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


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:

\[
\text{minus}(\text{exp}(a, 2), \text{exp}(b, 2)) = \text{mult}(\text{plus}(a, b), \text{minus}(a, b))
\]

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 Meertena 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 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 interpret 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 \( x + y + \cdot \cdot \cdot + z \) 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

\[
fg h: x + y = "(fg h) applied to (x + y)"
\]

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 \( + \) and \( \times \) 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 \( \alpha \star \), where \( \alpha \) is the type of the values (elements) contained in the structure; specifically we sometimes write -tree, -list, -bag, or -set instead of \( \star \). Operation \( ^\dagger: \alpha \to \alpha \star \) (written \( ^\dagger x \) or \( ^\dagger \))
forms a singleton structure containing only \( x \). Operation \( ++ : \alpha \times \alpha \rightarrow \alpha \) 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):

- **associative**: \( (x ++ y) ++ z = x ++ (y ++ z) \)
- **commutative**: \( x ++ y = y ++ x \)
- **idempotent**: \( x ++ x = x \)

The constant \( \emptyset : \alpha \) 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 (\( \,<\)\)) and left-reduce or left-insert (\( \langle \, \rangle \)). The first three are special only in that we write them as postfix operations, hence having the highest priority (exactly like primes). Thus

\[
\Theta / \quad f* \quad p\langle = (\Theta /) \quad (f*) \quad (p\langle) \quad \text{and} \quad \Theta / * = (\Theta /)*
\]

In other words, one may consider \( /, * , \langle, \rangle \) as normal binary operations for which the sections (\( \Theta / \)), (\( f* \)), and (\( p\langle \)) 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.

\[\text{map} \quad f* : x \quad \text{is the result of applying} \quad f \quad \text{to every element of} \quad x. \quad \text{Example:}\]

\[
f* : x_1 ++ \cdots ++ x_n = {}^* (f : x_1) ++ \cdots ++ {}^* (f : x_n)
\]

The laws are:

- **(map.0)** \( f* : \emptyset = \emptyset \)
- **(map.1)** \( f* : x = {}^* (f : x) \)
- **(map.2)** \( f* : x ++ y = (f* : x) ++ (f* : y) \)
reduce $\oplus/x$: $x$ is the result of inserting $\oplus$ at every construction node of $x$. Example:

$$\oplus/x/ x_i \ldots \oplus x_n = x_i \oplus \ldots \oplus x_n$$

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 $\omega$ to the domain of $\oplus$ and define $\omega \oplus x = \omega = x \oplus \omega$ for all $x$ (like the introduction of $\infty$ as the unit of the "minimum" operation). The laws are:

(reduce.0) $\oplus/\emptyset = \text{the (possibly fictitious) unit of } \oplus$

(reduce.1) $\oplus/\emptyset = x$

(reduce.2) $\oplus/x = (\oplus/x) \oplus (\oplus/y)$

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

$$\text{odd } x = \hat{\gamma} \triangleleft \hat{\gamma} \triangleleft \hat{\delta} \triangleleft \hat{\delta} \triangleleft \hat{\delta} \triangleleft \hat{\delta} = \hat{\gamma} \triangleleft \hat{\delta}$$

The laws are:

(filter.0) $p \triangleleft \emptyset = \emptyset$

(filter.1) $p \triangleleft \emptyset = \emptyset$ if $p; x$ else $\emptyset$

(filter.2) $p \triangleleft x = (p \triangleleft x) \triangleleft (p \triangleleft y)$

left-reduce $(\oplus \neq e)/x$: $x$ 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:

$$(\oplus \neq e)/x = x_i \ldots \oplus x_n = (\ldots (e \oplus x_1) \oplus \ldots) \oplus x_n$$

The laws are:

(lreduce.0) $(\oplus \neq e) / \emptyset = e$

(lreduce.1) $(\oplus \neq e) / \emptyset = x \oplus e$

(lreduce.2) $(\oplus \neq e) /x = (\oplus \neq ((\oplus \neq e)/x)) \cdot y$

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 $\alpha$, and $f: \alpha \rightarrow \mathbb{N}$, we have

$$+/ f * p \triangleleft s = \sum_{x \text{ in } s | p(x)} f(x)$$

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

$$++/ f * p \triangleleft s = \bigcup \{f(x) | x \text{ in } s \wedge p(x)\}$$
Exercise in transformational programming

