Optimisation problems in logic programming: an algebraic approach

Silvija Seres and Shin-Cheng Mu

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.
{ss,scm}@comlab.ox.ac.uk

Abstract. Declarative programming, with its mathematical underpinning, was aimed to simplify rigorous reasoning about programs. For functional programs, an algebraic calculus of relations has previously been applied to optimisation problems to derive efficient greedy or dynamic programs from the corresponding inefficient but obviously correct specifications. Here we argue that this approach is natural also in the logic programming setting.

1 Introduction

Dynamic programming[1] is the name for a general strategy used in algorithms that organises the computation so that subproblems are evaluated once instead of many times; traditionally this is done by combining a recurrence equation with tabling or memoing. As applied to combinatorial optimisation problems, dynamic programming was first popularised by Bellman in [9], where he introduced the *Principle of Optimality* which states that an optimal solution is composed of optimal solutions to subproblems. This is the essential (though only sufficient and not necessary) condition for the dynamic programming technique to be applicable. Greedy algorithms also suppose that the principle of optimality holds, but in addition they exploit a greedy condition which guarantees that on basis of some local information only the best subproblem needs to be computed. Some problems fall in between these two extremes of pursuing all or only one of the recursive decompositions. For these, the principle of optimality is strengthened with an additional condition which is used to narrow down the choice to a subset of decompositions which might eventually lead to an optimal solution.

The problems which satisfy the principle of optimality can be divided into two categories. Some are naturally solved in a top-down way, through a decomposition of a recursive data-type and a consequent combination of the partial answers. Others are better solved in a bottom-up way, where a recursive datatype which contains the answers is being composed from some seed. Provided that the conditions mentioned above hold, greedy algorithms can be derived for both of these classes of problems, but dynamic programming, as treated here,

[1] The nomenclature in the dynamic programming literature is somewhat inconsistent, so in what follows we have chosen to follow a neutral source [4].

only applies to problems with bottom-up, compositional specifications. For the problems which do not satisfy the greedy condition yet require a decompositional solution, we derive a third alternative: thinning algorithms.

From a programmers point of view, two main questions arise from this description of algorithms for optimisation problems. One is how to *formalise* the aforementioned conditions for each of the three strategies, in such a way that programmers can easily analyse the problem and identify the appropriate algorithm design strategy. The second is how can the programmers use this information to *derive* the correct and efficient algorithm within the given strategy. In the traditional, tabulating, approach to dynamic programming, there is no general answer to these two questions.

These two questions have recently been addressed by the relational algebraic approach to optimisation problems, and applied in the framework of functional programming. Helman [8] was the first to separate the ideas of problem structure and computation, and his ideas have been generalised by de Moor and Bird [6, 1] and later by Curtis [5].

Here, we apply these results to the setting of logic programs by showing how this approach can be used to answer these questions for three standard examples in dynamic programming: the string-edit, the minimum lateness and the 1/0 knapsack problem. In section 2 we present the general framework, and in section 3 we formulate in terms of higher-order logic programs the three central theorems for the classification of optimisation problems and for the derivation of the respective algorithms. In sections 4, 5 and 6 we show how the theorems can be applied to the problems mentioned above, and in section 7 we conclude. The full code for the three examples can be found in the appendices.

2 Algebraic approach to dynamic programming

The goal behind the algebraic approach to optimisation problems is to provide a general tool for both the analysis and the derivation of the appropriate efficient algorithm. The starting point for both these tasks is a uniform specification of optimisation problems. In this section, we first present this specification, and then we prepare the ground for the analysis and the derivation that will be presented in the following section. In order to make the theorems as general as possible, we use higher-order predicates, defined in terms of `call/n`, which applies a given predicate to the rest of the argument list. Also, for the sake of simplicity, when we need to return a collection of answers to a given predicate rather than the individual answers, we use the standard predicate `bagof/3`.

Optimisation problems ask for the best solution among all the solutions to a particular problem. This specification is most naturally formulated as a composition of two relations: first generate all the possible solutions to the problem, and then test this collection of answers to find the best answers among them. Which answers are "best" depends on the particular ordering of solutions which will inevitably vary from problem to problem, such as maximising the value, minimising the delay, etc. To abstract from such particulars, we write `r` to denote the ordering of the solutions, and use a higher-order predicate `best(r,Bag,Out)` that is true if `Out` contains the `r`-optimal solutions in a collection `Bag`. Given a predicate `bagof(X,p(X),Bag)` that collects in `Bag` all the `X`'s satisfying `p(X)`, the specification can be implemented as:

```
optimal(In,Out) :- bagof(X,solution(In,X),Bag),
                   best(r,Bag,Out).
```

The relation `r` must be a preorder, i.e. a reflexive and transitive relation. However, some elements of `Bag` may not be related by it, so it need not be a connected preorder. We say that a solution `A` is better then `B` with respect to `r` if and only if the predicate `r(A,B)` holds, and the predicate `best(r,X,C)` holds when `r` happens to be connected and `C` matches *all* the optimal elements in `X` with respect to `r`:

```
best(r,[A],A).
best(r,[A,B|X],C) :- call(r,A,B), best(r,[A|X],C).
best(r,[A,B|X],C) :- call(r,B,A), best(r,[B|X],C).
```

Further, we observe that most of the common optimisation problems can be formulated in terms of some initial data-types, such as lists or trees, and that the computation of individual solutions performed by `solution(In,X)` can be expressed in terms of higher-order relations over these initial data-types, such as `fold` or `unfold`. We simplify the presentation by expressing all the results in terms of lists, which suffices for the purposes of our three examples, although the more general setting of [1] includes any initial data-types.

