A Generic Program for Sequential Decision Processes

Oege de Moor

Programming Research Group, Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

1 Motivation

A *generic program* for a class of problems is a program which can be used to solve each problem in that class by suitably instantiating its parameters. The most celebrated example of a generic program is the algorithm of Aho, Hopcroft and Ullman for the *algebraic path problem* [1]. Here we have a single program, parameterised by a number of operators, that solves numerous seemingly disparate problems. Furthermore, the applicability of the generic program is elegantly stated as the condition that the parameters form a *closed semi-ring*.

Examples of such generic algorithms are admittedly scarce, and one might be led to believe that the majority of programs cannot be expressed in such a generic manner. I personally do not share that view, for the following two reasons:

Firstly, little effort has gone into attempts to classify existing algorithms, reflecting the fact that the computing community as a whole places more value on the invention of new algorithms than on the organisation of existing knowledge. In certain specialised areas where organisational tools are indispensable (for example in the design of architecture-independent parallel algorithms), numerous generic algorithms have been discovered, and form the core of the subject [13].

Secondly, to express generic algorithms one requires a higher-order notation in which both programs and types can be parameters to other programs. Such higher-order notations encourage the programmer to exploit genericity where possible, and indeed many functional programmers now do so as a matter of course. More traditional programming notations do not offer similar support for writing generic programs.

This paper is an attempt to persuade you of my viewpoint by presenting a novel generic program for a certain class of optimisation problems, named *sequential decision processes*. This class was originally identified by Richard Bellman in his pioneering work on dynamic programming [4]. It is a perfect example of a class of problems which are very much alike, but which has until now escaped solution by a single program.

Those readers who have followed some of the work that Richard Bird and I have been doing over the last five years [6, 7] will recognise many individual examples: all of these have now been unified. The point of this observation is that even when you are on the lookout for generic programs, it can take a rather long time to discover them. The presentation below will follow that earlier work, by referring to the calculus of relations and the relational theory of data types. I shall however attempt to be light on the formalism, as I do not regard it as essential to the main thesis of this paper. Undoubtedly there are other (perhaps more convenient) notations in which the same ideas could be developed. This paper does assume some degree of familiarity with a lazy functional programming language such as *Haskell*, *Hope*, *Miranda*[1] or *Gofer*. In fact, the LaTeX file used to produce this paper is itself an executable Gofer program.

2 Sequential Decision Processes

Many optimisation problems can be specified in the form

```
 spec r p gen = listmin r . filter p . gen
```

Here the *generator* `gen` generates a list of candidate solutions, `filter p` selects those solutions which are feasible (those that satisfy the predicate `p`), and finally the *selector* `listmin r` picks a minimum solution according to the order relation `r`.

Sequential decision processes are distinguished from other optimisation problems in the way the generator is formulated. The *sequential* nature of the generator is captured by expressing it as an instance of the function `fold`[2]. Informally, `fold` is defined by the equation

```
 fold add e  [a1,a2,...,an] = a1 `add` (a2 `add` ... (an `add` e))
```

In words one might say that `fold add e x` traverses the list `x` from right to left, starting with the seed value `e`, and summing with the operator `add` at each step. The recursive definition of `fold` is

```
>fold add e [] = e
>fold add e (a:x) = a `add` (fold add e x)
```

In a sequential decision process, we have

```
   gen = fold f e
```

[1] Miranda is a trademark of Research Software Ltd.
[2] The function `fold` is normally called `foldr`. Because I shall deviate from the common definition of `foldr1`, I have decided to use `fold` and `fold1` to avoid confusion.

and at each step, the operator `f` offers a nondeterministic choice between a number of alternatives. It is this choice that accounts for the term *decision* process.

The alternatives are presented as a list of operators, and the specification of a sequential decision process therefore reads

```
>sdp_spec r p fs c = listmin r . filter p . fold (choice fs) [c]
>choice fs a xs    = [ f a x | f <- fs, x <- xs ]
```

That is, at each step we have the choice of applying any of the operators from `fs` to any of the results so far produced in `xs`. Cognoscenti will recognise a *relational fold* here [2].

The above can be varied slightly according to the kind of lists one works with. For instance, for non-empty lists the counterpart of `fold` is

```
>fold1 f g [a]   = g a
>fold1 f g (a:x) = f a (fold1 f g x)
```

That is, the function `g` is applied to the last element of the list, and then we sum from right to left using the function `f`. Modifying the notion of sequential decision process accordingly, one obtains

```
>sdp1_spec r p fs l = listmin r . filter p . fold1 (choice fs) l'
>                     where l' a = [l a]
```

Note that this differs from the definition of `sdp_spec` in the base case only.

Below we shall also have a use for lists with at least two elements. Again `fold` can be modified for this particular data type:

```
>fold2 f g [a,b] = g a b
>fold2 f g (a:x) = f a (fold2 f g x)
```

In words, `g` is applied to the last two elements of the list, and then we sum (as before) from right to left, using the function `f`. The corresponding notion of sequential decision process is given by

```
>sdp2_spec r p fs l = listmin r . filter p . fold2 (choice fs) l'
>                     where l' a b = [l a b]
```

These variations already hint at the possibility that there exists a notion of decision process for other data types besides lists. We shall consider an example of this phenomenon at the end of this paper.

