Technische Hogeschool Twente

MEMORANDUM INF-86-18

BACKTRACKING AND BRANCH-AND-BOUND FUNCTIONALLY EXPRESSED.

M.M. Fokkinga.

Juni 1986

Twente University of Technology
Department of Informatics
P.O. Box 217
7500 AE Enschede
The Netherlands

Onderafdeling der Informatica

Backtracking and Branch-and-Bound functionally expressed

ABSTRACT

We develop a program scheme in a functional language with lazy evaluation for the kind of algorithms which are usually called Backtracking algorithms and Branch-and-Bound algorithms. Several examples are given, amongst others the Eight Queens Problem and the Problem of the Optimal Selection.

CONTENTS

1. Introduction

"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 already found approximate solution in all possible ways. Whenever it turns out that such an extension fails, one "backtracks" to the last point of choice where there are still alternatives available. For most problems it is of utmost importance to predict already very early that an approximate solution can not 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, be it that every time a solution is encountered the search space can be reduced further: from then onwards one need not try to extend approximate solutions if it is sure that their extensions can not be as good as the currently optimal one. In this case one speaks 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.

We set out to develop, in a functional language with lazy evaluation, program schemes for typical Backtrack and Branch-and-Bound problems. It will turn out that the Backtracking behaviour, which may be attributed to the program evaluation, is got almost for free thanks to the "Lazyness" of the evaluation. The main technique used in the program development is "filter promotion"; this technique has a much wider application than just the Backtracking area. In order to obtain the Branch-and-Bound behaviour we transform a Backtrack program in a way that we have not seen in the functional programming literature to date.

The remainder of this paper is organized as follows. Our motivation and our notation for sets and bags, besides lists, is given in the next section. Then,

in Section 3, we treat the Eight Queens Problem as a particular instance of Backtracking. In Section 4 Backtracking is treated more generally and schematically. In Section 5 Branch-and-Bound enters the picture --in a general and schematic way--. A particular example of Branch-and-Bound is given in Section 6: The Problem of the Optimal Selection. In Appendices A through C several other examples are given.

I thank Therese ter Heide-Noll for careful typing of the manuscript.

2. Notation

Except for the notation introduced below we use a Miranda [Turner 1985] -like syntax; this is so close to conventional mathematical notation that it needs no explanation. By way of experiment we shall try to avoid overspecification by using sets, bags and lists where appropriate. (Bags are lists in which the ordering is irrelevant; one might also say that bags are sets in which the members have multiple occurrences.) Our experience, obtained by the present paper only, is that a suitable use of sets and bags indeed simplifies some refinement steps in the program development, but requires some additional thinking too; (we sometimes fell in the pitfall of chosing sets where bags were necessary). From a theoretical point of view there are also problems concerning the nondeterminacy which arises; we shall return to this point below.

Our notation is heavily influenced by the work of Meertens [Meertens 1985]. Actually, we are convinced that his notation is far superior to ours!! The only advantage of our proposal is that it is closer to what is present in current functional languages.

We use the symbols { } and ⦃ ⦄ and [ ] for sets, bags and lists respectively. The list operations :, ++, # and -- are extended to bags and sets in the canonical way. Thus e.g.

a:{a,b,c}	a:⦃a,b,c⦄	a:[a,b,c]
={a}++{a,b,c}	=⦃a⦄++⦃a,b,c,⦄	=[a]++[a,b,c]
={a,a,b,c}	=⦃a,a,b,c⦄	=[a,a,b,c]
={a,b,c}	=⦃a,b,a,c⦄	≠[a,b,a,c]
={b,c,a}	≠⦃a,b,c⦄	≠[a,b,c]

We shall allow the colon and set/bag/list -brackets in formal parameters. A set S matches (x:X) if S is nonempty, say with member s; then x may stand for s and X denotes S--{s}. So the following function counts the number of elements in a set.

```
count {} = 0
count (x:X) = 1 + count X.
```

For practical, notational convenience it seems very important that functions are polymorphic in the sense that they may accept both sets and bags and lists. To this end, we use the brackets { }, possibly subscripted, to denote polymorphically either a set or a bag or a list, (but within one definition the same brackets denote the same thing.) Hence, we may define

```
count { } = 0
count (x:X) = 1 + count X
```

Then (count X) = #X for any set, bag or list X. More generally we use { } whenever the context determines uniquelly what kind of bracket is meant; in particular this holds for set-, bag- and list-abstractions.

On semantical grounds we require that all generator domains of a set-abstraction are sets, and similarly for bag-abstractions and list-abstractions. It is however allowed to write a list where a bag is required, and to write a bag where a set is required. The implicitly applied conversion is obvious. The conversion the other way around must be written explicitly by a prefix superscript `. Thus for a set S, `S is a bag (uniquely determined) and ``S is some list (some enumeration without duplicates). So we may define

```
set-to-bag {} = {|}
set-to-bag (x:X) = x: set-to-bag X

bag-to-list {|} = []
bag-to-list (x:X) = x: bag-to-list X
```

Then for a set S and bag B, `S = set-to-bag (S) and `B = (bag-to-list B) and `B = ``S.

Finally we need to say some words about the set and bag abstractions. In order that these constructs truly correspond to the mathematical notions, it is required that the various generators in the abstraction each get a fair chance. So

```
{(i,j) | i <- [1..], j <- [i..]}
```

really denotes the set in which each pair (i,j) with $i \geq 1$ and $j \geq i$ occurs. If we replace the comma by a semicolon, then this is meant as a directive to the evaluator to give precedence to the second generator; i.e. the generator processes are still fairly interleaved, but e.g. after each value generated for i, the number of values to be generated for j may be doubled. (And if the domain of j is finite, the evaluator may generate all values of j after each one for i.) By this replacement we may possibly improve the time and/or space efficiency of the computation. We shall not use a comma between the generators of a list.

Nondeterminacy enters the picture by having a set split into some member and the remainder, when used as argument for a parameter (x:X). Note, however, that we shall only make a very modest use of it: mostly we avoid overspecification by using determinate sets rather than nondeterminate elements. (Presumably one can avoid nondeterminacy altogether up to the very last action of the computation, where some member of a set is taken nondeterministically.) Notice for instance that the functions count and set-to-bag are deterministic! We shall also allow guards to be non-exclusive.

In view of the nondeterminacy, semantic equality of two expressions now means that both expressions allow for the same input/output behaviour; the possible outcomes of both expressions, including nontermination, are the same. In order that nondeterminacy is practically manageable (in proving semantic equalities), it seems necessary to require "environmental transparancy", i.e. in each defining clause and each expression different occurrences of the same variable denote the same (nondeterminate) value, (and not different values). In the context of lazy evaluation this can be achieved very easily by the sharing feature of lazyness, cfr [Fokkinga 1985].

We have no formal proof system for semantic equivalence which deals with nondeterminacy. Nevertheless we have the strong feeling that the claimed semantic equalities really hold true. (At least one thing is definitely true: the left and right hand side of definitions are semantically equal, so that definitional equality implies semantic equality.)

3. The Eight Queens Problem

In this section we treat the typical Backtracking problem, well-known as the Eight Queens Problem. In the next section we shall generalize what we do here

for this concrete example.

The problem is to place 8 queens on a 8x8 chess board in such a way that no queen is in check of another. As you know a queen covers the squares of the row, the column and both diagonals in which it is placed.