In the sequel the term $p \prec\prec f^* p \llhd$ 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 $\equiv$ for the latter and $=$ for the former. By definition, the left-hand side and right-hand side of a definition are equal, so that $\equiv$ may always be replaced by $=$. Conversely this is not true; e.g., for any object $x$ we have $x = x$, but the definition $x \equiv x$ 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.

$$
\begin{align*}
\text{(filter promotion)} & \quad p \llhd ++/ = ++/ p \llhd^* \\
\text{(map promotion)} & \quad f^* ++/ = ++/ f^{**} \\
\text{(reduce promotion)} & \quad \oplus/ ++/ = \oplus/ \oplus/* \quad \text{in particular} \quad ++/ ++/ = ++/ ++/* \\
\text{(map distribution)} & \quad f^* g^* = (f g)^* \\
& \quad ++/ ^ = \text{id} \quad \text{of type } \alpha \rightarrow \alpha \\
& \quad ++/ (^*)^* = \text{id} \quad \text{of type } \alpha \star \rightarrow \alpha \star \\
& \quad f^* ^ = ^ f \\
& \quad p \llhd q \llhd = (p \wedge q) \llhd \\
& \quad (e \odot) \odot/ = (\odot \not\! e) \quad \text{for associative } \odot
\end{align*}
$$

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.**

$$
\begin{align*}
\text{(lreduce-join fusion)} & \quad (\oplus \not\! e) ++/ = (\oplus \not\! e) \\
& \quad \text{where } e \odot x = (\oplus \not\! e) \cdot x \\
\text{(lreduce-map fusion)} & \quad (\not\! f \not\! e) f^* = (\not\! f \not\! e) \\
& \quad \text{where } e \odot x = e \oplus (f \cdot x) \\
\text{(lreduce-filter fusion)} & \quad (\not\! f \not\! e) p \llhd = (\not\! f \not\! e) \\
& \quad \text{where } e \odot x = e \oplus x \text{ if } p \cdot x \text{ else } e
\end{align*}
$$
Proof. By induction on the structure of the argument. For (lreduce-join fusion):

**Basis 1.**

\[
(\oplus e) \rightarrow^\dagger: \emptyset = (\oplus e): \emptyset = e = (\oplus e): \emptyset
\]

**Basis 2.**

\[
(\oplus e) \rightarrow^\dagger: \hat{x} = (\oplus e): x = e \otimes x = (\oplus e): \hat{x}
\]

**Induction step.**

\[
(\oplus e) \rightarrow^\dagger: s \rightarrow^\dagger t
\]

= law (reduce.2) in which \(\oplus:=\rightarrow^\dagger\)

\[
(\oplus e): (\rightarrow^\dagger: s) \rightarrow^\dagger (\rightarrow^\dagger: t)
\]

= law (lreduce.2)

\[
(\oplus ((\oplus e) \rightarrow^\dagger: s)) \rightarrow^\dagger: t
\]

= induction hypothesis

\[
(\ominus ((\ominus e): s)): t
\]

= law (lreduce.2)

\[
(\ominus e): s \rightarrow^\dagger t.
\]

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.**

\[
(\oplus e) \rightarrow^\dagger f^* p^\downarrow = (\oplus e)
\]

where \(e \otimes x := (\oplus e)f: x\) if \(p: x\) else \(e\).

**(3) Corollary.**

\[
(\oplus e) \rightarrow^\dagger f_N^* p_{N-1}^\downarrow \cdots \rightarrow^\dagger f_1^* p_0^\downarrow = (\ominus_0 e)
\]

where

\[
e \otimes_N x := e \oplus x
\]

\[
e \otimes_n x := (\ominus_{n+1} e) f_{n+1}: x\) if \(p_n: x\) else \(e\)

(for \(n = N - 1, \ldots, 0\)).
**Proof.** By induction on \( N - n \) it is easy to prove that
\[
(\oplus \rightarrow e) \rightarrow\! / \ f_N \rightarrow p_{N-1} \rightarrow\! / \ \cdots \rightarrow\! / \ f_1 \rightarrow p_0 \rightarrow\! /
\]
\[
(\otimes \rightarrow e) \rightarrow\! / \ f_N \rightarrow p_{N-1} \rightarrow\! / \ \cdots \rightarrow\! / \ f_1 \rightarrow p_0 \rightarrow\! /
\]
using Corollary (2). \( \square \)