3 Preorders, Minning, Squeezing, Merging

We have not yet discussed the function `listmin` and the order relation `r` on which it operates. As might be expected, it is the notion of order and various ways of selecting minimum elements that are at the heart of our generic program for sequential decision processes. For this reason, we shall now look at a number of combinators for manipulating orders (to be precise, *preorders*) and for rejecting non-minimum elements. These combinators are the building blocks for the generic program presented in the next section.

Preorders. A *preorder* is a relation of type

```
>type Rel a = a -> a -> Bool
```

A preorder `r` is furthermore required to be *reflexive* (`r a a` for all `a`) as well as *transitive* (`r a b && r b c` implies `r a c`). A trivial example of a preorder is the relation

```
>top a b = True
```

New preorders can be created from functions whose target type carries an implicit order or an equality test:

```
>leq :: Ord b => (a -> b) -> Rel a
>leq f a b = f a <= f b

>geq :: Ord b => (a -> b) -> Rel a
>geq f a b = f a >= f b

>eq :: Eq b => (a -> b) -> Rel a
>eq f a b = f a == f b
```

For instance, `leq length` is the length preorder on lists. We shall be combining preorders built from the above primitives using two combinators, named meet (also called intersection) and sequential composition (also called lexicographical composition) respectively:

```
>meet :: Rel a -> Rel a -> Rel a
>meet r s a b = r a b && s a b

>seq :: Rel a -> Rel a -> Rel a
>seq r s a b = r a b && (not (r b a) || s a b)
```

It is easy to check that the meet of two preorders is again a preorder, and sequential composition also preserves preorders. To illustrate sequential composition, consider `r = seq (leq length) (geq sum)`. We have `r a b` precisely when

```
(length a < length b) || (length a == length b && sum x >= sum y)
```

The degenerate preorder `top` is the identity element of both `meet` and `seq`, in the sense that

```
   top ‘meet‘ r = r = r ‘meet‘ top
and
    top ‘seq‘ r = r = r ‘seq‘ top
```

Both `meet` and `seq` are associative.

listmin. Let us now take a closer look at the selection of a minimum element by the function `listmin r`. It is defined by

```
>listmin :: Rel a -> [a] -> a
>listmin r = fold1 m id
>            where m a b = a, if r a b
>                        = b, otherwise
```

Note that `listmin r` can only be applied to non-empty arguments. Also, the definition implicitly assumes that `r` is *connected*: `r a b` or `r b a` for all `a` and `b`. It is important to realise that `listmin r x` returns the first minimum element (reading from left to right) in `x`. In particular, `listmin top = head`.

squeeze. It is sometimes desirable to reject non-optimal elements while comparing with preorders that are not connected. This can be achieved, at least to some extent, by the function `squeeze r`. It removes an element from a list if one of its neighbours is lesser in the preorder `r`:

```
>squeeze :: Rel a -> [a] -> [a]
>squeeze r []      = []
>squeeze r [a]     = [a]
>squeeze r (a:b:x) = squeeze r (a:x),   if r a b
>                  = squeeze r (b:x),   if r b a
>                  = a:squeeze r (b:x), otherwise
```

When `r` is connected, and `x` is a non-empty list, we have

```
squeeze r x = [listmin r x]
```

In this sense `squeeze` is a proper generalisation of `listmin`.

mmerge. Another operator for manipulating lists and preorders is `mmerge`[3]. It takes a preorder `r`, two lists `x` and `y` that are ordered with respect to `r`, and it merges the two lists to produce another ordered list whose elements are precisely those of `x` and `y`. The definition of `mmerge` is

```
>mmerge :: Rel a -> [a] -> [a] -> [a]
>mmerge r [] y        = y
>mmerge r x []        = x
>mmerge r (a:x) (b:y) = a:mmerge r x (b:y), if r a b
>                     = b:mmerge r (a:x) y, otherwise
```

Note that `mmerge` has been defined so that `mmerge top x y = x++y`.

purge. The combination of `squeeze` and `merge` is called `purge`. It takes two ordered lists, merges them, and then squeezes the result:

```
>purge :: Rel a -> Rel a -> [a] -> [a] -> [a]
>purge r s x y = squeeze r (mmerge s x y)
```

Note that if `r` is connected, we have `purge r top x y = [listmin r (x++y)]`. Curiously, the discovery of the generic algorithm in the next section was held up for over a year because an earlier version of this work used a slightly less general definition of `purge`. As every functional programmer knows, the design of one's primitive combinators can make all the difference...

4 A Generic Program

Consider the sequential decision process

```
sdp_spec r p fs c = listmin r . filter p . fold g [c]
                    where g a xs = [ f a x | f <- fs, x <- xs ]
```

[3] The double `m` in `mmerge` is just there to avoid a name clash with the existing `merge` function in Gofer, which differs from our definition in that it does not take the preorder as an explicit argument.