We shall represent the placement of one queen by a pair (i,j) where i is the row number and j the column number of the queen's square. The placement of several queens is represented by a bag of the individual placements, (thus allowing that two queens occupy the same square). For these representations it is easy to formulate the tests whether one queen (i,j) is in check of another (i',j'), and whether a placement p is legal:

```
check (i,j) (i',j') = i=i' \/ j=j' \/ i+j=i'+j' \/ i-j=i'-j'
legal p = all {~ check q q' | q,p'<<- p; q'<-p'}
```

Notes. 1. The generator x,Y <<- X generates in x the elements of X (with the right multiplicity and order if X is a bag or list), and in Y the remainder, not ranged over by x yet. So, if at any time (x_1,Y_1), ..., (x_n,Y_n) are the values generated so far, then $X = x_1:x_2:\ldots:x_n:Y_n$. It easy to define a function Heads-and-Tails so that x,Y <<- X can be written (x,Y) <- (Heads-and-Tails X).
2. We could have defined legal more abstractly by
`legal p = all {~ check q q' | q <- p, q' <- p--{q}}`.
The given definition is equivalent to this one thanks to the symmetry of check.

We shall now develop a program that yields a set LP of all legal placements of 8 queens. In case only one solution is requested, one may simply write (hd LP); thanks to Lazy Evaluation the set LP is not computed beyond its very "first" element!

We start with a definition of LP which is almost the problem specification itself:

```
LP = {p | p <- P 8; legal p}
P 0 = {{}}
P n = {q:p | q <- Q, p <- P (n-1)}, n>0
Q = {(i,j) | i <- {1..8}, j <- {1..8}}
```

Notes. 1. (P n) yields the set of all placements of n queens on the board.

2. We write a comma in between generators of an abstraction if the order is irrelevant; we may later choose a particular order to control the computation and improve the efficiency.

This program for LP, although being correct, is practically worthless: set (P 8) has 8^{64} members, all of which have to be subject to the legality test. Even if only one member of LP is asked for, and hence only a fraction of (P 8) has to be considered, the running time of the algorithm is far too large to be practical: moderate fractions of 8^{64} are still very large indeed.

So we need to improve the efficiency. We shall try to replace the sets involved in the construction of LP by much smaller sets, by omitting members which cannot become legal placements of 8 queens. There are two mutually independent possibilities to do so: one is to avoid placing two queens on the same ~~square~~ row and the other is to "promote" (part of) the legality test into the definition of P. We shall first treat the latter and then the former.

We observe that any extension of a partial placement p <- (P n) will be illegal if p itself is already illegal. So we aim for smaller sets (P' n) equal to {p | p <- P n; legal p}, (to be defined without referring to (P n), of course). We can easily construct (P' n) from (P'(n-1)); but knowing that the members p <- (P'(n-1)) are already legal, it suffices to test only whether the new n-th queen is acceptable with respect to p. Thus we are led to define:

```
LP = {p | p <- P' 8; legal p}
   = (P' 8)
P' 0 = {[]}
P' n = {q:p | q <- Q, p <- P' (n-1); acc q p}, 0<n
Q    = {(i,j) | i <- {1..8}, j <- {1..8}}
acc q p = all {~ check q q' | q'<- p}
```

Note. One may verify that (P' n)={p | p <- P n; legal p} indeed. Hence the outermost legality test in LP is superfluous.

Next we reduce sets (P' n) once more to smaller sets (P" n), thereby increasing the efficiency of the algorithm again. We observe that: (i) the order of queens within a placement p is irrelevant (actually p is a bag!), and (ii) for any p <- LP it is true that p only contains queens with different row numbers. So we may constrain the choice of q in the definition of (P' n) to a particular row uniquely determined by n, say the n-th row. This leads to:

```
LP = P" 8
P" 0 = {ξ}
P" n = {(n,j):p | j <- {1..8}, p <- P" (n-1); acc' (n,j) p}, 0<n
acc' q p = all {~check' q q' | q'<- p }
check' (i,j) (i',j') = j=j' \/ i+j=i'+j' \/ i-j=i'-j'
```

Note. The reduction of (P' n) to (P" n) also allows to increase the efficiency of the acceptability test: by construction no two queens in a placement are in the same row, hence the test i=i' is omitted.

At this point there doesn't seem to be another obvious possibility for further reduction of the sets involved. So let us now argue (rather vaguely, we admit) that the program thus obtained really evokes a "backtracking computation". To this end we notice that whenever during the computation a member of some (P" n) is demanded, the possibilities of j <- {1..8} and p <- P"(n-1) are computed (in some unspecified order) until (acc' (n,j) p) succeeds. The placement (n,j):p is then delivered and future demands for further members of (P" n) will only consider the remaining possibilities for j and p. This systematic, hierarchically organized search is really backtracking.

The program development is however not finished. It turns out, as you might expect, that in general the generation of a next member of (P" n) involves much more work (namely generation of part (P" (n-1)) and hence also part of (P" (n-2)), ..., (P" 0)) than the generation of a next member of {1..8}. Assuming that the pairs (j,p) satisfying (acc' (n,j) p) are distributed quite randomly over the cartesian product {1..8}x(P" n), it is more efficient to search that cartesian product while economizing on the number of generations of elements p, that is giving the generation of elements j some (or complete) precedence over the generation of p's. Thus we control the computation further by specifying a preferred order for the generators in the definition of (P" n):

```
P" n = {(n,j):p | p <- P" (n-1); j <- {1..8}; acc' (n,j) p}
```

Notes 1. If the evaluation of a composite generator x <- X, y <- Y proceeds by dove-tailing the sequences of computation steps of x <- X and y <- Y, then the evaluation economizes automatically on the more expensive generator! (Stef Joosten has brought this to my attention.) Nevertheless, a specific order is needed if one wants to implement sets by deterministic lists and then the above reasoning certainly applies. Also, an explicit preferred order might improve

the space requirements of the computation, (as Gerrit van der Hoeven has remarked). We shall not pursue this topic here.
2. The semicolon between generators of a set abstraction has no semantic meaning. It is to be considered as a directive to the evaluator. Within list abstractions, of course, the semicolon is semantically significant.

We have completed the development of a program for LP; we conclude this section by presenting a (quite different?) program for essentially the same algorithm. This version has advantages for the development of Branch-and-Bound algorithms and for the development of imperative programs. The main idea might perhaps be described as "doing the recursion bottom-up rather than top-down" or as "performing parameter accumulation", (similarly as with the transformation of the canonical definition of the Fibonacci function to the tail recursive one). We shall define a set (P_1 p n) of all extensions of the partial placement p with queens in rows n, n+1,...,8. Formally, the relation between P" and P_1 is given by: for each n in 0..8

(P" 8) = {p' | p <- P" n; p' <- P_1 p (n+1)}

The definition reads as follows:

P_1 p 9 = {p}
P_1 p n = {p' | j <- {1..8}; acc' (n,j) p; p'<-P_1((n,j):p)(n+1)}, n≤8

The correctness is easily proved by induction on 8-n, and actually the correctness proof and the development of the definiton could have gone hand-in-hand. Now we may define LP also by LP = P_1[]1.

We shall call the form of the definition of P_1 "elementwise iterative". The usefullness of this version will become clear in the next section, and in Appendix D where imperative implementations of P" and P_1 are given.

4. Backtracking treated schematically

In this section we redo the program development of the previous section in more general terms. Thus we shall obtain a program scheme for Backtracking problems. We shall not strive for the utmost generalization, for that would give notational and presentational problems. Rather it is the development of the program scheme that serves as an outline to follow when trying to solve a particular problem. In the next section the treatment is extended to Branch-

and-Bound.