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 \( \rightarrow\! / \) be the construction operation of \( s \). Then, **within the argument of** \( \otimes / \), operation \( \rightarrow\! / \) may be considered to be as rich as \( \otimes \) with respect to associativity, commutativity and idempotency. More precisely,
- \( \otimes \) associative \( \Rightarrow \otimes / : x \rightarrow\! / (y \rightarrow\! / z) = \otimes / : (x \rightarrow\! / y) \rightarrow\! / z \)
- \( \otimes \) commutative \( \Rightarrow \otimes / : x \rightarrow\! / y = \otimes / : y \rightarrow\! / x \)
- \( \otimes \) idempotent \( \Rightarrow \otimes / : x \rightarrow\! / x = \otimes / : x \)

This is easily proved; e.g., for commutativity we argue
\[
\otimes / : x \rightarrow\! / y = (\otimes / : x) \otimes (\otimes / : y) = \\text{commutativity of } \otimes
\]
\[
= (\otimes / : y) \otimes (\otimes / : x)
\]
\[
= \otimes / : y \rightarrow\! / x.
\]
Hence, for associative, commutative and idempotent \( \oplus \) we have, when \( m \) is in \( s \),
\[
\oplus / : s = (m \oplus) \oplus / : m \rightarrow\! / s = (m \oplus) \oplus / : s. \quad \square
\]

The next lemma is formulated for a specific operation \( \uparrow \). We suppose that the domain of \( \uparrow \) is linearly ordered, say by \( \leq \); then \( x \uparrow y \) is the maximum (with respect to \( \leq \)) 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 \( \leq \), and let \( m \) be arbitrary (not necessarily in \( s \)). Then
\[
(m \uparrow) \uparrow / : s = (m \uparrow) \uparrow / (m \leq) \uparrow / s
\]

**Proof.** By induction on the structure of \( s \).

*Case* \( s = \emptyset \). Trivial.

*Case* \( s = x \). Immediate from the meaning of \( \uparrow \) and \( (n \leq) \uparrow / s \).
Case $s = r \mapsto t$. For brevity define $p := (m \leq)$. Then

$$(m^\uparrow, \uparrow/: s)$$

$$= (m^\uparrow) \uparrow/: r \mapsto t$$

$$= m^\uparrow ((\uparrow/: r) \uparrow ((m^\uparrow) \uparrow/: t))$$

associativity, commutativity, and idempotence of $\uparrow$

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

$$= m^\uparrow ((m^\uparrow) \uparrow/: p <\leq: r) \uparrow ((m^\uparrow) \uparrow/: p <\leq: t)$$

$$= m^\uparrow ((\uparrow/: p <\leq: r) \uparrow (\uparrow/: p <\leq: t))$$

$$= (m^\uparrow) \uparrow/: ((p <\leq: r) \mapsto (p <\leq: t))$$

$$= (m^\uparrow) \uparrow/: p <\leq: s$$

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

$$f_0 := \text{some given value}$$

$$f_r := h_n: f_{n-1} \quad \text{for } n > 0$$

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 $h_n: x = n \times x$. 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

(★) \quad 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 (★) that $f_N = g_N \cdot f_N$; hence we may define

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

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

$$f_N$$

= induction hypothesis

$$g_{n+1} \cdot f_{n+1}$$

= definition of $f_{n+1}$

$$g_{n+1} \cdot h_{n+1} \cdot f_n$$
which we want to be equal to \( g_n \cdot f_n \). Hence we may define

\[
g_n := g_{n+1} \cdot h_{n+1}
\]

and by construction the aim (\( \star \)) has been achieved. All of the above may be clarified further by noticing that \( f_n = h_n \cdot h_{n-1} \cdots h_1 \cdot f_0 \) (by repeatedly unfolding the definition of \( f_n \)), \( f_n = h_n \cdots h_1 \cdot f_0 \) and therefore, immediately, \( g_n = h_n \cdots h_{n+1} \).