Bellman's original insight was that, provided the principle of optimality is satisfied, this sequential decision process can be solved by dynamic programming [4]. Below we formalise the principle of optimality in three parts, and give a single program that captures the dynamic programming solution. This program is the novel contribution of the present paper. To apply the program, one has to invent a predicate `q` and preorders `s` and `t` that satisfy the following three requirements:

1. Let `q` be a predicate such that for each function `f` in `fs`

```
  p (f a x) =  q (f a x) && p x
```

2. Furthermore, let `s` be a preorder such that for each `f` in `fs`

```
  r x y   &&   s x y  &&  q (f a y)
implies
  r (f a x) (f a y)   &&   s (f a x) (f a y)  &&  q (f a x)
```

3. Finally, let `t` be a connected preorder such that for each `f` in `fs`

```
  t x y     implies     t (f a x) (f a y)
```

If furthermore we have that `p c` holds, the function `sdp r s t q fs c x` defined below computes a solution to `sdp_spec r p fs c x`, for all lists `x`:

```
>sdp r s t q fs c = listmin r . fold (step (meet s r) t q fs) [c]
>step v t q fs a xs = fold1 (purge v t) id
>                      [filter q [f a x | x <- xs] | f <- fs ]
```

It is worthwhile to consider the conditions of this result in some detail. The first condition, in conjunction with the requirement that `p c` holds, says that `filter p` can be *promoted* into the generator part of the sequential decision process. Readers who are familiar with Bird's theory of lists [5] may recognise that when `f` is the `cons` operator (:), this is another way of expressing that `p` is *suffix-closed*. Similar conditions can be found in other work involving *filter promotion*.

The second property is a so-called *dominance* criterion. Informally, it says that when `x` is better than `y` (that is `(r x y) && (s x y)`), then the extension of `x` is better than the extension of `y`; furthermore, the extension of `x` is a feasible solution if the extension of `y` is. The exploitation of dominance criteria is a well-known strategy for speeding up dynamic programming algorithms [25].

The third property states that each `f` in `fs` is monotonic on the connected preorder `t`. This condition gives little guidance for choosing a preorder `t`, but in most cases `t` is a simple combination of `r` and `s`.

It would be nice if these three properties amounted to a well-understood mathematical structure, in analogy with closed semi-rings in the algebraic path problem. Unfortunately, I have been unable to identify such a connection with well-established mathematics, perhaps because the conditions are phrased as implications rather than as algebraic properties.

The program `sdp` generates a list of candidates which is ordered according to the preorder `t`. Because each `f` is monotonic on `t`, we can maintain this order at each stage by merging ordered lists. The result is then squeezed using the dominance criterion `meet r s`: this is where the improvement over the original formulation of `sdp_spec` occurs. If the squeezing is successful, only a handful of intermediate solutions will be kept at each stage. Finally, an optimum solution is selected using `listmin r`.

Some readers may wonder why the conclusion in the above result is not phrased as

```
sdp_spec r p fs c x  =  sdp r s t q fs c x
```

instead using the awkward phrase '... `sdp` computes a solution to `sdp_spec` ...'. The reason is that the above equation is false. The new program `sdp`, when applied to `x`, will yield a result that is equivalent to that of `sdp_spec`, in the sense that it is a minimum element of `filter p (fold (choice fs) [c] x)`, but it need not be the same minimum element as the one returned by `listmin r`. It is here that we pay the price for discussing these ideas within a purely functional framework. A truly satisfactory solution requires a programming notation based on relations (or predicates) rather than functions. One such notation is expounded in [8], and that book contains detailed proofs of the claims in this paper.

Again we remark that with minor changes, the above goes through for non-empty lists. Consider the specification

```
sdp1_spec r p fs l = listmin r . filter p . fold1 (choice fs) l'
                     where l' a = [l a]
```

The only change to the above result concerns the requirement that `p c` holds. For non-empty lists, the corresponding condition is

Suppose that for each `a`*, we have* `p (l a)`.

Conditions (1-3) (which are concerned with the newly invented functions `q`, `s` and `t`) are the same as for possibly empty lists. The generic program for non-empty lists is

```
>sdp1 r s t q fs l = listmin r . fold1 (step (meet s r) t q fs) l'
>                    where l' a = [l a]
```

Analogously, we have a generic program for lists that have at least two elements, namely

```
>sdp2 r s t q fs l = listmin r . fold2 (step (meet s r) t q fs) l'
>                    where l' a b = [l a b]
```

and here the requirement that `p c` holds is replaced by

> *Suppose that for each* `a` *and* `b`*, we have* `p (l a b)`*.*

Conditions (1-3) are the same for all three kinds of list.

5 Examples

Below we apply the generic algorithm to three different programming exercises. In each exercise, most of the effort goes into phrasing the problem as a sequential decision process. Once the problem is in the right form, inventing the additional predicate `q` and the preorders `s` and `t` is easy. Three exercises is all that space allows in this paper; [8] presents seven more applications, and that number is still growing.