In general terms formulated, the problem is to yield some or all members of a certain set S. Characteristic of the Backtracking problems is that a superset of S can be approximated inductively by sets S_0, S_1, S_2, ..., S_N, with $S_N \supseteq S$, where

- S_0 is explicitly known
- for $0<n\leq N$, $S_n = \{cons_n\ x\ s \mid x \leftarrow C_n,\ s \leftarrow S_{n-1}\}$

Here C_n denotes the set of possible choices for x and $cons_n$ is some constructor (possibly but mostly not dependent on n). For the Eight Queens Problems we had S_n = (P n) = "the placements of n queens on the 8 x 8 board", C_n = Q = {1..8} x {1..8} and $cons_n$ = "adding an element to a bag" (denoted by :). Notice that one may even generalize the construction of the sets S_n and of their members; for example one could have $(cons_n\ x\ s_1 \ldots s_k)$ rather than $(cons_n\ x\ s)$ and more importantly, sometimes some conglomeration of parameters, ~~say parm~~, plays the role of n (and some function ~~decr~~ plays the role of substraction), (see Section 6 and Appendices A,B,C).

So we may now define

$S = \{s \mid s \leftarrow S_N;\ legal\ s\}$

$S_0 = \ldots$

$S_n = \{cons_n\ x\ s \mid x \leftarrow C_n,\ s \leftarrow S_{n-1}\},\ 0<n$

The legality test should succeed only and precisely for those members of the superset S_N which indeed belong to S. The entities legal, $cons_n$, S_0, C_n and N depend on the particular problem.

Given the above program one should now seek methods to reduce the sets involved so that the efficiency of the algorithm increases accordingly. This can make a difference of several orders of magnitude! One of the most generally applicable ways is "filter promotion". For example, one may try to extend the legality predicate to the approximating sets S_n so that legality of an approximate element s_n is a necessary condition for the legality of a full extension $s \in S_N$ of s_n; formally,

- for $0<n\leq N$, $s \in S_{n-1}$, $x \in C_n$:
  $legal_n\ (cons_n\ x\ s) \Longrightarrow legal_{n-1}\ s$
- and for $s \in S_N$:

legal s ==> $legal_N$ s.

Having found such legality predicates, we define

$S = \{s \mid s <- S'_N;\ legal\ s\}$
$S'_0 = \{s \mid s <- S_0;\ legal_0\ s\}$
$S'_n = \{cons_n\ x\ s \mid x <- C_n,\ s <- S'_{n-1};\ legal_n\ (cons_n\ x\ s)\},\ 0<n$

(It is now easy to prove by induction on n that $S'_n = \{ s \mid s <- S_n;\ legal_n\ s\}$. From this it follows that the above definition of S is equivalent to the original one). With suitable legality predicates the sets S'_n involved in the construction of S are much smaller than the original sets S_n, so that the algorithm is much more efficient. Moreover, there may also exist "cheap" acceptability tests acc_n such that $(legal_{n-1}\ s)\ \&\ (acc_n\ x\ s) = legal_n\ (cons_n\ x\ s)$. Thus we redefine

$S'_n = \{cons_n\ x\ s \mid x <- C_n,\ s <- S'_{n-1};\ acc_n\ x\ s\}$

The program so obtained exemplifies the typical Backtracking behaviour in its evaluation. It may however be improved further by promotion of still other parts of the legality or even acceptability tests. There may also be an opportunity to reduce the sets involved in quite different ways which are peculiar to the problem at hand. One example has already been given in the Eight Queens program; another example of further improvement is shown in Appendix B.

Another, heuristic, way of improving the efficiency concerns the order of generators in the definition of the sets S'_n. In general the time needed to generate a next member of C_n is independent of n, but the time needed to generate a next member of S'_n increases (quite rapidly) with n because of the recursion (i.e. the sets S'_0, S'_1, ..., S'_{n-1}) involved. So we should economize on the demands for further members of S'_n; that is to say, we should let $x <- C_n$ vary fastest and $s <- S'_{n-1}$ vary slowest. Thus we refine the definition of S'_n by imposing a specific order on the generators:

$S'_n = \{cons_n\ x\ s \mid s <- S'_{n-1};\ x <- C_n;\ acc_n\ x\ s\}$

Finally, as we did for the Eight Queens program, we shall now give an alternative formulation of essentially the same algorithm. The idea is to do the recursion bottom-up rather than top-down. Thus we aim at a function S''_n

such that (S''_n s) extends the element s (which should be a member of S'_n) in all possible ways to elements of S'_N; formally, for all n

$$S'_N = \{s' \mid s \leftarrow S'_n,\ s' \leftarrow S''_{n+1}\ s\}$$

so that in particular $S'_N = \{s' \mid s \leftarrow S'_0,\ s' \leftarrow S''_1\ s\}$.
The definition may now be inferred by attempting to prove the above correctness statement by induction on N-n. We find

$S''_{N+1}\ s = \{s\}$
$S''_n\ s = \{s' \mid x \leftarrow C_n;\ acc_n\ x\ s;\ s' \leftarrow S''_{n+1}(cons_n\ x\ s)\},\ n \leq N$

We shall need this formulation in the development of the Branch-and-Bound version; it is also usefull for further transformation into an imperative program.

5. From Backtracking to Branch-and-Bound

In this section we continue the development of the previous section, after a slight adaptation of the problem formulation; namely it is now requested to yield an optimal member of S rather than all members or just one. We assume that some pre-order $\leq$ between the members of S is given; optimality is then nothing but maximality with respect to $\leq$. In the next section The Problem of the Optimal Selection is treated as a concrete example.

Obviously, the optimal solution can be expressed formally as (Max S) where S is defined as in the previous section and Max is defined by

Max {s} = s
Max (s:S) = s, Max S $\leq$ s
= Max S, s $\leq$ Max S

Note. Max is indeterminate for two reasons; the guards do not exclude each other and moreover, even if one of them is replaced by "otherwise", the outcome of Max (s:S) depends on the way the argument set is decomposed into s:S.

Unfortunately this program requires the construction of all members of S, whereas at each point of time during the evaluation only those members of S need be constructed which are at least as good as the maximal element constructed thus far. In other words, we would like to bound the remainder of the search

space dynamically by using the currently found maximal solution as a criterion. This we shall do now.

By way of preparation, suppose that a member m of S is known, playing the role of the maximal one found thus far. Can we then reduce the sets S'_n without the danger of omitting elements necessary for the construction of the ultimate (Max S)? Clearly, (Max S) = Max (m: {s | s <- S'_N; m $\leq$ s}), (assuming for simplicity that (legal s)=($legal_N$ s)). So we may try to promote (part of) the filter m $\leq$ s into the inductive definition of the S'_n. To this end we assume the existence of preorders $\leq_n$ on S x S'_n so that m $\leq_n$ s is a necessary condition in order that s can be extended to s' $\in$ S with m $\leq$ s'. Moreover we assume that there exist predicates $prom_n$ (from: promising) such that

$$m \leq_{n-1} s \;\&\; prom_n\ x\ s\ m \quad = \quad m \leq_n (cons_n\ x\ s)$$

for s $\in$ S'_n and x $\in$ C_n with acc_n x s. Now we can define sets (mS_n m) (from: majorizing m):

mS_0 m = {s | s <- S'_0; m $\leq_0$ s}
mS_n m = {$cons_n$ x s | x <- C_n, s <- mS_{n-1} m; acc_n x s & $prom_n$ x s m}

and it is easy to show that for arbitrary m $\in$ S, (Max S) = Max (m: mS_N m). This completes the preparatory step.