The `fold` relation for lists collapses an input list to a value according to a given relation `p`, while `unfold` constructs from an input value a list according to the relation `p`. They are converses of each other; declaratively, we do not need two separate definitions as we do in functional programming languages, as we could simply reverse the roles of two arguments in to produce a list rather than consume a list. Operationally, however, we need to reverse the order of the two premises so that Prolog and other systems with an unfair selection rule can solve the subgoals in the correct order. Still, in the relational setting of logic programming their definitions are almost identical, the only difference being ordering of the two literals in the second clause:

```
fold(_,E,[],E).
fold(p,E,[A1|X],A) :- fold(p,E,X,A2), call(p,(A1,A2),A).

unfold(_,E,[],E).
unfold(p,E,[A1|X],A) :- call(p,(A,A2),A), unfold(p,E,X,A2).
```

In case of `fold` it is expected that the input parameter will be the third and the output will be the fourth argument, while for `unfold` their positions will be swapped. For example, $\texttt{fold}(\texttt{add}, 0, [2, 7, 8], \texttt{X})$ will be true for $\texttt{X} = \texttt{17}$, while $\texttt{unfold}(\texttt{add}, 0, \texttt{Y}, 17)$ will return $\texttt{Y} = [2, 7, 8]$ as one of its many answers, for an appropriate definition of the predicate `add`.

As for an application of `fold` in a computation of a particular `solution` in `bagof`, it is the natural predicate to use whenever we are given a list and need to decompose it, say, in order to find an optimal way to select some elements from it. For example, in the knapsack problem, if `In` is a list of items and the task is to select most valuable sublist of it within a given knapsack capacity, a solution `Out` can be computed by a `fold` which uses the term `emptysack` as the initial input to the computation and the relation `consumeone` to consume the consequent elements of the `In` list:

```
solution(In,Out) :- fold(consumeone,emptysack,In,Out).
```

Alternatively, whenever we need to compose the solutions from some seeds, as for example in the string edit problem where we are looking at sequences of editing operations between two strings, we will use `unfold` to implement the predicate `solution`. The solutions can be computed by an `unfold` which uses the term `emptyedit` as the initial edit empty sequence and the relation `addone` to produce the consequent edit instructions in the list `Out`. Predicate `addone` builds the list `Out` from its previous result and the seed `In`, which simply contains the two strings to be edited:

```
solution(In,Out) :- unfold(addone,emptyedit,Out,In).
```

Finally, the notion of refinement in the next section is as follows: the derived predicate `fast` *refines* the specification `spec` if for all inputs `In`, we have:

$$\mathtt{fast(In,Out)} \Rightarrow \mathtt{spec(In,Out)}$$

With these preliminaries out of the way, we are now ready to express the three central optimisational theorems regarding the applicability and the derivation of dynamic, greedy and thinning algorithms. The proofs of these theorems are omitted in this paper for lack of space, but can be found in a more general form in [1]. These proofs are based on equational reasoning about the algebraic properties of the relations involved, which is why we refer to this as an "algebraic" approach. In this style of program calculation, theorems about higher-order functions like `fold` play a central role, with the *fusion* theorem for `fold` being probably the single most important theorem of the calculus and the pivotal point of algebraic program transformation.

3 The three theorems

Before we start introducing the main theorems of this paper, we need to introduce the notion of *monotonicity* of predicates. The predicate $\mathtt{p((A,X),NewX)}$ constructs a solution `NewX` by incrementing the partial solution `X` by `A`. We say that `p` is monotonic, or order preserving, on a preorder `r` if, for any arguments $\mathtt{X_i}$ and $\mathtt{X_j}$, if $\mathtt{X_i}$ is better than $\mathtt{X_j}$ with respect to `r`, no matter how `p` extends the inferior solution to some $\mathtt{NewX_j}$, we can always find at least one way to extend by `p` the superior solution so that the resulting $\mathtt{NewX_i}$ is better than $\mathtt{NewX_j}$. If this is the case, we know that it is safe to throw the inferior solutions away and only extend the best ones. Formally:

$$\begin{aligned} & \mathtt{r}(\mathtt{X_i}, \mathtt{X_j}) \wedge \mathtt{p}((\mathtt{A}, \mathtt{X_j}), \mathtt{NewX_j}) \Rightarrow \\ & \qquad \exists \mathtt{NewX_i}.\ \mathtt{p}((\mathtt{A}, \mathtt{X_i}), \mathtt{NewX_i}) \wedge \mathtt{r}(\mathtt{NewX_i}, \mathtt{NewX_j}). \end{aligned} \tag{1}$$

The predicate `p` can be non-deterministic. When `p` happens to be deterministic, in particular, when it is the list constructor `cons` defined by $\mathtt{cons}((\mathtt{A}, \mathtt{X}), [\mathtt{A}|\mathtt{X}])$, the condition (1) simplifies to:

$$\mathtt{r}(\mathtt{X_i}, \mathtt{X_j}) \Rightarrow \mathtt{r}([\mathtt{A}|\mathtt{X_i}], [\mathtt{A}|\mathtt{X_j}]). \tag{2}$$

The **Dynamic Theorem** is applicable to problems in which it is natural for the partial results to be computed by an `unfold`, i.e. by constructing candidate answers from a seed. The theorem also guides the first part of the derivation of the dynamic program, but the programmer is still left to his own ingenuity to derive the appropriate tabling scheme. Given that (2) holds, the specification:

```
d(In,Out) :- bagof(X,unfold(step,base,X,In),Bag),
             best(r,Bag,Out).
```

can be refined to:

```
d(base,[]).
d(In,Out) :- bagof((A,X),step((A,X),In),Bag),
             consmap(d,Bag,Bag1),
             best(r,Bag1,Out).

consmap(P,[],[]).
consmap(P,[(A,X)|Y],[[A|NewX]|NewY]) :- call(P,X,NewX),
                                         consmap(P,Y,NewY).
```