5.1 Knapsack

In the 0/1 knapsack problem, we are given a sequence `x` of items and a non-negative capacity `c`. Each item is a pair of non-negative numbers, called the value and the weight of that item. The objective is to compute a subsequence of `x` whose total weight does not exceed `c`, and whose total value is as large as possible. The total weight of a sequence is defined as the sum of the weights of the items it contains, and the total value of a sequence is the sum of the individual values.

Formal specification. An item is a pair of non-negative numbers

```
>type Item = (Int, Int)
```

whose first component signifies the value of the item, and the second component the weight:

```
>v,w :: Item -> Int
>v (a,b) = a
>w (a,b) = b
```

Our aim is to give an efficient implementation of

```
>kp_spec :: Int -> [Item] -> [Item]
>kp_spec c    =   listmin rval . filter (ok c) . subs
```

The function `subs` is the generator: it generates all subsequences of its argument, in some order. One can define `subs` as

```
>subs  =  fold (choice [(:),rhs]) [[]]
>         where rhs a x = x
```

This definition reflects the observation that, at each stage, we have the choice of either including `a` in a subsequence by applying `(:)`, or excluding it by applying `rhs`. The predicate `ok c` checks whether the total weight of a subsequence does not exceed the capacity `c`. The function `weight` returns the total weight of a sequence, so the definition of `ok` is

```
>ok :: Int -> [Item] -> Bool
>ok c x  =  weight x <= c

>weight :: [Item] -> Int
>weight  =  sum . map w
```

Finally, we are interested in a subsequence whose value is as large as possible. Taking the maximum in the value preorder means taking the minimum in the *reverse* value preorder, which is given by

```
>rval :: Rel [Item]
>rval = geq value

>value :: [Item] -> Int
>value = sum . map v
```

Here `value` is the function that returns the total value of a sequence. Summarising, the knapsack problem can be formulated as a sequential decision process:

```
kp_spec c   =   sdp rval (ok c) [(:),rhs] []
```

In order to apply the generic algorithm, we need to determine a predicate `q`, and preorders `s` and `t` that satisfy the three applicability conditions.

Checking the conditions. First, note that `ok c []` holds because `c` is non-negative. We shall take `q = (ok c)`. This does satisfy the promotion condition on `q`

```
   ok c (a:x)   implies   ok c x
```

This implication is valid because weights are non-negative. For the dominance criterion, observe that

```
  value x >= value y  &&  weight x <= weight y
  &&  ok c (a:y)
implies
  value (a:x) >= value (a:y) && weight (a:x) <= weight (a:y)
  &&  ok c (a:x)
```

It follows that we can take `s = leq weight`. Finally, observe that both `(:)` and `rhs` are monotonic on `rval`, so we can take `t = rval = geq value`. We conclude that the generic algorithm is applicable.

Program. There is still one minor source of inefficiency, namely the repeated computation of the total value and weight of a subsequence. The solution is to store the total value and weight of a sequence together with the sequence itself, and maintain them incrementally. The resulting program is displayed below:

```
>kp c =
> the.sdp (geq value) (leq weight) (geq value) ok [cons,rhs] empty
> where value (x,v,w) = v
>       weight (x,v,w) = w
>       the (x,v,w) = x
>       ok x = weight x <= c
>       cons (a,b) (x,v,w) = ((a,b):x,a+v,b+w)
>       rhs (a,b) (x,v,w) = (x,v,w)
>       empty = ([],0,0)
```

This is the standard pseudo-polynomial time $\mathcal{O}(cn)$ algorithm for the knapsack problem. Unlike the program found in many textbooks, however, it does not depend on the values and weights being integers: it is equally applicable to floating point numbers. Starting from the above algorithm as a *specification*, my research student Michael Zhu Ning has developed a novel, highly optimised algorithm that outperforms all other methods in practice [22]. The main improvement in his work comes from a very simple change to the generic program: because we have chosen `t = r = geq value`, the maximum value solution appears at the head of the list produced by `fold` and so — for this particular application — `listmin r` in the generic program can be replaced by `head`. Lazy evaluation then leads to a substantial time saving.

5.2 Bitonic Tours

Our next programming exercise is taken from [10]:

> The *Euclidean travelling salesman* problem is the problem of determining a shortest closed tour that connects a given set of n points in the plane. The left-hand side of Figure 1 shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.
> J.L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. The right-hand side of Figure 1 shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
> Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. (*Hint:* Scan right to left, maintaining optimal possibilities for the two components of the tour.)

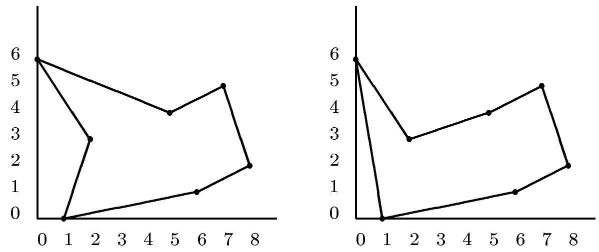

Fig. 1. An optimal tour and an optimal *bitonic* tour.