Next we have to try to choose the m-arguments for mS_n in the best possible way, and in particular so that at each point during the evaluation the m-argument is the maximal element produced so far. It seems impossible to achieve this using the above recursive version (at least, we can't succeed); but for the elementwise iterative version we succeed quite elegantly. So, first consider the elementwise iterative version of mS_n, called mS'_n.

mS'_{N+1} m s = {s}
mS'_n m s = {s' | x <- C_n; acc_n x s & $prom_n$ x s m;
s' <- mS'_{n+1} m ($cons_n$ x s) }, n$\leq$N

We transform this definition into a definition of a function MS_n such that

$$MS_n\ m\ s = Max\ (m: mS'_n\ m\ s)$$

The definition reads

```
MS_{N+1} m s = Max {m, s}
MS_n m s =   last M, n≤N
               where
               M = m: [m" | (m',x) <- zip M ``C_n;
                            let ap = acc_n x s & prom_n x s m;
                            let m" = m', ~ap
                                   = MS_{n+1} m' (cons_n x s), ap
                    ]
last [x] = x
last (x:X) = last X, X≠[]
zip X [] = []
zip (x:X) (y:Y) = (x,y): zip X Y
```

Notes. 1. The let-construct let x =expr is short for x <- [expr] or rather x <- [x where x=expr]. In the above text they can also be brought to the expression part of the list abstraction in the form of a where construct.
2. The list $``C_n$ is some nondeterminate enumeration of the set C_n.

In Appendix E we ~~sketch~~ give a formal correctness proof of (MS_n m s) = Max (m: mS'_n m s); let us here explain the definition informally. First of all notice the strong similarity with the definition of mS'_n; MS_n has indeed been obtained from mS'_n by some textual transformation. Next consider the locally defined list M. From its definition we see (by induction) that each member m", computed from (m',x) <- zip M $``C_n$, is the successor of m' in the list, so that #M = 1+#C_n. Further, using (MS_{n+1} m' ($cons_n$ x s)) = Max (m': mS'_{n+1} m' ($cons_n$ x s)) as induction hypothesis, we see that m" $\geq$ m', so that list M consists of successively better and better elements. Moreover, each member of (mS'_n m s) is majorized by at least one element of M, so that indeed Max (m: mS'_n m s) = last M = (MS_n m s).

A requested optimal/maximal member of S is now given by (MS_1 m_0 s_0), where m_0 is some real (or hypothetical, see below) member of S and $\{s_0\}$ is S_0. In case S_0 is not a singleton set, we need to write

```
last M
where
M = m_0: [MS_1 m s | (m,s) <- zip M ``S_0]
```

The initial "currently maximal element" m_0 need not be a real member of S. It

is only used in the program for comparison, via the promising test, and it may well be the case that only one component or one aspect of it need be defined. If however no better element in S exists, it is m_0 that is delivered. So care should be taken in subsequent computations. We shall see applications of this technique in Appendices B and C.

Finally, we rewrite the definition of MS_n so as to obtain a more conventional form: we eliminate the intermediate list M. To this end we introduce an auxiliary, local, function lastM which performs the tasks of last and M simultaneously. We get

```
MS'N+1 m s = Max {m,s}
MS'n m s = lastM m ''Cn, n≤N
              where
              lastM m' [] = m'
              lastM m' (x:C) = lastM m" C
                               where ap = --- as above ---
                                     m" = --- as above ---
```

maar met MS'_{n+1} ipv MS_{n+1}.

6. The Problem of the Optimal Selection

In this section we develop a program for the Optimal Selection problem along the lines sketched in the previous section. The problem is typical for Branch-and-Bound.

There is given a bag B containing N objects; each object x ∈ B has its own weight (w x) and value (v x).*
It is requested to make a selection of the objects so that their aggregate weight does not exceed a given limit W and their aggregate value is as large as possible.

It is not hard to cast the problem into the general terms used in the schematic treatment of Backtracking and Branch-and-Bound. For example, after numbering the objects, the set of all selections, both legal and illegal, is just the powerset of {1..N} and we might express this set by (powerset N) where

---

*) Assuming that "objects" are by definition mutually different (because their identities differ), one could also take a set S of objects.

```
powerset 0 = {{}}
powerset n = {x ++ s | x <- {{n}, {}}, s <- powerset (n-1)}
```

So, in this case 'cons' is 'append' and C_n = {{n}},{}}. It seems however more natural to treat the two choices for x separately and have the objects rather than a number as parameter of powerset. Therefore we define

```
pset {} = {{}}
pset (x:X) = {x:s | s <- pset X} ++ pset X
```

Consequently we cannot merely instantiate the program schemata of the previous section, but have to redo the development again. The problem may thus be solved by

```
Max {s | s <- pset B; legal s}
```

where legal and $\leq$ (in terms of which Max has been expressed) are given by

```
legal s = wgt s ≤ W
s ≤ s' = val s ≤ val s'
wgt s = sum {w x| x <- s}     *)
val s = sum {v x| x <- s}
```

Clearly this program is impractical for it demands the inspection of all 2^N subbags of B. Even for moderately small values of N this will take too much time. Fortunately it is easy to reduce the sets involved considerably by promoting the legality test into the construction of selections; for once a selection is illegal, it can't become legal any more by adding elements. So we define the sets of legal selections:

```
sel {} = {{}}
sel (x:X) = {x:s | s <- sel X; acc x s} ++ sel X
acc x s = w x + wgt s ≤ W.
```

and the problem is now solved by Max (sel B).

Now we shall apply the technique of Bounding the search space, as given in the previous section. We first give the elementwise iterative version of sel.

---

*) In case s is a set rather than bag, one should write 's instead of s.

```
sel' s { } = {s}
sel' s (x:X) = {s'| acc x s; s' <- sel' (x:s) X} ++ sel' s X
```

Next we need to transform this definition into one of a function msel so that (msel m s X) = {s'| s' <- sel s X; s'$\geq$ m}. To this end we need a "promising" predicate; in the construction of selections an approximate or partial selection s looks promising only if its value together with the value of the still remaining objects is at least the value of m. So we define

```
msel m s {} = {s}
msel m s (x:X) = {s' | acc x s; s' <- msel m (x:s) X}
                 ++ {s' | prom s X m; s'<- msel m s X}
prom s X m = val s + val X ≥ val m
```

Finally we transform msel into Msel so that

```
  Msel m s X = Max (m: msel m s X)
```

The transformation scheme is given in the previous section. The actual instance here is somewhat simpler, because the set C_n contains only two choices and these are dealt with separately in msel. The definition thus reads:

```
Msel m s {} = Max {m, s}
Msel m s (x:X) = m2
                 where m1 = Msel m (x:s) X, acc x s
                          = m             , otherwise
                       m2 = Msel m1 s X, prom s X m
                          = m1           , otherwise
```

An optimal selection is now given by (Msel {} {} B). This completes the major development of the program. What remains is some minor improvements such as the addition of some redundant parameters denoting (val m), (val s), (wgt s) and (val X) in order to avoid frequent recomputation of these values; (an easier but less efficient method to avoid these recomputations is to memoize the functions wgt and val).

Note, (for the expert semanticist only). Semantic equivalence, denoted by the symbol =, means that both sides yield the same value if they are determinate and yield the same possible values if they are indeterminate. In this sense the

equality (Msel m s X) = (Max (m: msel m s X)) holds. When the righthand side (Max {m,s}) in the first clause of Msel is replaced by s alone, then (Msel m s X) $\geq$ x for any x $\in$ X but --assuming X is a list-- it will yield the last such element of X and is therefore not semantically equal to Max (m: msel m s X).

