laboratory, Programming Research Group (1986).

[5] R.S. Bird, Lecture notes on constructive functional programming, in: M. Broy, ed., *Constructive Methods in Computing Science*, International Summer School directed by F.L. Bauer et al., NATO Advanced Science Institute Series F: Computer and System Sciences (Springer, Berlin, 1989).

[6] R.S. Bird, J. Gibbons and G. Jones, Formal derivation of a pattern matching algorithm, *Sci. Comput. Programming* **12** (1989) 93-104.

[7] H.J. Boom, Further thoughts on Abstracto, Working Paper ELC-9, IFIP WG 2.1 (1981).

[8] J. Darlington, A synthesis of several sorting algorithms, *Acta Inform.* **11** (1) (1978) 1-30.

[9] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, *Comm. ACM* **18** (8) (1975) 453-457.

[10] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[11] M.S. Feather, A survey and classification of some program transformation approaches and techniques, in: L.G.L.T. Meertens, ed., *Program Specification and Transformation* (North-Holland, Amsterdam, 1987) 165-196.

[12] M.M. Fokkinga, Backtracking and branch-and-bound functionally expressed, in: *Computing Science in the Netherlands, SION Congres 1987* (CWI, Amsterdam, 1987) 207-224; Extended version: Memorandum INF-86-18, University of Twente, Enschede, Netherlands (1986).

[13] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **12** (10) (1969) 567-580 and 583.

[14] G. Malcolm, Homomorphisms and promotability, in: J.L.A. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science **375** (Springer, Berlin, 1989) 335-347.

[15] G. Malcolm, Algebraic Data Types and Program Transformation, Ph.D. Thesis, Groningen University, Netherlands (1990).
[16] L. Meertens, Algorithmics: towards programming as a mathematical activity, in: J.W. de Bakker and J.C. van Vliet, eds., *Proceedings CWI Symposium on Mathematics and Computer Science* (North-Holland, Amsterdam, 1986) 289–334.
[17] L. Meertens, Paramorphisms, Tech. Rept. CS-R9005, CWI, Amsterdam (1990); also in: *Formal Aspects of Computing* (to appear).
[18] H. Partsch, Transformational program development in a particular problem domain, *Sci. Comput. Programming* **7** (1986) 99–241.
[19] D.R. Smith, On the design of generate-and-test algorithms: subspace generators, in: L.G.L.T. Meertens, ed., *Program Specification and Transformation* (North-Holland, Amsterdam, 1987) 207–220.
[20] P. Wadler, How to replace failure by a list of successes, in: J.P. Jouannaud, ed., *Functional Programming Languages and Computer Architecture* (Springer, Berlin, 1985) 113–128.
[21] N. Wirth, Program development by stepwise refinement, *Comm. ACM* **14** (4) (1971) 221–227.
[22] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall Series in Automatic Computation (Prentice-Hall, Englewood Cliffs, NJ, 1976).