Representation of tours. We represent a bitonic tour by a pair of lists `(u,v)`. The components `u` and `v` are the outward and return journeys of the tour. Note that the roles of `u` and `v` are symmetrical: if `(u,v)` is a bitonic tour, so is `(v,u)`, and these tours are essentially the same. To prevent the existence of different representations of the same tour, we stipulate that the first element of `u` is the leftmost point of the tour. For instance, the pair `(u,v)` represents the optimal bitonic tour in Figure 1:

```
u  =  [(0,6),(1,0),(6,1)]
v  =  [(2,3),(5,4),(7,5),(8,2)]
```

Note that `u` and `v` are chosen to be disjoint non-empty lists.

Representation of input. It does not make sense to talk about a bitonic tour of one point because such a tour cannot be split into two disjoint unidirectional components, so we assume that there are at least two points. Furthermore, the input list is assumed to be presented in increasing order of x-coordinate. Because the input list is a list of length at least two, we shall be using the `fold2` operator, and the sequential decision process skeleton `sdp2_spec`.

Constructing tours. The operator `wrap` takes two input points, and turns them into the only possible bitonic tour, namely

```
>wrap a b      = ([a],[b])
```

The result of `wrap` is a correct representation of a bitonic tour provided that the x-coordinate of `a` is strictly smaller than the x-coordinate of `b`.

The binary operator `addl` takes an element and a tour, and adds the element to the left-hand component of the tour:

```
>addl a (u,v)  =  (a:u,v)
```

This yields a correct representation of a tour only if `a` has an x-coordinate strictly smaller than the elements of `u` and `v`. If this condition is satisfied, there is another way of adding `a` to the tour, namely by adding `a` to the other component of the tour. As we wish to keep the leftmost point of the tour at the head of the first list, this interchanges `u` and `v`:

```
>addr a (u,v)  =  (a:v,u)
```

Generating all tours. Given a list of points in increasing order of x-coordinate, all tours are returned by

```
>tours  = fold2 (choice [addl,addr]) wrap'
>         where wrap' a b = [wrap a b]
```

That is, the sequence is scanned from right to left, each point being appended with either `addl` or `addr`.

Length of a tour. Let path_len be the function that returns the length of a path:

```
>path_len x = sum (zipWith dist x (tail x))
```

Here `dist a b` returns the distance between two points `a` and `b`. The Euclidean distance measure is coded as

```
>dist (a0,a1) (b0,b1) = sqrt ((a0 - b0)^2 + (a1 - b1)^2)
```

The total length of a tour can then be defined by

```
>len  (u,v)  =  path_len u + path_len v +
>               dist (head u) (head v) + dist (last u) (last v)
```

So the formal specification of Bentley's exercise reads

```
>mt_spec  =  listmin (leq len) . tours
```

This specification is a sequential decision process:

```
 mt2 =   sdp2_spec (leq len) (const True) [addl,addr] wrap
```

where `const True` is the predicate that is always true.

Checking the conditions. Again we need to find a predicate and two preorders to satisfy the applicability conditions of the generic algorithm. In this case, the choice of predicate is trivial, as in the original sequential decision process, `p` is the constant function returning `True`; we take `q` to be `(const True)` as well.

For the dominance criterion, observe that

```
  len (u,v) <= len (u',v')  &&  head u == head u'
  && head v == head v'
implies
  len (a:u,v) <= len (a:u',v') && head (a:u) == head (a:u')
  && head v == head v'
```

which gives the dominance criterion for `addl`; a similar property holds for `addr`. We can therefore take `s = eq heads`, where `heads(u,v) = (head u,head v)`.

Finally, to satisfy the last condition we need a connected preorder `t` on which `addl` and `addr` are monotonic. There is no obvious choice for such a preorder dictated by the problem, so we might as well set `t = top`. Trivially, any total function is monotonic on the degenerate preorder `top`.

Program. In analogy with the knapsack example, it makes sense to maintain the length of a tour incrementally, so that it does not have to be recomputed at each comparison. A tour `(u,v)` will be represented by the three-tuple

```
  (u, v, path_len u + path_len v + dist (last u) (last v))
```

The resulting program is shown below:

```
>mt =
> the.sdp2 (leq len) (eq heads) top (const True) [addl,addr] wrap
> where len (a:x,b:y,n) = dist a b + n
>       heads (a:x,b:y,n) = (a,b)
>       the (a:x,b:y,n) = (a:x,b:y)
>       wrap a b = ([a],[b],dist a b)
>       addl a (b:x,c:y,n) = (a:b:x,c:y,n+dist a b)
>       addr a (b:x,c:y,n) = (a:c:y,b:x,n+dist a c)
```

This program has the required $\mathcal{O}(n^2)$ time complexity.

5.3 Bus stops

Many ideas in this paper have been gleaned from the operations research literature, which has been much more concerned with exposing connections between algorithms than the computing literature. It seems fitting, therefore, to illustrate our generic algorithm with a typical operations research problem. The following exercise occurs in [11]:

> A long one-way street consists of m blocks of equal length. A bus runs "uptown" from one end of the street to the other. A fixed number n of bus stops are to be located so as to minimise the total distance walked by the population. Assume that each person taking an uptown bus trip walks to the nearest bus stop, gets on the bus, rides, gets off at the stop nearest his destination, and walks the rest of the way. During the day, exactly B_j people from block j start uptown bus trips, and C_j complete uptown bus trips at block j. Write a program that finds an optimal location of bus stops.