Appendix A. Stable Marriages

The problem statement

There are given N men and N women, numbered 1 through N respectively 1+N through 2*N. Each person has its own preference list for the persons which he/she would like to be married; (prefs k) is the list of person k, and this list enumerates the persons of the other sex in order of decreasing preference. Computable from prefs, as well as the other way round, is the function rks (from: ranks); (rks k k') gives the place of k' in the preference list of k, i.e. for all k and k' of different sexes, (prefs k)!(rks k k') = k, (we use ! to denote list subscription). It is requested to arrange a marriage of the men with the women so that no two marriages are instable. We call two marriages (i,j) and (i',j') instable if i prefers j' to j and at the same time also j' prefers i to i':

```
instab (i,j) (i',j') = rks i j'< rks i j & rks j' i < rks j' i'
```

Notice that instab is not symmetric in its arguments.

The program development

Our first program is very close to the above problem formulation; it is an executable specification. Define

```
pair {} {} = {{}}
pair X Y = {(x,y):P | (x,y) <-X*Y, P <- pair (X--{x})(Y--{y})}
stable P = all {~instab p q | p <- P, q <- P--{p}}
         = all {~instab p q & ~instab q p | p,P'<<- P, q <- P'}
```

Then one has all solutions to the marriage problem in hand by {P | P <- pair {1..N} {N+1..2*N}; stable P}. Of course, X*Y = {(x,y) | x <- X, y <- Y}.

We see now two mutually independent ways to improve the efficiency: one is to improve the definition of pair and the other is to apply filter promotion. We deal with them in order.

Improvement of pair. By definition sets cannot contain duplicate elements, so somewhere in the construction or the use of sets an equality test has to be done. The above definition of pair generates in many ways one and the same set;

or in other words, had the members of (pair X Y) be bags or lists rather than sets, then (pair X Y) would contain many duplicates! We can avoid those duplicates by construction; for example by restricting the choice for x to one (nondeterminate!) possibility. At the same time we shall represent pairings by bags so that the equality test are avoided indeed.

```
pair {} {} = {{}}
pair (x:X) Y = {(x,y):P | y <- Y, P <- pair X (Y--{y})}
```

Filter promotion. We observe that an instable pairing P cannot get stable by extending it. We may therefore reduce the sets considerably by promoting the stability filter:

```
pair {} {} = {{}}
pair (x:X) Y = {(x,y):P | y <- Y, P <- pair X (Y--{y}); acc (x,y) P}
acc p P = all {~instab p q & ~instab q p | q <- P }
```

(Remember, an abstraction {...|...} is a list, bag or set according to the kind of the generator domain.) This completes the major steps in the program development. We shall now speed up the algorithm once more by a factor of about two, by halving the instability tests. We observe

```
acc (i,j) P
= all {~instab (i,j) (i',j') & ~instab (i',j') (i,j) | (i',j') <- P }
= all {~instab (i,j) (i',j') | (i',j') <- P } &
  all {~instab (i',j') (i,j) | (i',j') <- P }
= all {~instab (i,j) (i',j') | j'<-prefs i; married P j'; let i'=partner
  j' P} &
  all {~instab (i',j') (i,j) | i'<-prefs j; married P i'; let j'=partner
  i' P}
```

Here, married is to be a function such that (married P j) = true precisely when there is some pair (i,j) in P, and similarly for (married P i). Analogously, (partner j P) = the i such that (i,j) ∈ P. (Note that P is a bag whereas (prefs i) is a list; the abstractions change accordingly from bags to lists.) Now we see that in the first conjunct the j's are enumerated in order of decreasing preference of i, so that initially (rks i j' < rks i j) is steadily true and finally steadily false. So we need only enumerate (prefs i) upto j itself; the test (rks i j' < rks i j) will yield true for all these values and can therefore be omitted. A similar reasoning applies to the second

conjunct. We thus get

```
pair :- as before but with acc' instead of acc
acc' (i,j) P
= all {rks j'i≥rks j'i' | j'<-upto j (prefs i); married P j'; let i'=partner
j' P} &
  all {rks i'j≥rks i'j' | i'<-upto i (prefs j); married P i'; let j'=partner
i' P }
upto x {} = {}
upto x (y:Y) = {}, x=y
             = y: upto x Y, x≠y
```

The test (married P j') can be implemented as j' ∉ Y where Y is the set (bag or list) of women not yet married; this set is available at the place of invocation of acc' and can be given to acc' as an additional parameter. The test (married P i') can be implemented analogously, but there is a better method too. The current program has a set X, initially {1..N}, as argument for pair. There is no objection to keeping that set in increasing order, that is to represent that set by a list, without duplicates, initially [1..n]. It then happens that we can implement (married P i') simply by i'≤ x where [1..x] are the men already married; this x is available at the place of call of acc' and can be given as an additional argument to acc'. It remains to implement partner. One possible definition reads

```
partner k P = j where (k',j) = hd {(k',j) | (k',j) <- P; k'= k}, k≤N
            = i where (i,k') = hd {(i,k') | (i,k') <- P; k'= k}, k>N
```

Another possibility is to represent P as an array [1..2*N] of 1..2*N so that P[k] = (partner k P). This representation is the obvious choice if one wants to implement the above algorithm in a Pascal-like language.

We conclude this section by presenting the elementwise iterative version. All stable marriages are given by (pair {} 1 {N+1..2*n}), where

```
pair P x {} = {P}, x=N+1
pair P x Y = {P' | j<-Y; acc'(x,j) P; P'<-pair ((x,j):P) (x+1) (Y--[j])}, x≤N
acc' :- as before
```

Note. The x-parameter of pair stands for the list [x..N].

***

An implementation in Pascal, based on the above elementwise iterative version of pair, reads as follows. The P and Y parameters of pair are available as the values of global variables Partner and Y respectively.

```
var N: integer;
    prefs : array [1..2*N] of array [1..N] of 1..N*2;
    rks   : array [1..2*N, 1..2*N] of 1..N;
            {only rks[x,y] with x ∈ 1..N & y ∈ N+1..2*N and
             x ∈ N+1..2*N & y ∈ 1..N are needed}
--initialize N and prefs and compute rks--

var Y : array [N+1..2*N] of boolean;
    Partner: array [1..2*N] of 1..2*N

procedure pair (x: integer);
    var j: integer;
    function acc (i,j: integer): boolean;
            var a: boolean; {function result}
                i',j': integer;
                t, tlimit: integer; {loop variable}
        begin a := true;
              t := 1; j' := prefs [i,t];
              while j' ≠ j & a
              do    if not Y[j']
                    then  a:= rks [j',i] ≥ rks [j', Partner[j']] fi
              od;
              t := 1; tlimit := rks [j,i]; i' := prefs [j,t];
              while t ≠ tlimit & a
              do if i'≤ x
                 then a := rks [i',j] ≥ rks [i', Partner[i']] fi
              od;
              acc := a
        end {acc};
    begin if x= N+1 then output-Partner else
          for j := N+1 to 2*N
          do if Y[j] then if acc(x,j)
             then Y[j] := false; Partner[x] := j; Partner[j] := x;
                  pair (x+1);
                 {Partner[x] := Partner[j] := undef;} Y[j} := true
```

```
            fi fi
         od fi
    end

--the main call reads--
for j := N+1 to 2*N do Y[j] := true od; pair (1)
```

Appendix B. Optimal Partition

There is given a set S of objects, each object x having the value (v x). It is requested to partition the objects into M groups. We shall number the groups with 1..M and define for a partition P and m ∈ 1..M:

grp P m = the objects in the m-th group of P
val P m = sum {v x | x <- grp P m}
maxv P = max {val P m | m <- {1..M}}

Actually, the request is not simply to yield a partition, but to yield a partition P for which (maxv P) is minimal.