The derived program is better since it filters the input through `best` at each level of recursion, rather than maintain all the unprofitable solutions and choosing the optimal ones at the very end, as the specification does. The second program for `d` describes a recursive scheme in which first `step((A,X),In)` is applied to `In` in all possible ways. These results $(\mathtt{A_1}, \mathtt{X_1}), \ldots (\mathtt{A_n}, \mathtt{X_n})$ are collected by `bagof` in the bag `Bag`. Then the recursive calls to `d` are applied by `consmap` to each of the new seeds $\mathtt{X_1}, \ldots \mathtt{X_n}$. These calls to `d` generate the lists $\mathtt{NewX_1}, \ldots \mathtt{NewX_n}$, which are consequently "consed" with $\mathtt{A_1}, \ldots \mathtt{A_n}$, resulting in the list `Bag1` of solution lists.

As mentioned earlier, the monotonic condition (2) is actually the *Principle of Optimality* stated formally for lists. Here is the reason why it is needed in this derivation. Each seed $\mathtt{X_i}$ may generate many alternatives for $\mathtt{NewX_i}$, and all of these will be contained in `Bag1`. However, since each of these $\mathtt{NewX_i}$ is consed with the same $\mathtt{A_i}$, according to (2), the best $\mathtt{X_i}$'s will always lead to the best solutions. That is why we only need to consider the result of `best` for each decomposition of subproblems.

If the set of decompositions associated with each subproblem is overlapping, a naive evaluation of the derived program will involve much repeating work. However, several declarative languages provide a built-in tabulation facility where the solutions to subproblems can be implicitly recorded and retrieved for subsequent use, and with such evaluation the derived program would have polynomial rather than exponential time complexity. Notice that such implicit tabulation would not reduce the time complexity of the specification in the same way; this is because the specification produces a bag with all the solutions and if there are exponentially many of them, the `best` predicate must take exponential time.

Similarly, the **Greedy Theorem** is used to check whether one can solve an optimisational problem by a greedy algorithm, and for the positive instances to derive this algorithm from the specification. While the dynamic theorem allows us to improve the efficiency of the specification by only considering the best partial solutions for *each* decomposition, the greedy theorem goes much further: the derived greedy program arrives to the optimal solution by only computing the best partial solutions of *one*, best, decomposition at each recursion level. Obviously, the conditions required by this theorem must be rather strong, since we need an additional ordering which will provide us with a hint exactly which decomposition to use, based on local information.

There are actually two greedy theorems, one for `fold` and one for `unfold`, and here we choose to present only the one relevant for our examples, based on `unfold`. If the monotonicity condition (2) is satisfied, and if we can find a preorder `q` defined on pairs which represent problem decompositions, such that:

$$\begin{aligned} &\mathtt{q}((\mathtt{A_i},\mathtt{In_i}),(\mathtt{A_j},\mathtt{In_j})) \wedge \mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{Out_j},\mathtt{In_j}) \;\Rightarrow \\ &\qquad \exists\ \mathtt{Out_i}.\ \mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{Out_i},\mathtt{In_i}) \wedge \mathtt{r}([\mathtt{A_i}|\mathtt{Out_i}],[\mathtt{A_j}|\mathtt{Out_j}]) \end{aligned} \tag{3}$$

then the following program segment:

```
g(In,Out) :- bagof(X,unfold(step,base,X,In),Bag),
             best(r,Bag,Out).
```

can be refined to:

```
g(base,[]).
g(In,[A1|NewX1]) :- bagof((A,X),step((A,X),In),Bag),
                    best(q,Bag,(A1,X1)), g(X1,NewX1).
```

As in the dynamic theorem, the condition (2) enables us to consider only the best partial solutions for each decomposition. Even better, only the best decomposition is needed, and the greedy condition (3) uses `q` to chose this decomposition, say, $(\mathtt{A_i},\mathtt{In_i})$. In cases where the input is such that `best` matches more than one element in `Bag`, we can refine this program further by replacing `best` with a predicate `onebest` which is matched only once.

Note that `q` is an ordering on decompositions, while `r` is an ordering on results. If we have an ordering `q` such that for any pair of decompositions, say $(\mathtt{A_i},\mathtt{In_i})$ and $(\mathtt{A_j},\mathtt{In_j})$, if $(\mathtt{A_i},\mathtt{In_i})$ is preferred to $(\mathtt{A_j},\mathtt{In_j})$ by `q`, that we know that for any possible outcome $\texttt{Out}_\texttt{j}$ of the inferior decomposition, we can find at least one outcome $\texttt{Out}_\texttt{i}$ of the superior decomposition, such that the total result $[\texttt{A}_\texttt{i}|\texttt{Out}_\texttt{i}]$ will be better than $[\texttt{A}_\texttt{j}|\texttt{Out}_\texttt{j}]$, according to the ordering `r`. This is why, in the derived program, we can after each `step` safely chose the best decomposition $(\texttt{A1}, \texttt{X1})$ according to `q`, and recursively apply `g` only to this optimal decomposition without considering others.

We have previously mentioned that two distinct form of greedy theorem exists, one for problems defined in terms of `fold` and one for those defined in terms of `unfold`. The dynamic theorem, however, only works for problems specified with `unfold`. Problems that satisfy the principle of optimality and require a specification in terms of `fold` can be solved by the **Thinning Theorem**. This theorem specifies a monotonicity condition that can be used to discard some of the unprofitable decompositions in the derived program.