Representing the input. We shall represent the input as a list

```
[B1 + C1, B2 + C2, ... , Bm + Cm]
```

Here each item `Bj + Cj` represents the number of people passing through block `j`. If `B` and `C` are originally given as separate lists, one may convert them into the above representation by evaluating

```
zipWith (+) [B1,B2,...,Bm] [C1,C2,...,Cm]
```

Representing bus stop arrangements. The original formulation of the problem is not quite clear about this, but we shall assume that bus stops are placed in front of blocks, and not at block boundaries. Furthermore, if a bus stop is placed in front of block `j`, people passing through `j` do not have to walk at all in order to take the bus. An example arrangement of bus stops is shown below. The first row are the numbers of blocks, and the second row shows how many people pass through each block. In the last row, a tick ($\surd$) indicates that a bus stop has been placed at a certain block, and the arrows show the direction by which people walk to bus stops.

1	2	3	4	5	6	7	8	9	10	11
50	12	25	33	15	19	22	58	12	42	30
$\rightarrow$	$\rightarrow$	$\surd$	$\leftarrow$	$\leftarrow$	$\rightarrow$	$\surd$	$\leftarrow$	$\rightarrow$	$\surd$	$\leftarrow$

For instance, people passing through block 5 have to walk two blocks to the left in order to take the bus at block 3. It follows that the total walking distance required by the above arrangement is

$$2*50+1*12+0*25+1*33+2*15+1*19+0*22$$
$$+1*58+1*12+0*42+1*30$$

We shall represent an arrangement of bus stops for a sequence `x` of blocks as a partition of `x++[0]`, with a bus stop at the end of each partition component except the last. The role of `0` at the end is to allow the placement of a bus stop at the very last block in the street. The above example would thus be represented as

```
[[50,12,25],[33,15,19,22],[58,12,42],[30,0]]
```

Having the decided on this representation, we have the following specification:

```
>buses_spec n x = (listmin(leq walk) . filter(atmost n) . parts)
>                  (x++[0])
```

where the function `parts` generates all partitions of its argument, the filter removes partitions that have more than `n+1` components (`n` bus stops), and the selector picks a partition that realises the minimum walking distance. The definition of `atmost` is easy:

```
>atmost n xs  =  length xs <= n+1
```

The definitions of `parts` and `walk` are elaborated below.

Total walking distance. To give a formal definition of the total walking distance, we first introduce the *convoluted sum* `cnv x` of a list of numbers `x`. Informally, it is given by

```
 cnv [a1, a2, ... , an]   =   1*a1 + 2*a2 + ... + n*an
```

and that translates to the Gofer definition

```
>cnv x = sum (zipWith (*) [1..] x)
```

The total walking distance is then defined

```
>walk [x]      =  cnv x
>walk (x:xsy)  =  cnv (reverse (init x)) + sum (map wd xs) + cnv y
>                 where xs = init xsy
>                       y  = last xsy
```

The walking distance of a singleton partition is the convoluted sum of its single component: everybody has to walk to the very end of the street, in fact to a bus stop at the imaginary block that has `0` people passing through it. In the first partition component `x` of a non-singleton partition, people have to walk to the last block in that component. The function `init x` returns all of `x` except the last element. The contribution to the total walking distance is therefore `cnv (reverse (init x))`. The cost of the last component `y` is similarly computed, except that we do not have to strip off the last element: it is zero anyway. The walking distance contributed by a non-extremal partition component is defined by

```
>wd x = cnv y + cnv (reverse z)
>       where (y,z) = split (init x)
```

The function `split` splits its argument `x` into two approximately equal halves. Using the standard function `splitAt`, which splits a list at a specified position, it can be defined by

```
>split x = splitAt (length x / 2) x
```

Generating partitions. We know that the input list for `parts` is non-empty (its last element is `0`), so we aim for a definition of `parts` in terms of `fold1`. The function `wwrap` takes an element and creates a singleton partition:

```
>wwrap a   =   [[a]]
```

Given an element `a` and a partition `xs`, we can either add `a` as a new partition component, or glue `a` to the first component of `xs`:

```
>new a xs    =    [a]:xs
>glue a (x:xs)  =  (a:x):xs
```

This leads us to the following definition of *parts*:

```
>parts  =  fold1 (choice [new,glue]) wwrap'
>          where wwrap' a = [wwrap a]
```

In summary, we have specified the bus stop problem as a sequential decision process

```
 buses_spec  n x  =  sdp (leq walk) (atmost n) [new,glue] (x++[0])
```

Checking the conditions. In analogy with the knapsack example, we take the predicate `q` to be `atmost n` itself. This satisfies the requirements because we have `atmost n (wwrap a)` for any `a`, and

```
    (atmost n ([a]:xs))   implies  (atmost n xs)
and
    (atmost n ((a:x):xs)) implies  (atmost n (x:xs))
```

For the dominance criterion, we take our cue from the bitonic tours example, and observe that the following property holds for `new`:

```
  walk xs <= walk ys && length xs <= length ys
  && head xs == head ys && atmost n ([a]:ys)
implies
  walk([a]:xs) <= walk([a]:ys) && length([a]:xs) <= length([a]:ys)
  && head([a]:xs) == head([a]:ys) && atmost n ([a]:xs)
```

The analogous property for `glue` is also easily verified. It follows that we can take `s = meet (leq length) (eq head)`. In fact, since we are comparing partitions of the same sequence, we can actually use the computationally more efficient definition `s = meet (leq length) (eq (length . head))`.