Our first duty is to formulate the requirements as clear as possible. The value requested is (Min (parts S)) where

parts {} = {emptyPartition}
parts (x:X) = {add m x P | m <- {1..M}, P <- parts X}
Min :- standard, defined in terms of P ≤ P'
P ≤ P' = maxv P ≤ maxv P'

We shall postpone choosing a representation for partition as long as possible; consequently we cannot define empyPartition and add, but we can fomulate the essential property which they should satisfy:

grp emptyPartition m = {} for all m ∈ 1..M
grp (add x P m) m' = grp P m', m' ≠ m
= x: grp P m', m' = m

Next we observe that two partitions which differ only in their numbering are to be considered the same. This, at least, seems to be the intention in the problem statement. Notice, for instance, that the numbering is introduced only to facilitate the definition of maxv, and that maxv itself is independent of the particular numbering used. (Exercise. Reformulate the problem statement and the program obtained so far, avoiding the numbering completely; let a partition be a bag of sets.) Let us therefore try to reduce the sets (parts X) (thus increasing the efficiency of the algorithm), by generating only nonequivalent partitions. Suppose (neqparts X) is the set of nonequivalent partitions of X; how can we form (neqparts (x:X))? Let P, P' ∈ (neqparts X) and m ≠ m'. Clearly (add m x P) and (add m' x P) are equivalent if both (grp P m) and (grp P m') are empty; a simple exchange of the m-th and the m'-th group transforms the partitions into each other. Conversely, assume (add m x P) be equivalent to

(add m' x P'), say via renumbering r. In view of the uniqueness of object x, it must be true that r renumbers m into m' and that (grp m P) equals (grp m' P'). Hence we find that P and P' are equivalent via r. For $P \neq P'$ this cannot be true, because P and P' $\notin$ (neqparts X). So P' equals P and, again in view of the uniqueness of objects, r can only exchange the numbers of empty groups; it follows that both (grp P m) and (grp P m') are empty. Therefore, in order to construct (neqparts (x:X)), the choice for m with (grp P m) empty should be restricted to precisely one possibility; say the minimal such value. So, m is to vary over

```
{1.. min {M, legP}}
```

where (leg P) = number of least numbered empty group of P, (taken to be M+1 if no group is empty). Thus

```
neqparts {} = {emptyPartition}
neqparts (x:X) = {add m x P | m <- {1..min{M, leg P}},
                              P <- neqparts X}
```

(We shall define leg together with add, grp and emptyPartition.) The value requested is (Min (neqparts S)).

Next we want to apply the Bound technique of Section 5. To do so, we need the elementwise iterative version of neqparts. This one reads

```
neqparts P {} = {P}
neqparts P (x:X) = {P' | m <- {1..min{M,leg P}},
                         P' <- neqparts (add m x P) X}
```

From this we make the version mneqparts such that

$$\text{neqparts } P\ X \supseteq \text{mneqparts } Q\ P\ X \supseteq \{P' \mid P' \leftarrow \text{neqparts } P\ X;\ P' \leq Q\}$$

for any partition Q of all objects. Fortunately, the relation $\leq$ can be easily extended: an approximate partition P is doomed to be worse than Q if (maxv P) already exceeds (maxv Q); this can be formulated by means of the original (definition of) $\leq$ between full partitions. Now, if $P \leq Q$ then (add m x P) $\leq$ Q is equivalent to (grp m (add m x P)) $\leq$ (maxv Q), i.e. (val P m) + v x $\leq$ maxv Q. Hence,

```
mneqparts Q P {} = P
mneqparts Q P (x:X) = {P'| m <- {1..min{M,leg P}};
                           prom x m P Q;
                           P' <- mneqparts Q (add m x P) X}
prom x m P Q = val P m + v x ≤ maxv Q
```

This then is tranformed into the final program in the standard way. We get

```
Mneqparts Q P {} = Min{Q,P}                    --or even P itself
Mneqparts Q P (x:X)
 = last Qs
   where
   Qs = Q : [ Q" | (Q',m) <- zip Qs [1..min{M,leg P}];
                  let prm = prom x m P Q';
                  let Q" = Mneqparts Q (add x m P) X, prm
                         = Q'                       , ~ prm
            ]
```

and the requested partition is expressed by

$$\text{Mneqparts } Q_o \text{ emptyPartition } S$$

where Q_o is some "hypothetical" partition such that (maxv Q_o) is infinite or at least large enough, say sum {v x | x <- ''S}. Of course, we can also construct some real partition Q_o by e.g. distributing S evenly over the M groups:

```
Q_o = last Qs
      where
      Qs = emptyPartition: [add (1 + n mod M) x Q |
                                (Q,x,n) <- zip (Qs, ''S, [0..])]
```

It remains to give a representation for partitions and to define emptyPartition, add, leg (and possibly Q_o) accordingly. There are umpty ways to do so. We shall work out here the choice to represent a partition P as a two-tuple, consisting of

- the value (leg P), and
- the list [(grp P 1, val P 1), ..., (grp P M, val P M)]

Then

```
emptyPartition = (1, [({}, 0) | m <- [1..M]])
grp P m = group where P = (leg,list); (group,val) = list!m
val P m = val   where P = (leg,list); (group,val) = list!m
add m x P = (leg',list')
            where (leg, list) = P
                  (group, val) = list!m
                  leg' = max {m+1, leg}
                  list' = take list (m-1) ++
                          [(group',val')] ++
                          drop list m
                  group' = x : group
                  val' = v x + val
```

It is also easy to define the hypothetical partition Q_o:

```
Q_o = (dontcare, ({}, maxnbr): [({}, 0) | m <- [2..M])
```

But actually we had better improve the program slightly by adding an extra parameter to denote (maxv Q), and, if done so, we can simply call the program with maxnbr for this parameter and let Q_o completely undefined.

Note. An inexperienced programmer may be tempted to exploit the knowledge of the representation given above elsewhere. If that is not the intention, then this unintentional use of the representation can be made illegal by adding the following abstract data type definition.

```
absttype Prepr
    with emptyPartition :: Prepr;
         val :: Prepr -> Nbr -> Nbr
         grp :: Prepr -> Nbr -> {objecttype}
         leg :: Prepr -> Nbr
         add :: Nbr -> objecttype -> Prepr -> Prepr
```

and defining, say together with val, grp, and so on:

```
Prepr == (Nbr, [({objecttype}, Nbr)]
```

Finally we present the imperative program suggested by the above functional

program. We shall represent a partition by two arrays, one giving for each object the group to which it belongs and another giving for each group the value. The leg-value of the partition is available as value-parameter; the arrays are globally available. The objects are identified with the numbers 1..N, and the value-parameter n represents the set n..N of objects still to be distributed over the groups.

```
var Q,P : array [1..N] of [1..M];
    maxvQ : integer;
    valP : array [1..M] of integer;

procedure Mneqparts (n:integer; leg: 1..M+1);
begin if n = N+1
      then Q := P;
           maxvQ := 0; for m to M do maxvQ :+ valP [m] od
      else for m to min (M, leg)
           do if valP [m] + v(n) ≤ maxvQ
              then P[n] := m; valP[m] :+ v(n);
                   Mneqparts (n-1, max(m+1, leg));
                   {P[n] := undef;} valP[m] :- v(n)
              fi
           od
      fi
end
```

The main call should read

```
maxvQ := maxint;
for m to M do valP[m] := 0 od;
Mneqparts (1, 1);
output (Q, maxvQ)
```

Appendix C. Well-balanced Partition

A set S of objects is given, each object x having weight (w x). We consider three problem statements: it is requested to partition the objects in two groups such that

a. the difference in weight between the groups is minimal;
b. that difference is zero, (if such a partition exists);
c. that different is again zero but this time with the additional requirement that the difference in number of objects in the groups is maximal.

We shall treat variants (a), (b), (c) in succession. We shall be very brief in our explanation, because the problem looks quite similar to the problem of the Optimal Partition, treated in Appendix B.