Assume that we have preorders `r` and `q` respectively on solutions and on problem decompositions, and predicates `thin` and `pow`, with the following specifications. The preorder `q` is a sub-relation of preorder `r`, meaning $\texttt{q}(\texttt{X},\texttt{Y}) \Rightarrow \texttt{r}(\texttt{X},\texttt{Y})$, and it is not necessarily connected. The predicate `step` is monotonic (as in (1)) on the converse of `q`, which we denote by $\texttt{q}^\circ$. Furthermore, `thin(r,XS,YS)` holds if `YS` is a subset of `XS` and $\forall \texttt{X} \in \texttt{XS}.\ \exists \texttt{Y} \in \texttt{YS}.\ \texttt{r}(\texttt{Y},\texttt{X})$, that is, for each solution in `XS` there is a better solution in `YS`. Finally, `pow(p,(A,XS),Y)` holds if $\exists \texttt{X} \in \texttt{XS}.\ \texttt{p}((\texttt{A},\texttt{X}),\texttt{Y})$, that is, if `p` applied to some arbitrary element of `X` of `XS` yields `Y`. Then the program:

```
t(In,Out) :- bagof(X,fold(step,base,In,X),Bag),
             best(r,Bag,Out).
```

can be refined to:

```
t(In,Out) :- fold(tstep,[base],In,Bag),
             best(r,Bag,Out).

tstep((A,AS),YS) :- bagof(X,pow(step,(A,AS),X),XS),
                    thin(q,XS,YS).
```

The motivation here is that we promote `bagof` in `fold`, so that we can thin it at each recursion stage. At each stage, we use a `pow` operator to apply `step` in all possible ways to the bag of partial solutions, then collect the results. The role of `thin` is to use the preorder `q` to shrink the size of the bag of solutions.

The specification of `thin(q,X,Y)` in effect means that if an element in `X` is worse than some other element with respect to `q`, then by monotonicity we need not keep it in `Y`. Obviously, this specification allows `thin` to return its input unshrank, and in that case the derived program is as inefficient as the specification. To gain from this refinement without relying on tabling, we need to find a `q` that removes a considerable portion of the bag of the partial solutions. On the other hand, if we manage to find a `q` so strong that it is a connected preorder, we can simply keep the best partial solution at each stage, which would correspond to a greedy algorithm.

Finally, the premises for the three theorems are actually stronger than they need to be, so both (2) and (3) can be additionally restricted. A weaker form of (2) states that $\mathtt{X_i}$ and $\mathtt{X_j}$ are composed from the same seed $\mathtt{In}$:

$$(\exists \mathtt{In}.\mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{X_i},\mathtt{In}) \wedge \mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{X_j},\mathtt{In})) \wedge \mathtt{r}(\mathtt{X_i},\mathtt{X_j}) \\ \Rightarrow \ \mathtt{r}([\mathtt{A}|\mathtt{X_i}],[\mathtt{A}|\mathtt{X_j}]) \qquad (4)$$

and, similarly, a weaker form of (3) requires that both $(\mathtt{A_i},\mathtt{In_i})$ and $(\mathtt{A_j},\mathtt{In_j})$ are decompositions of the same input $\mathtt{In}$:

$$(\exists \mathtt{In}.\mathtt{step}((\mathtt{A_i},\mathtt{In_i}),\mathtt{In}) \wedge \mathtt{step}((\mathtt{A_j},\mathtt{In_j}),\mathtt{In})) \\ \wedge\ \mathtt{q}((\mathtt{A_i},\mathtt{In_i}),(\mathtt{A_j},\mathtt{In_j})) \wedge \mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{Out_j},\mathtt{In_j}) \qquad (5) \\ \Rightarrow\ \exists \mathtt{Out_i}.\ \mathtt{unfold}(\mathtt{step},\mathtt{base},\mathtt{Out_i},\mathtt{In_i}) \wedge \mathtt{r}([\mathtt{A_i}|\mathtt{Out_i}],[\mathtt{A_j}|\mathtt{Out_j}])$$

We will use these conditions in section 6.

Now we go on to apply these three theorems to three classic optimisation problems: dynamic theorem to the string edit problem, thinning theorem to 1/0 knapsack problem, and greedy theorem to the minimal lateness problem.

4 Dynamic programming example: string edit

Given two strings x and y, the string edit problem asks for the minimal sequence of editing operations required to transform x into y. The choice of the editing operations varies in different formulations of this problem, and we choose the simplest possible set: *insert* a character into x, *delete* a character from x and *copy* a character in x, i.e. simply retain the character. These three operations contain enough information to construct both strings from scratch, if one interprets *copy* a as "append a to both strings", *insert* a as "append a to the right string" and *delete* a as "append a to the left string".

We choose to represent the strings as lists of characters and the edit sequence as a list containing pairs of $(op, char)$, where op is one of ins, del or cpy. Each operation costs one unit, so the optimal edit sequence is one with a minimal length. Since there might be more than one such solution, we choose the first.

As discussed earlier, the string edit problem constructs the solutions, which are lists of editing instructions, by means of an `unfold` from the seed `(S1,S2)` containing the two input strings. The specification of the problem is thus:

```
edit((S1,S2),Out) :- bagof(X,unfold(step,([],[]),X,(S1,S2)),Bag),
                     best(lleq,Bag,Out).
```

The predicate `step` applies and records one editing instruction to the pair of input strings, and the predicate `lleq` compares the lengths of two edit sequences:

```
step(((cpy,A),(X,Y)),      ([A|X],[A|Y])).
step(((del,A),(X,[])),     ([A|X],[])).
step(((del,A),(X,[B|Y])), ([A|X],[B|Y])).
```

```
step(((ins,B),([],Y)),    ([],[B|Y])).
step(((ins,B),([A|X],Y)), ([A|X],[B|Y])).

lleq(X,Y) :- length(X,M), length(Y,N), M =< N.
```

Obviously `lleq` is monotonic on `cons`, since consing an element to two sequences preserves the length comparison between them. So (2) holds and we can apply the dynamic theorem. A direct application of the theorem results in the following program:

```
edit2(([],[]),[]).
edit2((S1,S2),Out) :- bagof(X,step(X,(S1,S2)),Bag),
                      consmap(edit2,Bag,Bag1),
                      best(lleq,Bag1,Out).
```

Using XSB [10], or some other tabling Prolog system, `edit2` can be automatically tabled and the execution complexity becomes polynomial because there are only $m * n$ different subproblems, where m and n are the lengths of the two sequences. The complexity of the specification `edit`, on the other side, would remain exponential even after tabling.