It remains to find a connected preorder `t` on which both `new` and `glue` are monotonic. For efficiency reasons it is best to take a preorder that is as restrictive as possible, and a heuristic that often succeeds is to take a relaxation of `s`. In this case, that heuristic suggests a sequential composition of `leq length` and `leq (length.head)`:

```
   t = seq (leq length) (leq (length . head))
```

Program. Each partition `xs` will be represented as a 5-tuple

```
  (xs, walk(tail xs), cnv(reverse(init(head xs))),
   length xs, length (head xs))
```

This facilitates constant time comparison in all three preorders defined above. The resulting program, which is displayed below, has time complexity $\mathcal{O}(nm^2)$, where n is the number of bus stops and m the number of blocks in the street:

```
>buses n x =
>   the (sdp1 r s t atmost [new,glue] wwrap (x++[0]))
>   where r = leq walk
>         s = leq length `meet` eq length_head
>         t = leq length `seq` leq length_head
>         atmost xs = length xs <= n+1
>         the (xs,s,c,n,m) = xs
>         walk (xs,s,c,n,m) = s+c
>         length (xs,s,c,n,m) = n
>         length_head (xs,s,c,n,m) = m
>         wwrap a = ([[a]],0,0,1,1)
```

```
>          new a (xs,s,c,1,m) = ([a]:xs,0,cnv(head xs),2,1)
>          new a (xs,s,c,n,m) = ([a]:xs,0,wd(head xs)+c,n+1,1)
>          glue a (x:xs,s,c,n,m)= ((a:x):xs,m*a+s,c,n,m+1)
```

Note that we have not made essential use of the fact that the blocks are all of equal size; it would be quite easy to adjust the program to cope with different sizes.

6 Why polytypism matters

Some readers may feel that I have cheated in the above examples: after all, I am not using a single program but three different programs which operate on possibly empty lists, non-empty lists, and lists with at least two elements. Surely it would have been possible to code the problems in such a way that only one kind of lists and only one program sufficed?

While this is probably true of the specific examples shown above, I would rather say that I have presented a single program that is parameterised by the type on which it operates. Such a program is said to be *polytypic*. To reinforce the point, let me show you one more instance of the generic program for sequential decision processes, this time applied to trees rather than lists. The following problem is the topic of [18]:

> Let G be an acyclic directed graph with weights and values assigned to its vertices. In the partially ordered knapsack problem we wish to find a maximum-valued subset of vertices whose total weight does not exceed a given knapsack capacity, and which contains every predecessor of a vertex if it contains the vertex itself. We consider the special case where G is an out-tree. Even though this special case is still NP-complete, we observe how dynamic programming techniques can be used to construct pseudopolynomial time optimisation algorithms.

The data structure described as an *out-tree* in the above quotation can be defined in a functional programming style as

```
>data Tree a = Node a [Tree a] | Null
```

where every node has at least one descendant. A *pruning* of a tree `t` is a copy of `t` where some of its subtrees have been replaced by `Null`. Given a tree `t :: Tree (Int,Int)` of (value,weight) pairs, we can define the total value of `t` as the sum of the values it contains, and similarly the total weight is defined as the sum of the individual weights. The *tree knapsack* problem is to compute a pruning whose total weight does not exceed a given capacity `c`, and whose value is as large as possible. This problem has sinister applications in cutting a hierarchical workforce; here it is merely a programming challenge.

Clearly we have a decision process here, albeit not a sequential one: at every node we have the choice between including that node or leaving it out. Indeed, one can define a notion of *fold* over trees, and replay the discussion that led us to a generic list-based algorithm, to get a generic tree-based algorithm. These algorithms will be 'essentially the same'; in fact, if we had a programming notation that allowed the parameterisation of programs by type structure, both programs would be instances of the same *polytypic* program. Theories and notations for reasoning about polytypic programs are under construction, but it is beyond the scope of this paper to go into details. The specific instance of that polytypic program, specialised for the above programming exercise, is shown below:

```
>foldTree f c Null = c
>foldTree f c (Node a x) = f a (map (foldTree f c) x)

>treekp c =
>  the . listmin (geq value) . foldTree f [nul]
>  where f a tss = fold1 step (filter ok . map (add a)) tss++[nul]
>        step ts ss = fold1 (purge dom (geq value)) id
>                     [filter ok [glue t s | t<- ts] | s <- ss]
>        the (x,v,w) = x
>        weight (x,v,w) = w
>        value (x,v,w) = v
>        nul = (Null,0,0)
>        ok x = weight x <= c
>        add (a,b) (t,v,w) = (Node (a,b) [t],v+a,w+b)
>        glue (t,v,w) (Node a ts,vs,ws) = (Node a(t:ts),v+vs,w+ws)
>        dom = geq value `meet` leq weight
```

