Dynamic Programming: a different perspective

Sharon Curtis

Abstract

Dynamic programming has long been used as an algorithm design technique, with various mathematical theories proposed to model it. Here we take a different perspective, using a relational calculus to model the problems and solutions using dynamic programming. This approach serves to shed new light on the different styles of dynamic programming, representing them by different search strategies of the tree-like space of partial solutions.

1 INTRODUCTION AND HISTORY

Dynamic programming is an algorithm design technique for solving many different types of optimization problem, applicable to such diverse fields as operations research (Ecker and Kupferschmid, 1988) and neutron transport theory (Bellman, Kagiwada and Kalaba, 1967).

The mathematical theory of the subject dates back to 1957, when Richard Bellman (Bellman, 1957) first popularized the idea, producing a mathematical theory to model multi-stage decision processes and to solve related optimization problems. He was also the first to introduce the idea of the *Principle of Optimality*:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

This he presented as the essential condition that such an optimization problem had to satisfy for the dynamic programming technique to be applicable. (Morin (1982) subsequently showed that this is a sufficient but not necessary condition.)

The early standard formalizations of dynamic programming as applied to combinatorial optimization problems concerned *discrete decision processes*, which are a set of strings of decisions (or *policies*) and a cost function on policies. These were modelled as automata by Shreider (1961), and further extended to *sequential decision processes* (discrete decision processes with an added cost structure imposed) by Karp, Held and others (Karp and Held, 1962 and 1967; Elmaghraby, 1970; Ibaraki, 1973). These structures model the problem itself, then the solution is obtained by checking that the principle of optimality applies, and producing the relevant *functional equation*, a recursion equation giving a recipe for the actual computer program. The method of solution was either via *tabulation* of the values of the function concerned, or by direct application of the functional equation, using the technique of *memoization* to avoid computing the same subproblem twice.

The method of performing the computation is little addressed in the early literature, although Bonzon did explore a particular theory for executing particular types of tabulation (Bonzon, 1970).

More recent research has considered the method of computation more fully, and also

alternative formulations of the Principle of Optimality. Mitten (1964) considered a Monotonicity Assumption and more recently, Morin (1982) presented more recent work on this, connecting it to the Principle of Optimality.

Also, in line with recent trends concerning data structures, Helman (Helman and Rosenthal, 1985; Helman, 1986; Helman, 1989) has reworked traditional dynamic programming using trees rather than just strings, and specifically separating the ideas of problem structure and computation. Further work by de Moor (de Moor, 1992; Bird and de Moor 1992) has extended the theory of data structures used within dynamic programming to initial datatypes.

In this paper, dynamic programming will be presented from a different angle, more similar to a technique already in existence that involves keeping several candidate partial solutions at each stage of the algorithm, and discarding unnecessary ones. This is a summary of work from Curtis (1996). Helman's strategy of separating structure from computation will be exercised, but we will generalize from both Helman and de Moor's work, as the datatypes used are not restricted to initial datatypes.

A calculus of relations will be used to model both optimization problems and the computation of their solutions. A simple imperative-style loop operator will model the construction of feasible solutions to a solution, similar to a discrete decision process. The computation of an optimal solution will also be modelled using a relational loop, and the relationship of this particular style of dynamic programming towards other more traditional styles will be investigated.

This work is more abstract than previous approaches, exploring the subject from a higher level.

And as every walker knows, if you take a high level route, you get a much better view.

2 THE *LIM* MODEL

Firstly, we briefly review the operators needed, before going on to examine the structure of dynamic programming problems and their solutions.

2.1 Preliminaries

A relation $R : A \leftarrow B$ is a subset of $A \times B$, and we will use the shorthand xRy (or sometimes $x \overset{R}{\longleftarrow} y$) for $(x, y) \in A \times B$.

The intersection, union and converse operators, $\cap, \cup, ^\circ$ respectively, have their usual meanings, and relational composition $\cdot$ will be from right to left, in a similar style to functional programming. The universal properties for $\cap$ and $\cup$ are as follows:

$$P \cup Q \subseteq R \quad \equiv \quad P \subseteq R \ \wedge \ Q \subseteq R \tag{1}$$

$$P \subseteq Q \cap R \quad \equiv \quad P \subseteq Q \ \wedge \ P \subseteq R \tag{2}$$

Converse distributes over $\cup$ and $\cap$, which also distribute over each other. Composition distributes over $\cup$, and converse has the following property:

$$(P \cdot Q)^\circ \;=\; Q^\circ \cdot P^\circ \tag{3}$$

The left and right quotient operators are defined by the following universal properties

$$R \cdot S \subseteq T \;\equiv\; R \subseteq T/S \tag{4}$$
$$R \cdot S \subseteq T \;\equiv\; S \subseteq R\backslash T, \tag{5}$$

and are called quotients because of the following cancellation laws

$$R/S \cdot S \subseteq R \tag{6}$$
$$R/(S \cdot T) \cdot S \subseteq R/T \tag{7}$$
$$R/S \cdot S/T \subseteq R/T \tag{8}$$