5 Thinning example: 1/0 knapsack

Given n items, each of weight w_i and of value v_i, and a knapsack of capacity K, the goal is to find the subset of items with maximal total value whose total weight does not exceed K. The naive implementation of this problem constructs all subsets of items within weight limit and returns the subset with the greatest value; the size of the powerset of a set of n elements is 2^n and therefore the complexity of this algorithm is exponential in the number of items:

```
knapsack(W,In,Out) :- bagof(X,fold(step(W),([],0,0),In,X),Bag),
                      best(vgeq,Bag,Out).
```

The predicate `knapsack(W,X,Y)` holds if `Y` is the optimal way to select items from the list `X` within weight `W`. We adopt a `fold` to capture the process of examining all the items one by one. In each `step` we have two choices: to ignore this item, or to add it to the bag if the total weight does not exceed our limit. This is reflected in the 2 premises `step1` and `step2`, respectively:

```
step(W,(A,X),Y) :- step1(W,(A,X),Y).
step(W,(A,X),Y) :- step2(W,(A,X),Y).
step1(_,(_,X),X).
step2(W,(A,X),Y) :- addone(A,X,Y), within(W,Y).

vgeq(A,B) :- value(A,VA), value(B,VB), VA >= VB.
```

The explicit use of nondeterminism here is suggestive and we consider that the relations of a logic programming language give us in this case a clear notational advantage over conventional functional languages. The predicate `addone(A,X,Y)` holds if `Y` is a partial solution gotten by adding the item `A` to the partial solution `X`. The predicate `within(W,Y)` holds if the total weight of all the items in list `Y` is within the weight limit `W`.

Since the specification is expressed in terms of `fold`, we can try to apply either the thinning or the greedy theorem. Optimally, we would like to try to use the greedy theorem, since it results in the simplest and the most efficient algorithm. Unfortunately, `step` is not monotonic on `vgeq`. We cannot give up a selection of items just because it is less valuable than another selection, because the `step` might not be able to add it to the partially filled knapsack due to overflow. However, we can identify a sub-relation `q` of `value` such that `step` is monotonic on a converse of `q`. If a selection of items is not only less valuable, but also heavier than another selection, then this choice of items is definitely not leading to the optimal solution. Given an auxiliary predicate `wleq` which is true if the weight of its first argument is less than or equal to that of its second argument, we define `q` simply as a conjunction:

```
q(A,B) :- vgeq(A,B), wleq(A,B).
```

The proof that `step` is monotonic on `q°` follows directly from the definition of `step`. If `q°(A,B)` holds, then `A` has smaller value and greater weight than `B`. Given a partial solution `X`, we can always find a way to use `step` to extend it with these two items so that the resulting solutions, say `XA` and `XB`, are related by `q°(XA,XB)`, i.e. so that the solution resulting from the less valuable and heavier item will also be less valuable and heavier.

We now know that we can apply the thinning theorem. Actually, a special form of this theorem, known as the *Binary Thinning Theorem* can be applied in this case, because the definition of `step` has only two alternatives. This theorem states that under the conditions described above, we can derive the following program from our knapsack specification:

```
knapsack2(W,In,Out) :- fold(step3(W),[([],0,0)],In,List),
                       head(List,Out).

step3(W,(A,X),YS) :-
   bagof(Y,pow(step1(W),(A,X),Y),Bag1),
   bagof(Y,pow(step2(W),(A,X),Y),Bag2),
   merge(vgeq,Bag1,Bag2,Bag),
   thin(q,Bag,YS).
```

Instead of referring to a theorem not presented here, we could have used simple algebraic calculations to derive this program from the code resulting from the original thinning theorem in a few steps. The sketch of the main steps in this proof is as follows. First, `best r` is refined by a composition of predicates `sort r°` and `head`, since taking the first element of the list sorted in reverse order of `r` gives the optimal element. Then, the fusion theorem is used to push `sort` into the `fold`, and the conditions of the fusion theorem are used to calculate the composition of predicates `bagof`, `merge` and `thin` used in the code above.

Let n be the number of items, and w their total weight. As discussed earlier, the time complexity of the specification is $O(2^n)$. The derived program computes `step3` n times, once for each item in the input list. The size of the bag in `bagof` is in this case bounded by the the total weight w because for each weight the bag contains at most one element, and `thin` and `merge` are easily implemented such that they are linear in the length of the input lists, so the time complexity of the derived program is $O(n * w)$.

This problem is traditionally solved by tabling, where one builds a table containing a row for each item and a column for each weight, up to the total sum of the weights of all items. In each entry the best possible value within the given subset of items and the given weight is recorded. Since the number of entries is the product of the number of items and the total weight, the running time becomes polynomial, so the complexities of the tabled program and our program are comparable. However, unlike the tabling program, our program also works for non-integer weights and values, but in that case the running time becomes exponential.