For the factorial example we find

\[
\begin{align*}
g_N \cdot x &= x \\
g_n \cdot x &= g_{n+1} \cdot (n+1) \cdot x \\
&= N \cdot \cdots \cdot (n+1) \cdot x
\end{align*}
\]

Notice also that \( g_n \) has one parameter more than \( f_n \). This parameter is sometimes called the \textit{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 \textit{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:

\[
\begin{align*}
fct f(n : int) &= \text{if } n = 0 \text{ then } f0 \text{ else } h(n, f(n-1)) \\
fct f(N : int) &= \begin{var}
  x := f0, n := 0;
  \text{while } n < N \text{ do } n := n + 1, h(n + 1, x); \\
  f := x
\end{var}
\end{align*}
\]

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 \textit{Backtracking problems}: the task is to yield any or all of \( p \prec s_N \) where \( s_N \) is inductively defined
by

\[ s_0 := \text{some given structure} \]

\[ s_n := ++/ f_n*: s_{n-1} \quad \text{for } n > 0 \]

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 \textit{legality constraint}. Thus we have the typing:

\[ s_n : \alpha\star, f_n : \alpha \rightarrow \alpha\star \text{ and } p : \alpha \rightarrow \mathbb{B}, \]

for some \( \alpha \). Mostly \( \alpha \) is \( \beta\)-bag or \( \beta\)-set, and then in imperative implementations the members of \( s_n \) are represented by an \textit{array of} \( \beta \).

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

\[ s_0 := \emptyset \quad : \mathbb{N}\text{-set} \]

\[ f_n : x := \hat{x} \uplus (x \uplus \hat{n}) : \mathbb{N}\text{-set} \rightarrow \mathbb{N}\text{-set-set} \]

\[ p := (W\geq) +/ w* : \mathbb{N}\text{-set} \rightarrow \mathbb{B} \]

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 \( p \subseteq s_N \), 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 \( s_n \):

\[ s_N \]

\[ = ++/ f_N* : s_{N-1} \]

\[ = ++/ f_N* \leftrightarrow / f_{N-1}* : s_{N-2} \]

\[ \vdots \]

\[ = \quad \text{by equation (7)} \]

\[ ++/ f_N* \leftrightarrow / f_{N-1}* \cdots \leftrightarrow / f_1*: s_0 \]

Next, we apply map promotion \((f_* \leftrightarrow / = \leftrightarrow / f_***)\) repeatedly, and obtain

\[ s_N \]

\[ = \quad \text{by equation (7)} \]

\[ ++/ f_N* \leftrightarrow / f_{N-1}* \cdots \leftrightarrow / f_1*: s_0 \]

\[ = \quad \text{by (map promotion) on the subterm } f_N* \leftrightarrow / \]

\[ ++/ ++/ f_N** f_{N-1}* \cdots ++/ f_1* : s_0 \]

\[ \vdots \]

\[ = \quad \text{by (map promotion) on the subterm } f_1* \]

\[ ++/ f_1* : s_0 \]

\[ = \quad \text{by (map promotion) on the subterm } f_1* \]

\[ (+/)^N f_N* f_{N-1}^* \cdots f_1* : s_0 \]
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)

\[
(9) \quad s_N = (+++)^N (\ldots (f_N^* f_{N-1}^*)^* \ldots f_1^*)^*: s_0
\]

Consider once more equation (7):

\[
(10) \quad s_N = r_n^(: s_n
\]

\[
(11) \quad r_n = +++/ f_N^* \ldots +++/ f_{n+1}^*: s_0
\]

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

\[
(12) \quad s_N = +++/ t_n^*: s_n
\]

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 \( t_N := \). Next, proceeding by induction and therefore assuming that \( s_N = +++/ t_{n+1}^*: s_{n+1} \), we aim at \( t_n \) such that

\[
(++)/ t_{n+1}^*; s_{n+1}/ (\text{map promotion})
\]

\[
(++)/ (++)/ t_{n+1}^*; s_n
\]

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:

\[ t_N \coloneqq \wedge \]

\[ t_n \coloneqq t_{n+1} \star f_{n+1}. \tag{13} \]

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 \( p < s_N \) 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 \( p < s_N \) 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 \( p < \). In other words, one should try to promote (parts of) the filter \( p < \) 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.,

\[ \emptyset = p < \leftrightarrow t_n \star (\neg p_n) < \colon s_n \]

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

\[ \begin{align*}
\emptyset & = p < \leftrightarrow t_n \star (\neg p_n) < \colon s_n \\
& = p < \leftrightarrow f_n \star \cdots \star f_n \star \cdots \star f_1 \star \colon s_0 \\
& = \text{proved in detail in the appendix, Theorem (27)}
\end{align*} \]

\[ p < p_N < \leftrightarrow f_{N-n} \star \cdots \star p_n < \leftrightarrow f_n \star \cdots \star p_1 < \leftrightarrow f_1 \star \colon s_0. \tag{14} \]
Now notice that

\[ p_n < \llcorner \triangleright/ f_n \]  

\[ = \text{filter promotion} \]  

\[ \triangleright/ p_n < \llcorner f_n \]  

\[ = \text{map distribution} \]  

\[ \triangleright/ (p_n < f_n) \]  