Variant (a).
We use P to denote a partition, (L,R) to denote the left and the right group of a partition. Some auxiliary functions of general utility are:

```
Max :- standard, defined in terms of P ≤ P'
P ≤ P' = abs (dw P) ≥ abs (dw P')
dw (L,R) = wgt L - wgt R                    --difference in weight
wgt X = sum {w x | x <- X}
```

A best-balanced partition is now given by (Max (parts S)) where

```
parts {} = {({}, {})}
parts (x:X) = {(x:L, R) | (L,R) <- parts X} ++ {(L,x:R) | (L,R) <- parts X}
            = {P | (L,R) <- parts X, P <- {(x:L,R), (L,x:R)}}.
```

Actually we have been overspecific: a partition should be a bag of two groups rather than a tuple of two groups. So partitions (L,R) and (R,L) are considered to be the same, or equivalent. We may reduce therefore the set (parts S) by a factor 2 by generating only nonequivalent partitions, say by fixing one element into the left group.

```
parts' {} = {({}, {})}
parts' (x:X) = {(x:L,R) | (L,R) <- parts X}
```

The required result is now (Max (parts' S)). The elementwise iterative version of parts' reads

```
parts' {} = {({}, {})}
parts' (x:X) = parts ({x},{}) X
parts P {} = {P}
parts (L,R) (x:X) = parts (x:L,R) X ++ parts (L,x:R) X
```

Next we look for a necessary condition in order that an approximate partitioning P can be extended with elements from X to a full partitioning that is at least as good as a given partitioning Q. The --or rather one such-- condition is that the lightest of the two groups of P can be made so heavy with the remaining elements that, on a balance, its pan is below the lightest pan of Q:

```
prom (L,R) Q X = dw (X++L,R) ≥ -abs(dw Q) , dw(L,R) ≤ 0
               = dw (L,X++R) ≤ +abs(dw Q) , dw(L,R) ≥ 0
```

An alternative formulation of this same condition is

```
prom (L,R) Q X   = true                  , -adw ≤ dwP ≤ +adw
                 = dw (X++L,R) ≥ -adw , dwP ≤ -adw
                 = dw (L,X++R) ≤ +adw , dwP ≥ +adw
                 where
                 adw = abs (dw, Q)
                 dwP = dw (L,R)
```

This enables us to define mparts so that (parts P X) ⊇ (mparts Q P X) ⊇ {P' | P' <- parts P X; Q ≤ P'}, provided that the last set is nonempty. (Actually the second inclusion is an equality.)

```
mparts' Q {} = {P | Q ≤ P} where P = ({}, {})
mparts' Q (x:X) = {P' | prom P Q X; P' <- mparts Q P X}
                  where P=({}, {})
mparts Q P {} = {P}
mparts Q (L,R) (x:X)
     = {P | prom (n:L,R) Q X; P <- mparts Q (n:L,R) X} ++
       {P | prom (L,n:R) Q X; P <- mparts Q (L,n:R) X}
     = {P | P' <- {(n:L,R), (L,n:R)}; prom P' Q X; P <- mparts Q P' X}
```

This definition then is transformed to the required definition of Mparts so that (Mparts Q P X) = (Max (Q: mparts Q P X)).

```
Mparts' Q {} = {P | Q ≤ P} where P = ({}, {})
Mparts' Q (x:X) = Q, ~prom ({x},{}) Q X
                = Mparts Q ({x}, {}) X , otherwise
Mparts Q P {} = Max {Q, P}    --or simply P
Mparts Q (L,R) (x:X) = Q2
                       where
                       Q1 = Mparts Q (n:L,R) X , prom (n:L,R) Q X
                          = Q                  , otherwise
                       Q2 = Mparts Q1 (L,n:R) X, prom (L,n:R) Q X
                          = Q1                 , otherwise
```

A best balanced partition is delivered by (Mparts' Q_o S) where Q_o is some --possibly hypothetical-- partition. Actually only (dw Q_o) need be defined; nothing else of it is used by Mparts. And only if no better partition exists, Q_o will be delivered; (this requires to replace Max{Q, P} by P in the first clause of Mparts). So we may even choose a hypothetical Q_o with (dw Q_o) = 0; in this case (Mparts Q_o S) yields a well-balanced partition if it exists and Q_o otherwise. Thus variant (b) is solved as well.

Variant (c).
Now it is requested to yield a partition with no difference in weight between the groups and a maximal difference in numbers of elements in the groups. So the comparison relation between partitions reads

```
P ≤ P' = abs (dw P) = 0 & abs (dc P) ≤ abs (dc P')
dc (L,R) = #L - #R                    --difference in cardinality
```

We shall avoid equivalent parititions by forcing #L to be minimal and #R to be maximal. (This choice of avoiding equivalent partitions allows for a better adaptation of prom than the choice to fix one specific element in one specific group.) What necessary condition can we think of for an approximate partitioning to be extendable to someone that is better than Q? Again, the lightest of the pans of P must be made heavy "enough" (as with variant (a) and (b)), but this time not with all remaining elements X but only with the k heaviest from them, where k is the maximal number of objects still allowed on that pan (without becoming worse than Q). So we define

```
Hst X k = the k heaviest elements from X
Lst X k = the k lightest elements from X
```

```
prom (L,R) (LQ, RQ) X
   = #L ≤ #LQ & #RQ ≤ #R & prm
     where
     kL = #LQ - #L        --maximally to be added to L
     kR = #RQ - #R        --minimally to be added to R
    prm = dw (Hst X kl ++ L, Lst X kR ++ R) ≥ 0 , dw (L,R) ≤ 0
        = dw (L, X ++ R) ≤ 0                    , dw (L,R) ≥ 0
```

The remainder of the program outline is the same as in the previous variants. Let us therefore consider only Hst and Lst. If we keep the argument X of elements still to be distributed, sorted in order of decreasing weight, then Hst and Lst are quite simple to define and to compute. For example, for Hst' X k = wgt(Hst X K):

```
Hst' X 0 = 0
Hst' (x:X) k = w x + Hst X (k-1) , k > 0
```

If we now also make Hst' into a lazy memo function, see [Hughes 1985], then frequent recomputation is eliminated at once. Alternatively, we may compute beforehand a table with for each tail X of the sorted list S' and for each suitable k the value (Hst' X k) as entry.
This can be done in n^2 time, where n = #S.

Further details are left to the industrious reader.

Appendix D. Imperative program for the Eight Queens

In this section we present imperative Pascal-like programs inspired by the functional programs developed in Section 3. Recall those programs:

```
LP = P" 8
P" 0 = {[]}
P" n = {(n,j):p | p <- P" (n-1); j <- {1..8}; acc' (n,j) p}, 0 < n
acc' q p = all {~ check' q q' | q' <- p}
check' (i,j) (i',j') = j=j' \/ i+j=i'+j' \/ i-j=i'-j'
```

```
LP = P1 [] 1
P1 p 9 = {p}
P1 p n = {p' | j <- {1..8}; acc'(n,j) p; p' <- P1((n,j):p)(n+1)}, n ≤ 8
acc', check' :- as above.
```

We shall first transform the latter into an imperative program. Our aim is to have one globally available array in which at each time the current argument p of P_1 is stored. We assume that the output, the placements of 8 queens, has to be printed.

```
var p : array [1..8] of 1..8;

proc P1 (n: integer);
   {p[1..n-1] is significant at entry}
begin if n=9 then output(p)
      else for j := 1 to 8
           do if acc (n,j) then
              begin p[n] := i;
                    P1 (n+1)
                   {p[n] := undef}
              end
           od
      fi
end
```

The main call should read: P_1 (1).