6 Greedy example: minimal tardiness

The minimum tardiness problem is a scheduling problem from Operations Research. Given a bag of jobs, it is required to find some scheduling of it, that is, some permutation of the bag, that minimises the worst penalty incurred if the scheduled jobs are not completed in their due time. Each job `J` is associated with three quantities: the *completion time* `ct(J,C)`, determining how long it takes to complete the job; the *due time* `dt(J,D)`, determining the latest time before which the job must be completed; and a *weighting* `wt(J,W)`, measuring the importance of the job.

Given the predicates `bagify(X,In)`, which holds if `X` is some permutation of the bag `In`, and `costleq(A,B)`, which holds if the schedule `A` has a cost less than or equal to the schedule `B`, the scheduling problem `sche` can be specified as:

```
sche(In,Out) :- bagof(X,bagify(X,In),Bag),
                best(costleq,Bag,Out).
```

We can implement `bagify` using `unfold` with the auxiliary predicate `bcons`. In one direction, `bcons` adds an item `A` to a bag `X`; used in the reverse way, it nondeterministically picks an arbitrary item from the bag and pairs it with the rest of the bag. With `unfold`, it can be used to generate all the permutations for a bag. The predicate `costleq` is as expected:

```
bagify(Y,X) :- unfold(bcons,[],Y,X).

bcons((A,X),[A|X]).
bcons((B,[A|X]),[A|Y]) :- bcons((B,X),Y).

costleq(X,Y) :- cost(X,CX), cost(Y,CY), CX =< CY.
```

The cost of a scheduling is the maximum penalty of any scheduled job. The relation `penalty((J,X),P)` below denotes that the penalty `P` is incurred when the job `J` is performed after some jobs `X`. Notice that in this example schedules should be read backwards, i.e. the last job is written at the head of the list. If a job is completed before its due time, according to the definition below its assigned penalty is negative, but since we only need to be concerned with the maximum penalty, we choose to ignore negative penalties in the definition of `cost`. To this end we use the predicate `bmax` which simply relates the maximum of its first and second arguments to the third. Given a predicate `totaltime` which calculates the time taken to complete all the jobs in `X` by summing up their completion time, the definitions of `penalty` and `cost` are:

```
penalty((J,X),P) :- ct(J,C), wt(J,W), dt(J,D),
                    totaltime(X,TT), P is (TT+C-D)*W.

cost([],0).
cost([J|X],C) :- penalty((J,X),C1), cost(X,C2), bmax(C1,C2,C).
```

As the number of permutations of a list is exponential in its length, the above specification of `sche` takes exponential time to run. Fortunately, the two conditions of the greedy theorem hold for this specification, so we can derive a greedy algorithm for this problem. We give an informal argument for their validity below, and a formal, calculational proof can be found in [1].

The monotonicity condition states that if the cost of a scheduling `X` is less than or equal to the cost of scheduling `Y`, attaching job `J` to both of them does not change the ordering. In this example we find the weaker version (4) easier to prove. Given a job `J`, schedules `S1`, `S2`, and a bag `B`, we claim that:

$$
\begin{aligned}
&(\exists \mathtt{B}.\ \mathtt{bagify}(\mathtt{S1},\mathtt{B}) \wedge \mathtt{bagify}(\mathtt{S2},\mathtt{B})) \wedge \mathtt{costleq}(\mathtt{S1},\mathtt{S2})\\
&\qquad \Rightarrow\ \mathtt{costleq}([\mathtt{J}|\mathtt{S1}],[\mathtt{J}|\mathtt{S2}])
\end{aligned}
$$

The premise $\exists \mathtt{B}.\ \mathtt{bagify}(\mathtt{S1},\mathtt{B}) \wedge \mathtt{bagify}(\mathtt{S2},\mathtt{B})$, also called the context, says that both `S1` and `S2` are schedulings of the same bag of jobs `B`. For any permutation of a bag of jobs `B`, i.e. any schedule resulting form `B`, the total completion time must be the same. Therefore also adding the same job to both schedules results in a same completion time. But the penalty of $[\mathtt{J}|\mathtt{S1}]$ only depends on the weight of `J` and the total completion time of `S1`, so the penalties for doing `J` after `S1` and `S2` are the same. Since the cost of a scheduling is defined to be the maximum penalty, the monotonicity condition trivially holds.

Proving the greedy condition is trickier, and again we chose to prove the weaker version (5). We need to invent an ordering which will help us to choose the best job to pick in each stage. Formally, we need to find an ordering `q` such that, for jobs `J1`, `J2`, bags `B1`, `B2`, `B` and schedules `S1`, `S2`, we can prove:

$$
\begin{aligned}
&(\exists \mathtt{B}.\ \mathtt{bcons}((\mathtt{J1},\mathtt{B1}),\mathtt{B}) \wedge \mathtt{bcons}((\mathtt{J2},\mathtt{B2}),\mathtt{B}))\\
&\wedge\ \mathtt{q}((\mathtt{J1},\mathtt{B1}),(\mathtt{J2},\mathtt{B2})) \wedge \mathtt{bagify}(\mathtt{S2},\mathtt{B2}))\\
&\qquad \Rightarrow\ \exists \mathtt{S1}.\ \mathtt{bagify}(\mathtt{S1},\mathtt{B1}) \wedge \mathtt{costleq}([\mathtt{J1}|\mathtt{S1}],[\mathtt{J2}|\mathtt{S2}])
\end{aligned}
$$

We claim that the ordering `penaltyleq` is the right choice for `q`, that is, that in each step we only need to follow the decomposition with the least penalty job:

```
penaltyleq(X,Y) :- penalty(X,PX), penalty(Y,PY), PX =< PY.
```

The context of the greedy condition states that (J1, B1) and (J2, B2) are decompositions of the same bag B. Further, the premise penaltyleq((J1, B1), (J2, B2)) means that the job J1 after any scheduling of B1 incurs less penalty than the job J2 after any scheduling of B2. Then, we argue that for any scheduling S2 of the bag B2, there must exist some way to construct a schedule S1 out of the bag B1, such that the total schedule [J1|S1] has lower cost than the schedule [J2|S2].