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

\[ (15) \quad p < \llcorner; s'_n = p < \llcorner \triangleright/ f'_n * \cdots \triangleright/ f'_1 *; s_0. \]

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 \( p < \llcorner; s_N \); cf. (6) and (13):

\[ s'_0 := p_0 < \llcorner; s_0 \]

\[ s'_n := \triangleright/ f'_n *; s'_{n-1} = \triangleright/ (p_n \llcorner f_n) *; s'_{n-1} = p_n < \llcorner \triangleright/ f_n *; s'_{n-1} \]

\[ p < \llcorner; s_N = p < \llcorner; s'_N \]

and

\[ t'_n := \]  

\[ (17) \quad t'_n := \triangleright/ t'_{n+1} * f'_{n+1} = \triangleright/ t'_{n+1} * p_{n+1} \llcorner f_{n+1} \]

\[ p < \llcorner; s_N = p < \llcorner \triangleright/ t'_0 *; s_0. \]

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 \( s_n = \triangleright/ f_n *; s_{n-1} \), but it is actually used in a filter on \( \triangleright/ f_n *; s'_{n-1} \) 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

\[ p_n \llcorner \triangleright/ f_n *; s_{n-1} = q_n \llcorner \triangleright/ f_n *; p_{n-1} \llcorner s_{n-1}. \]

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 $s_n^r$, $t_n^r$ and $f_n^r$. It turns out that

$$f_n^r := p_n^r \otimes f_n^r = p_n^r \otimes (p_n \otimes f_n) = (p_n^r \land p_n) \otimes f_n$$

and therefore we conclude that repeated filter promotions may be done at once, taking $p_n^r \land p_n$ as the filter on $s_n^r$. (Here, $p \land q$ is a notation of the predicate $r$ defined by $r(x) := (p(x) \land q(x))$. 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 $s_0^r$, followed by the full computation of $s_1^r$, and so on. Another method is as follows. The evaluator tries to output the requested result and therefore computes $s_N^r$ only as far as is needed—and this in turn may trigger the computation of $s_{N-1}^r$ (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 $f_n^r$) an already found partial solution (elements of $s_{N-1}^r$) and checks whether the extensions pass the filter $p_n^r$. 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 $p \otimes s_N^r$. Now we consider the task of computing the optimal element of $p \otimes s_N^r$. To this end we assume that there exists a linear order $\leq$ on the element of $s_N^r$ and that $\uparrow/ p \otimes s_N^r$ is requested; operation $\uparrow$ is defined by

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

Without further knowledge we cannot, of course, give a more efficient algorithm than the specification $\uparrow/ p \otimes s_N^r$ itself. So let us assume that we know something more. First of all, as in the previous section there may exist predicates $p_n^r$ that are a necessary condition for elements of $s_n$ in order that their contribution to $s_N^r$ 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 $f_n^r$ are simply replaced by $f_n^r = p_n^r \otimes f_n$. 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 $s_N^r$ may dominate $m$; here $m$ is some element that plays the role of "the currently found maximum of $s_N^r$" and informally $p_{n,m}$ says whether an element of $s_n$ "looks promising" with
Exercise in transformational programming

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

\[
\uparrow / \; p \triangleleft \downarrow s_N
\]

= Lemma (4)

\[
(m \uparrow) \uparrow / \; p \triangleleft \downarrow s_N
\]

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

\[
(m \uparrow) \uparrow / \; (m \leq) \triangleleft p \triangleleft \downarrow s_N
\]

= "filter promotion" as in Section 4

\[
(m \uparrow) \uparrow / \; (m \leq) \triangleleft p \triangleleft p_{N,m} \triangleleft \rightarrow \cdots \rightarrow p_{1,m} \triangleleft \rightarrow f_1 * p_{0,m} \triangleleft \downarrow s_0.
\]

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:

\[
\begin{align*}
\text{var} & \quad m := \text{some (fictitious) element of } p \triangleleft \downarrow s_N; \\
\text{fct} & \quad \text{test}(x) := \text{if } m \leq x \text{ then } m := x; \text{true else false fi}; \\
\text{result-is} & \quad (m \uparrow) \uparrow / \; \text{test} < p \triangleleft \downarrow p_{N,m} \triangleleft \rightarrow \rightarrow p_{1,m} \triangleleft \rightarrow f_1 * p_{0,m} \triangleleft \downarrow s_0.
\end{align*}
\]