Notice the close correspondence between this imperative formulation and the

functional program. In particular the generator j <- {1..8} now appears as for j := 1 to 8.

It is slightly problematic to translate the first functional formulation into a Pascal-like language, if we still insist on using one global array to store the arguments of P". The way out is to give P" an additional parameter, namely the process to be performed on the results of P". The generator p <- P"(n-1) is then translated as a single call with the process "j <- {1..8}; acc' (n,j) p" as additional argument.

```
var p: array [1..8] of 1..8;

proc P" (n: integer; proc process);
   {all possible/legal placements are generated in p[1..n]
    and each of them is subject to 'process'}
    proc new-process;
    begin for j := 1 to 8
          do if acc (n,j) then
             begin p[n] := j; process {; p[n] := undef}
             end
          od
    end;
begin if n=0 then process else P" (n-1, new-process) fi
end

main call: P" (8, output-p)
```

The use of the additional process parameter seems a bit unconventional. We can avoid it by translating the functional program into coroutines; in the present case P" becomes a recursive coroutine. We use the notation of [Tennent 1981].

```
var p: array [1..8] of 1..8

coroutine P" (n: integer);
var more: boolean;                          --accessible from outside.
initial                                     --just a key-word.
  var j: integer;                           --private local variable.
  begin
      more := true;
      if n=0
```

```
      then detach                          --gives control back to caller.
      else Q.P" (n-1);                     --recursive activation, named Q.
           while Q.more                    --the 'more' variable of Q.
           do    for j:=1 to 8
                 do if acc (n,j)
                    then p[n] := j;
                         detach
                    fi
                 od;
                 call Q                    --gives control again to Q,
                                           --(resumption of Q)
           od
      fi;
      more := false; detach
  end

main call: Q_o.P"(8);
           while Q_o.more do output-p; call Q_o od
```

Appendix E. Correctness proof of Branch-and-Bound

Here we shall prove the correctness of the most tricky transformation, namely $(MS_n\ m\ s) = (Max\ (m{:}\ mS_n\ m\ s))$ where MS_n and mS_n are defined by

$mS_{N+1}\ m\ s = \{s\}$

$mS_n\ m\ s = \{s' \mid x \leftarrow C_n;\ ap_{x,m};\ s' \leftarrow mS_{n+1}\ m\ cons_x\},\ n \leq N$

$MS_{N+1}\ m\ s = Max\ \{m, s\}$

$MS_n\ m\ s = last\ M\ ,\ n \leq N$

where

$M = m{:}\ [m'' \mid (m', x) \leftarrow zip\ M\ ``C_n;$

$\quad \underline{let}\ m'' = m' \quad , \sim ap_{x,m'}$

$\quad\quad = MS_{n+1}\ m'\ cons_x \quad , \quad ap_{x,m'}]$

Note that the above function mS_n is called mS'_n in Section 5. The proof is independent of the expressions $ap_{x,m'}$ and $cons_x$; we have indicated the occurrences of x and m in these expressions by subscripts, so that substitution of c for x and m for m' can be denoted by $ap_{c,m}$ and $cons_c$. The free occurrences of s and n in $ap_{x,m}$ and $cons_x$ are not explicitly indicated.

We prove the equality by induction on N+1-n.
The case n=N+1 is simple:

$MS_{N+1}\ m\ s$

$= \{def\ MS\}\ Max\ \{m, s\}$

$= Max\ (m{:}\ \{s\})$

$= \{def\ mS\}\ Max\ (m{:}\ mS_{N+1}\ m\ s)$

The case $n \leq N$ is less simple; the remainder of this appendix is devoted to it.

The induction hypothesis reads

(1) $MS_{n+1}\ m\ s = Max\ (m{:}\ mS_{n+1}\ m\ s)$ for all m, s.

We shall use this in the following lemma.

Lemma

Let C be finite

and $M = m{:}\ [m'' \mid (m', x) \leftarrow zip\ M\ ``C;\ \underline{let}\ m'' = m' \quad , \sim ap_{x,m'}$

$\quad\quad = MS_{n+1}\ m'\ cons_x, ap_{x,m'}]$

Then $last\ M = Max\ (m{:}\ \{s' \mid x \leftarrow C;\ ap_{x,m};\ s' \leftarrow mS_{n+1}\ m\ cons_x\})$

proof

By induction on the structure (or legnth) of ``C.

Case ``C = []. Then last M = last (m: []) = m =
    Max (m: {}) = Max (m: {s' | x<-{}; ---}) q.e.d.
Case ``C = c:C'. We distinguish two subcases.
subcase ~$ap_{c,m}$ holds:
   last M
   ={def M,C} last (m: M')
    where M' = [m" | (m',x)<-zip (m:M') (c:C'); let m" = ---]
   ={expanding M'} last (m: m_1: M")
    where m_1 = m {because ~$ap_{c,m}$ holds}
          M" = [m" | (m',x)<-zip M' C'; let m" = ---]
   ={last,m_1} last (m: M")
    where M" :- as above
   ={induction hypothesis of the Lemma}
    Max (m: {s' | x<-C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$})
   ={because ~$ap_{c,m}$ holds}
    Max (m: {s' | $ap_{c,m}$; s'<-mS_{n+1} m $cons_c$} ++
            {s' | x<-C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$})
   =Max (m: {s' | x<-c:C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$}) q.e.d.
subcase $ap_{c,m}$ holds:
   last M
   ={def M,C} last (m: M')
    where M' = [m" | (m',x)<-zip (m:M') (c:C'); let m" = ---]
   ={expanding M'} last (m: m_1: M")
    where m_1 = MS_{n+1} m $cons_c$ {because $ap_{c,m}$ holds}
          M" = [m" | (m',x)<-zip M' C'; let m" = ---]
   ={last, induction hypothesis (1)} last (m_1: M")
    where m_1 = Max (m: mS_{n+1} m $cons_c$)
          M" :- as above
   ={induction hypothesis of the Lemma}
    Max (Max (m: mS_{n+1} m $cons_c$):
         {s' | x<-C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$})
   ={property of Max}
    Max (m: mS_{n+1} m $cons_c$ ++ {s' | x<-C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$})
   ={because $ap_{c,m}$ holds}
    Max (m: {s' | x<-c:C'; $ap_{x,m}$; s'<-mS_{n+1} m $cons_x$})
   =Max (m: mS_n m s) q.e.d.
This completes the proof of the Lemma.

Now the induction step in the main proof is simple:
   MS_n m s

$=\{$def MS$\}$ last M

where M = m: [--- ''C_n ---]

$=\{$lemma, taking $C=C_n\}$

Max (m: $\{s' \mid x<-C_n;\ ap_{x,m};\ s'<-mS_{n+1}\ m\ cons_x\}$

$=$Max (m: mS_n m s) q.e.d

This completes the main proof.

Literature

Fokkinga, M.M., Nondeterminisme moet lazy zijn.
Hand written note, dec 1985.

Hughes, J., Lazy memo-functions. In Functional Programming Languages and Computer Architecture, (ed JP Jouannaud), Springer-Verlag, LNCS 201, (1985) pp 129-146

Meertens, L., Algorithmics - towards programming as a mathematical activety. Proc. CWI Symp on Mathematics and Computer-Science, CWI Monographs Vol 1 (eds JW de Bakker, M Hazewinkel, JK Lenstra), North Holland, 1986, pp 289-234.

Tennent, R.D., Principles of Programming Languages.
Prentice-Hall, 1981.

Wadler, P., How to replace failure by a list of successes. In Functional Programming Languages and Computer Architecture, (ed. J.P. Jouannaud). Springer, LNCS 201 (1985) pp 113-128