The argument is as follows. Remember that the cost of [J1|S1] is the maximum of the cost of S1 and the penalty of J1. Regarding the cost of S1, notice first that inserting a job into a schedule either keeps the total schedule cost the same or increases it. Because both [J1|S1] and [J2|S2] are schedulings of the same bag B, we can chose the schedule S1 to be the same as schedule [J2|S2] without the job J1; then the cost of S1 must be less than or equal to the cost of [J2|S2]. Regarding the penalty of J1 after S1, by the premise we know that it is less than or equal to the penalty of J2 after S2.

And then, through a direct application of the greedy theorem, we derive:

```
sche2([],[]).
sche2(B,[J1|S1]) :- bagof(X,bcons(X,B),Bag),
                    best(penaltyleq,Bag,(J1,B1)), sche2(B1,S1).
```

The complexity of this is cubic in n, since there are totally n recursive calls to sche2, in each call we need to examine among a linear number of decompositions to pick the one with least penalty, and the calculation of penalty also takes linear time (to sum up the completion time). We could have refined the data structure such that computing the penalty takes constant time, but we keep the code in this form to emphasise the structure of the program described in the theorem.

7 Conclusion

Following [1], in this paper we propose an alternative approach to optimisation problems in logic programming. While the traditional approach views dynamic programming and greedy algorithms as separate and unrelated programming styles, the approach taken here relates them to the same specification and thereby also makes clear the differences between the two algorithm strategies. Further, this approach also makes it easier to derive the efficient algorithms from the common specification, thus achieving two things: helping the programmer to write clear and efficient code and making him aware of the algorithmical issues at hand. Declarative algorithm design techniques have been an important topic in functional programming; we believe that it is important to translate some of those results into the setting of logic programs.

Traditional tabulation methods have been successfully used to solve dynamic programming problems in logic programming: examples of tabular logic programming systems include XSB [10], DyALog [12], and B-Prolog [13]. There is an alternative approach, advocated by Clocksin in [3], where recomputation is avoided by using data-flow analysis at compile time; this approach is implemented for logic programming in [2, 7]. However, their focus in not on the algorithms; ours is. Our aim in this paper was to introduce, through three examples, an algebraic approach to a classification of some optimisation algorithms and to their derivation. We have not aimed for a thorough semantic treatment here, although that may be a topic for further research.

The examples presented here have thought us a lesson about the expressivity of the two programming styles. We would argue that these problems make a good case for functional logic programming languages, as we found that we needed the best of both worlds for a simple and clear presentation. The examples and the theorems required relations, which we found in logic programming. Non-determinism comes naturally in logic programs, and need not be simulated as in functional code, and the predicate `bagof` and the use of logical variables were a big convenience. On the other hand, higher-order functions are essential for the derivations, and we find that the `call` notation is somewhat cumbersome. We missed currying. Types would have made the theorems more intuitive, because many of our predicates use sets with non-trivial structure. In summary, a language that incorporates these features would be the perfect tool for declarative algorithm derivations. There exist several good candidates, such as Mercury, Curry, Oz or Escher, but one does not even need to go that far: we have earlier presented a simple embedding [11] of logic programs into lazy functional ones, which would add all the relational features to, say, Haskell, for a cost of a few dozen lines of code. Even `bagof` comes almost for free in that setting.

The three theorems in this paper are presented in specialised forms for lists. The class of problems we talked about in this paper either consumes an input list or produces a list as its output. The more general form can be found in [1], where an optimisation problem is represented as `fold` and `unfold` for any initial data-type. However, not all dynamic programming or greedy algorithms can be expressed in terms of `fold` and `unfold`. [5] generalises the model further to cover most dynamic programming or greedy problems. The price, however, is that it is too abstract to guide program transformation.

8 Acknowledgements

We would like to thank Richard Bird, Jeremy Gibbons, and Mike Spivey for their comments on an earlier draft of this paper.

References

1. R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.
2. M. Bruynooghe, L. De Raedt, and D. De Schreye. Explanation based program transformation. In *Proc. of the 11th Intl. Conference on Artificial Intelligence*, pages 407–412, 1989.

3. W.F. Clocksin. Logic-programming specification and execution of dynamic-programming problems. *Journal of Logic Programming*, 12(4), 1990.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* McGraw-Hill, 1990.
5. S. Curtis. *A Relational Approach to Optimization Problems.* PhD thesis, University of Oxford, 1995.
6. O. de Moor. *Categories, relations and dynamic programming.* PhD thesis, Computing Laboratory, Oxford, 1992.
7. D. De Schreye, B. Martens, G. Sablon, and M. Bruynooghe. Compiling bottom-up and mixed derivations into top-down executable logic programs. Technical report, Katholieke Univ. Leuven, 1989.
8. P. Helman. The principle of optimality in the design of efficient algorithms. *Journal of Mathematical Analysis and Applications*, 119:97–127, 1986.
9. Bellman R. *Dynamic Programming.* Princeton University Press, 1957.
10. K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proc. of ACM SIGMOD Intl Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994.
11. J.M. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell'99*, Paris, France, 1999.
12. E. Villemonte de la Clergerie. A tool for abstract interpretation : Dynamic programming. In *Proc. of JTASPEFL*, 1991.
13. N-F Zhou. Beta-prolog: An exted prolog with boolean tables for combinatorial search. In *Proc. 5th IEEE Intl Conference on Tools with Artificial Intelligence*, pages 312–319, November 1993.