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:

\[
(18) \quad \emptyset = (m \leq) \triangleleft p \triangleleft \rightarrow t_n * (\neg p_{n,m}) \triangleleft \downarrow s_n
\]

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

\[
(19) \quad (m \leq) \triangleleft \rightarrow t_n *: s = (m \leq) \triangleleft \rightarrow t_n * p_{n,m} \triangleleft \downarrow s \text{ for } s \leq s_n
\]

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 \( \hat{x} \subseteq s_n \):

\[
(20) \quad (m \uparrow) \uparrow / \; p \triangleleft \downarrow t_n; x \text{ if } p_{n,m}; x \text{ else } m
\]
\[
(m'\uparrow) \uparrow/ p \ll x
\]

\[= \text{equation (20), noting that } x \subseteq s \subseteq s_n
\]

\[= (m'\uparrow) \uparrow/ p \ll x \text{ if } p_{N,m'}; x \text{ else } m'
\]

\[= m' \uparrow x \text{ if } (p \land p_{N,m'}) \land x \text{ else } m'
\]

\[= m' \oplus_N x
\]

\[= (\oplus_N \not\triangleright m) \triangleright s.
\]

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

\[= (m\uparrow) \uparrow/ p \ll t_{n-1}; s
\]

\[= (m\uparrow) \uparrow/ p \ll t_{n}; s
\]

\[= \text{induction hypothesis for } n
\]

\[= (\oplus_n \not\triangleright m) \triangleright f_n; s
\]

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

\[= (\oplus \not\triangleright m) \triangleright s
\]

\[= \text{with } m' \otimes x := (\oplus_n \not\triangleright m') \triangleright f_n; x
\]

\[= \text{induction hypothesis}
\]

\[= (m'\uparrow) \uparrow/ p \ll t_{n}; x
\]

\[= (m'\uparrow) \uparrow/ p \ll t_{n-1}; x
\]

\[= \text{equation (20), noting that } x \subseteq s \subseteq s_{n-1}
\]

\[= (m'\uparrow) \uparrow/ p \ll t_{n-1}; x \text{ if } p_{n-1,m'}; x \text{ else } m'
\]

\[= \text{back again}
\]

\[= (\oplus_n \not\triangleright m') \triangleright f_n; x \text{ if } p_{n-1,m'}; x \text{ else } m'
\]

\[= m' \otimes_{n-1} x
\]

\[= (\oplus_{n-1} \not\triangleright m) \triangleright s
\]

This completes the proof. \(\square\)

As an immediate corollary we have that, when } m \text{ is the smallest with respect to } \leq \text{ or when } m \text{ is in } p \ll s_N,

\[p \ll s_N = (m\uparrow) \uparrow/ p \ll s_N = (\oplus_0 \not\triangleright m) \triangleright s_0
\]

Algorithm \(\oplus_0 \not\triangleright m\) 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 \( p_o \circ s_0 = \emptyset \) (a singleton).

6.1. Implementations 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.

```plaintext
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 \( \mathcal{F} \mathcal{S} s \) is transcribed as