Hopefully this example gives some indication of the possibilities for writing generic, type independent programs. Recently a number of papers have suggested that Gofer's *constructor classes* are the appropriate medium for polytypic programming [20, 21]. The examples in those papers certainly show the great flexibility of constructor classes, and they present an interesting experiment in pushing the limits of genericity in programming. I would, however, like to add a word of caution. Constructor classes are essentially a means of introducing *ad hoc* polymorphism into the Hindley-Milner type system. The polytypic programs that I have in mind are — if we ignore exponential types — parametric in the type constructors. To find a type system that reflects such parametricity would, in my view, be a major step towards a truly generic style of programming. Jay's exploration of *shapely types* may be a step in the right direction [16].

7 Related work

The view of sequential decision processes put forward here is, by the standards of operations research, rather restrictive: many experts in that community would classify all applications of dynamic programming as sequential decision processes. It then becomes difficult to give a precise algorithmic definition of the concept. Several semi-algorithmic definitions have however been put forward, and especially the work of Helman and Rosenthal [14] has been a major influence on the results reported here. Our view of sequential decision processes is quite close to that in the seminal paper by Karp and Held [19].

In the algorithm design community, a slightly less general class of problems has received considerable attention under the name *least weight subsequence* problems [15]. In terms of the terminology introduced here, all of these problems are concerned with list partitions. Most recent work in this area has been concerned with improving the time complexity of naive dynamic programming solutions, by exploiting special properties of the cost function [12].

In programming methodology, our work is very much akin to that of Smith and Lowry [23, 24]. Smith's notion of *problem reduction generators* is quite similar to the generic algorithm presented here, but his results are more concerned with similarities in the derivation process, and not with a single generic program.

There has recently been a surge of interest in polytypic programming. Initially most of this work was concerned with generalising combinators such as $fold$ and zip to arbitrary data types [3]. Starting with [9], more algorithmic problems have been considered in a polytypic setting. The most spectacular example so far is Jeuring's polytypic pattern matching algorithm [17].

Acknowledgements

Much of the work reported here was done in collaboration with Richard Bird, and I am grateful to him for his support, and his insistence that there is always a better, more efficient, and more elegant solution. Sharon Curtis suggested an improvement over my original solution to the Bitonic Tours problem. Roland Backhouse, Richard Bird, Jeroen Fokker and Doaitse Swierstra suggested substantial improvements over a first draft.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, 1974.
2. R. C. Backhouse, P. De Bruin, G. Malcolm, T. S. Voermans, and J. C. S. P. Van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers B.V., 1991.
3. Roland Backhouse, Henk Doornbos, and Paul Hoogendijk. Commuting relators. Technical Report. Available by anonymous ftp from `ftp.win.tue.nl`, directory `pub/math.prog.construction.`, 1992.
4. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
5. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer–Verlag, 1987.

6. R. S. Bird and O. de Moor. List partitions. *Formal Aspects of Computing*, 5(1):61–78, 1993.
7. R. S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction*, volume 669 of *Springer Lecture Notes in Computer Science*, 1993.
8. R. S. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, To appear, 1996.
9. R. S. Bird, P. Hoogendijk, and O. de Moor. Generic programming with relations and functors. Submitted to Journal of Functional Programming, 1993.
10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT electrical engineering and computer science series. MIT Press, Cambridge, Mass. ; London / McGraw-Hill, New York, 1990.
11. E. V. Denardo. *Dynamic Programming — Models and Applications*. Prentice–Hall, 1982.
12. D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming II: Convex and concave cost functions. *Journal of the ACM*, 39:546–567, 1992.
13. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
14. P. Helman and A. Rosenthal. A comprehensive model of dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):319–334, 1985.
15. D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.
16. B. Jay. Matrices, monads and the Fast Fourier transform. Available by anonymous ftp from `ftp.socs.uts.edu.au` directory `pub/jay`, August 1993.
17. J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture, FPCA '91*, 1995.
18. D. S. Johnson and K. A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8(1):1–15, 1983.
19. R.M. Karp and M. Held. Finite–state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.
20. E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture, FPCA '91*, 1995.
21. E. Meijer and M. Jones. Gofer goes bananas. Unpublished manuscript, 1994.
22. M. Z. Ning. A lazy algorithm for the 0/1 knapsack problem. Technical Report (to appear), Oxford University Computing Laboratory, 1995.
23. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305–321, 1990.
24. D.R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *Proc. of the IFIP TC2 Working Conference on Constructing Programs from Specifications*. North–Holland, 1991.
25. F. F. Yao. Speed–up in dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 3(4):532–540, 1982.

This article was processed using the LaTeX macro package with LLNCS style