A Appendix: General definitions

```
fold(_,E,[],E).
fold(P,E,[A|X],B) :-
        fold(P,E,X,B1), call(P,(A,B1),B).

unfold(_,E,[],E).
unfold(P,E,[A|X],B) :-
        call(P,(A,B1),B), unfold(P,E,X,B1).

best(_,[A],A).
best(R,[A,B|X],C) :- call(R,A,B), best(R,[A|X],C).
best(R,[_,B|X],C) :- call(R,A,B), best(R,[B|X],C).

onebest(_,[A],A).
onebest(R,[A,B|X],C) :- call(R,A,B), !, onebest(R,[A|X],C).
onebest(R,[_,B|X],C) :- onebest(R,[B|X],C).

merge(_,[],X,X).
merge(_,X,[],X).
merge(R,[A|X],[B|Y],[A|Z]) :- call(R,A,B),!,merge(R,X,[B|Y],Z).
merge(R,[A|X],[B|Y],[B|Z]) :- merge(R,[A|X],Y,Z).

thin(_,[],[]).
thin(R,[A|X],Y) :-
        thin(R,X,Y1), bump(R,A,Y1,Y).

bump(_,A,[],[A]).
bump(R,A,[B|X],Y) :-
        call(R,A,B) -> Y = [A|X] ;
        call(R,B,A) -> Y = [B|X] ;
                       Y = [A,B|X].

consmap(_,[],[]).
consmap(P,[(A,X)|Y],[[A|NewX]|NewY]) :-
        call(P,X,NewX), consmap(P,Y,NewY).

pow(P,(A,XS),Y) :-
        member(X,XS),call(P,(A,X),Y).

head([A|_],A).

bmax(X,Y,X) :- X >= Y.
bmax(X,Y,Y) :- X < Y.
```

B Appendix: definitions for string edit

```
%% the input strings (S1,S2) are given as character lists

%% Problem specification

edit((S1,S2),Out) :-
        bagof(X,unfold(step,([],[]),X,(S1,S2)),Bag),
        best(lleq, Bag, Out).

step(((cpy,A),(X,Y)),     ([A|X],[A|Y])).
step(((del,A),(X,[])),    ([A|X],[])).
step(((del,A),(X,[B|Y])), ([A|X],[B|Y])).
step(((ins,B),([],Y)),    ([],[B|Y])).
step(((ins,B),([A|X],Y)), ([A|X],[B|Y])).

lleq(X,Y) :- length(X,M), length(Y,N), M =< N.

%% Refined dynamic programming algorithm

edit2(([],[]),[]).
edit2((S1,S2),Out) :-
        bagof((A,X), step((A,X),(S1,S2)), Bag),
        consmap(edit2,Bag,Bag1),
        best(lleq,Bag1,Out).
```

C Appendix: definitions for 1/0 knapsack

```
%% items should be declared in the form item(Name, Value, Weight).

%% Problem specification

knapsack(W,In,Out) :-
        bagof(X,fold(step(W),([],0,0),In,X),Bag),
        best(vgeq,Bag,Out).

step(W,(A,X),Y) :- step1(W,(A,X),Y).
step(W,(A,X),Y) :- step2(W,(A,X),Y).
step1(_,(_,X),X).
step2(W,(A,X),Y) :- addone(A,X,Y), within(W,Y).

addone(A,(NS,VS1,WS1),([A|NS],VS2,WS2)) :-
        item(A,V,W), VS2 is V + VS1, WS2 is W + WS1.

%% The refined thinning algorithm
```

```
knapsack2(W,In,Out) :-
        fold(step3(W),[([],0,0)],In,List),
        head(List,Out).

step3(W,(A,X),YS) :-
        bagof(Y,pow(step1(W),(A,X),Y),Bag1),
        bagof(Y,pow(step2(W),(A,X),Y),Bag2),
        merge(vgeq,Bag1,Bag2,Bag),
        thin(q,Bag,YS).

%% Auxilary functions for problem specification

within(W,X) :- weight(X,WX), W >= WX.

value((_,VS,_),VS).
weight((_,_,WS),WS).

vgeq(A,B) :-
        value(A,VA), value(B,VB), VA >= VB.
wleq(A,B) :-
        weight(A,WA), weight(B,WB), WA =< WB.
q(A,B) :-
        vgeq(A,B), wleq(A,B).
```

D Appendix: definitions for minimal tardiness

```
%% jobs should be declared in the form job(Name, CT, DT, WT).

%% Problem Specification

sche(In,Out) :-
        bagof(X,bagify(X,In),Bag),
        best(costleq,Bag,Out).

bagify(Y,X) :- unfold(bcons,[],Y,X).

bcons((A,X), [A|X]).
bcons((B,[A|X]), [A|Y]) :- bcons((B,X),Y).

%% The refined greedy algorithm.

sche2([],[]).
sche2(B,[J1|S1]) :-
        bagof((A,X), bcons((A,X),B), Bag),
```

```
         onebest(penaltyleq,Bag,(J1,B1)),
         sche2(B1,S1).

%% Auxilary functions for problem specification

ct(J,C) :- job(J,C,_,_).
dt(J,D) :- job(J,_,D,_).
wt(J,W) :- job(J,_,_,W).

penalty((J,X),P) :-
         ct(J,C), wt(J,W),dt(J,D),
         totaltime(X,TT),
         P is (TT+C-D)*W.

totaltime([],0).
totaltime([J|X],T) :-
         totaltime(X,T1), ct(J,T2), T is T1+T2.

cost([],0).
cost([J|X],C) :-
         penalty((J,X),C1), cost(X,C2), bmax(C1,C2,C).

costleq(X,Y) :-
         cost(X,CX), cost(Y,CY), CX =< CY.
penaltyleq(X,Y) :-
         penalty(X,PX), penalty(Y,PY), PX =< PY.
```