```plaintext
for each return of \( \tilde{s} \) do
    for each return of \( \tilde{f} \) 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 \( \text{wgt} \) that equals the aggregate weight of \( x \). For the representation of elements from \( s_n \) only \( a[1], \ldots, a[n] \) and \( \text{wgt} \) are significant; \( a[n+1], \ldots, a[N] \) are meaningless. (Hence, in the context of \( n = 0 \) the initialization \( x := x_0 \) is implemented by \text{skip}.) The problem dependent definitions now read as follows.

```haskell
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 \( \tilde{f}_n \) is to be performed for each result (element) of \( s'_n \). The results of \( s'_{n-1} \), however, are stored one after the other in \text{var} \( x \). Nevertheless this can be done satisfactorily. The idea is to pass \( \tilde{f}'_n \) as a "continuation parameter" to the procedure that implements \( s'_n \). 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), \( s_N = r'_n \cdot s'_n \), 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

\[
s''_n \cdot r'_n = s_N.
\]

From this aim one derives quite easily the definition

\[
s''_0 \cdot c := c \cdot s_0 = c \cdot \hat{x}_0
\]

\[
s''_n \cdot c := s''_{n-1} \cdot (c \leftrightarrow f'_n \cdot)
\]

\[
p \lhd s_N = p \lhd s''_N ; r'_N = p \lhd s''_N ; id = s''_N ; p \lhd.
\]
Exercise in transformational programming 41

(The very last equation is justified by an inductive proof of $f \ s_n^\tau; \ c = s_n^\tau; (f \ c)$ for all $f$ and $c$.) Similarly we assume that also $c \leftrightarrow f_n^\tau$ can be turned inside-out: that is there exists some $f_n^\tau$ for which $f_n^\tau; c = c \leftrightarrow f_n^\tau$. The imperative implementation now suggests itself:

\begin{verbatim}
var x;
proc f''(n : int; proc c) =
    {yields in succession in var x each element of (f'', c): x};
proc s''(n : int; proc c) =
    {yields in succession in var x each element of s'', c}
    if n = 0 then x := x0; c else s''(n - 1, proc: f''(n, c));
...

s''(N, proc: if p() then print)
...
\end{verbatim}

Specifically for PLS the problem dependent definitions read:

\begin{verbatim}
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 $f_n; x$.

$$x, p(), p(n: int), f(n: int) :- as \text{ in Section 6.1}$$

\begin{verbatim}
proc t'(n : int) =
    {stores each element of $t'_N; x$ in succession in var x;
     or rather, prints the elements of $p\triangleright t'_N; x$ 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)
...
\end{verbatim}
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:

$$\text{proc } t'(n:\text{int}) =$$

$$\text{if } n = N$$

$$\text{then print}$$

$$\text{else begin } x.a[n] := \text{true}; \text{ wgt := wgt + w(n);}$$

$$\text{if } p(n+1) \text{ then } t'(n+1);$$

$$\text{wgt := wgt - w(n);}$$

$$x.a[n] := \text{false}; t'(n+1)$$

$$\text{end;}$$

6.4. Implementation of (22)

The implementation of

$$(\oplus \neq m): \hat{x}_1 + \cdots + \hat{x}_n = (\cdots (m \oplus x_1) \neq \cdots) \oplus x_n$$

suggests itself: an iteration of $\oplus$ over $x_1, \ldots, x_n$ with one global variable \text{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 $\oplus_n$.

$$x, p(), f'(n):= \text{as before}$$

$$\text{fct } p(n:\text{int}, m:\text{elt}): \text{bool} = \{\text{yields } p_{n,m}: x\};$$

$$\text{proc } op(n:\text{int}) =$$

$$\{\text{yields the result of } (\oplus \neq m): x \text{ in } \text{var } m\}$$

$$\text{if } n = N$$

$$\text{then if } p() \text{ and } p(N, m) \text{ then } m := m \uparrow x \text{ else } m := m$$

$$\text{else if } p(n, m)$$

$$\text{then for each return of } f'(n+1) \text{ do } op(n+1)$$

$$\text{else } m := m;$$

$$\ldots$$

$$x := x_0; m := \text{some (fictitious) value such that } (m \uparrow) p \triangleq s_N = p \triangleq s_N;$$

$$op(0); \text{ write}(m)$$

Specifically for POS we instantiate the above to:

$$\text{var } x, m : \text{record } a: \text{array [1..N] of bool; \ wgt : real \ end;}$$

$$\text{fct } p1(n:\text{int}): \text{bool} = x.wgt \leq W;$$

$$\text{fct } p2(n:\text{int}, m:\ldots) = x.wgt + \sum_{i=n+1}^N w(i) \geq m.wgt;$$

$$\text{proc } op(n:\text{int}) =$$

$$\text{if } n = N$$

$$\text{then if } \{p2(N, m) \text{ and } m.wgt \leq x.wgt \text{ then } m := x \text{ else skip}$$

$$\text{else if } p2(n, m) \text{ then}$$
Exercise in transformational programming

begin x.a[n+1] := true; x.wgt := x.wgt + w(n+1);
if pl(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 /, *, <, ≠, ++ 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-lreduce/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 [203]. 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: x to s_N may satisfy p" by

\[ \emptyset = p \triangleleft ++/ t_n* (\neg p_n) \triangleleft: s_n. \]

Note that if we had chosen the formalization as \( \emptyset = p \triangleleft ++/ t_n* (\neg p_n) \triangleleft: s \) 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. \( s' \triangleleft t' \subseteq s \triangleleft t \) 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. \( s \subseteq s \triangleleft t \) and \( t \subseteq s \triangleleft t \),
4. \( s \subseteq t \subseteq s \) implies \( s = t \),
5. \( s \subseteq t \) implies \( p \triangleleft: s \subseteq p \triangleleft: t \),
6. \( p \triangleleft: s \subseteq s \),
7. \( s \subseteq t \) implies \( ++/ f*: s \subseteq ++/ f*: t \).
Exercise in transformational programming

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

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

This completes the proof. □

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

$$
p \mid \map{++/}{t_n*}{s} = \map{++/}{t_n*}{p_n \mid s}
$$

Proof. By induction on the structure of $s$.

Case $s = \emptyset$. Trivial.

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

$$
\begin{align*}
& \quad p \mid \map{++/}{t_n*}{\hat{x}} \\
& = \quad p \mid \map{++/}{t_n*}{(p_n \lor \neg p_n)\mid \hat{x}} \\
& = \quad p \mid \map{++/}{t_n*}{((p \mid \hat{x}) ++ (\neg p_n \mid \hat{x}))} \\
& = \quad (p \mid \map{++/}{t_n*}{p_n \mid \hat{x}}) ++ (p \mid \map{++/}{t_n*}{\neg p_n \mid \hat{x}}) \\
& = \quad \text{assumption (23)} \\
& \quad p \mid \map{++/}{t_n*}{p_n \mid \hat{x}}. \\
\end{align*}
$$

Case $s = r ++ t$. Now

left-hand side

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

$$
\begin{align*}
 s_0' &:= p_0 \sqsubseteq s_0 \\
 s_n' &:= p_n \sqsubseteq \langle n/ f_n \rangle. s_{n-1}' = p_n \sqsubseteq \langle n/ f_n \rangle \cdot \cdots \cdot p_1 \sqsubseteq \langle n/ f_1 \rangle p_0 \sqsubseteq s_0
\end{align*}
$$

We show by induction on $n$ that

$$
 p \sqsubseteq s_n = p \sqsubseteq \langle n/ t_n \rangle: s_n' \quad \text{and} \quad s_n' \subseteq s_n.
$$

Basis.

$$
\begin{align*}
 p \sqsubseteq s_N &= p \sqsubseteq \langle n/ t_0 \rangle: s_0 = \{\text{Lemma (26)}\} p \sqsubseteq \langle n/ t_0 \rangle: s_0' \\
 s_0' &= p_0 \sqsubseteq s_0 \subseteq \{\text{Lemma (25.6)}\} s_0.
\end{align*}
$$

Induction step. For $p \sqsubseteq s_N$ we argue:

$$
\begin{align*}
 p \sqsubseteq s_N &
= \quad \text{induction hypothesis} \\
 p \sqsubseteq \langle n/ t_n \rangle: s_n' \\
= \quad \text{definition of } t_n \\
 p \sqsubseteq \langle n/ (\langle n/ t_{n+1} \rangle f_n) \rangle: s_n' \\
= \quad \text{(map distribution), (map promotion)} \\
 p \sqsubseteq \langle n/ t_{n+1} \rangle \langle n/ f_n \rangle: s_n' \\
= \quad \text{induction hypothesis gives } s_n' \subseteq s_n, \quad \text{Lemma (25.7) gives } \langle n/ f_n \rangle: s_n' \subseteq \langle n/ f_n \rangle: s_n = s_{n+1}; \quad \text{apply Lemma (26)} \\
 p \sqsubseteq \langle n/ t_{n+1} \rangle p_{n+1} \sqsubseteq \langle n/ f_n \rangle: s_n' \\
= \quad \text{definition of } s_{n+1}' \\
 p \sqsubseteq \langle n/ t_{n+1} \rangle: s_{n+1}'.
\end{align*}
$$
Exercise in transformational programming

And for $s_{n+1}'$ we calculate:

$$s_{n+1}' = \text{definition}$$

$$p_{n+1} \triangleq ++/ f_{n+1}*: s_n'$$

$$\subseteq \text{Lemma (25.7), induction hypothesis}$$

$$p_{n+1} \triangleq ++/ f_n*: s_n$$

$$\subseteq \text{Lemma (25.6)}$$

$$++/ f_n*: s_n$$

$$= \text{definition}$$

$$s_{n+1}.$$

This completes the proof. ∎

References