(and similarly for $\backslash$). A relation f is a function if $f \cdot f^\circ \subseteq id$ and $id \subseteq f^\circ \cdot f$, where id is the identity relation. Functions interact with quotients with the following rules:

$$R/S \cdot f \;=\; R/(f^\circ \cdot S) \tag{9}$$
$$f \cdot R/S \;=\; (f \cdot R)/S. \tag{10}$$

The projection functions $outl$ and $outr$ return the left and right element of an ordered pair, respectively. The product operator $\times$ is such that if $R : A \leftarrow B$ and $S : C \leftarrow D$ then $R \times S : A \times C \leftarrow B \times D$ and

$$(a, c)(R \times S)(b, d) \;\Leftrightarrow\; aRb \;\wedge\; cSd. \tag{11}$$

Products interact with the projection functions as follows

$$outl \cdot (R \times S) \subseteq R \cdot outl, \tag{12}$$
$$outr \cdot (R \times S) \subseteq S \cdot outr. \tag{13}$$

(Note that the precedence of the operators introduced so far is (from loosest to tight binding): $\cup, \cap, \times, \cdot, /, \backslash, ^\circ$.)
The converse of the membership relation $\in$ is denoted by $\ni$, and the power transpose operator Λ applied to a relation R yields a function returning the set of elements relating by R to the input, also defined by the following equivalence:

$$f \;=\; \Lambda R \;\Leftrightarrow\; \in \cdot f \;=\; R. \tag{14}$$

A useful property known as Λ-cancellation is the following

$$\Lambda R \cdot R^\circ \subseteq \ni. \tag{15}$$

A related operator is E, which returns a function defined as follows

$$(\mathsf{E}R)X \;=\; \{y \mid \exists x \bullet yRx\}, \tag{16}$$

and satisfies the following properties:

$$\mathsf{E}R \;=\; \Lambda(R \cdot \in) \tag{17}$$
$$\in \cdot \mathsf{E}R \;=\; R \cdot \in \tag{18}$$
$$\mathsf{E}R \cdot \tau \;=\; \Lambda R, \tag{19}$$

where τ applied to x returns $\{x\}$. The powerset operator P relates two sets of elements as follows

$$\mathsf{P}R \;=\; \in\backslash(R \cdot \in) \;\cap\; (\ni \cdot R)/\ni \tag{20}$$

That is, $X(\mathsf{P}R)Y$ precisely when every element of X is related by R to an element of Y, and vice versa.
The following two operators are subsets of the identity relation: $dom\, P$ corresponds to those elements in the domain of P, and vice versa for $notdom\, P$

$$dom\, P \;=\; \{(x,x) \mid \exists y \bullet yPx\} \tag{21}$$
$$notdom\, P \;=\; \{(x,x) \mid \neg\exists y \bullet yPx\}. \tag{22}$$

Properties concerning these operators that we will need are

$$dom\, P \;\cup\; notdom\, P \;=\; id \tag{23}$$
$$Q \cdot notdom\,(P \cdot Q) \;\subseteq\; notdom\, P \cdot Q \tag{24}$$
$$notdom\,(P \cdot Q) \cdot Q^\circ \subseteq\; Q^\circ \cdot notdom\, P \tag{25}$$
$$notdom\, P \cdot dom\, P \;=\; \emptyset \tag{26}$$
$$notdom\, P \;\subseteq\; notdom\, Q \;\equiv\; dom\, Q \;\subseteq\; dom\, P \tag{27}$$

The operator min is used for taking a minimum with respect to a relation:

$$min\, R \;=\; \in \;\cap\; R/\ni, \tag{28}$$

that is, $x(min\, R)X$ precisely when $x \in X$ and for all $y \in X$, xRy. For a reflexive relation R, we also have that

$$min\, R \cdot \ni \;=\; R. \tag{29}$$

A universal property concerning minima is the following:

$$P \;\subseteq min\, R \cdot \Lambda Q \;\equiv\; P \subseteq Q \;\wedge\; P \cdot Q^\circ \subseteq R \tag{30}$$

2.2 The Problem Structure

To formally specify an optimization problem, the feasible solutions to the problem and the criterion for optimality must both be expressed.

For expressing the choice of an optimal solution, we use the *min* operator. This yields slightly more generality than non-relational formalisms: the use of a relation $min\, R$ can be used to minimize or maximize a cost function, for example $R = f^\circ \cdot \geq \cdot f$, but more general orderings can also be used, such as the lexicographical ordering.

To express relationally the feasible solutions to a problem, we will use the following notion of limits. The operator $lim\ T$ is defined to be the least solution X of

$$X \;=\; notdom\ T \;\cup\; X \cdot T. \tag{31}$$

This gives a relational model of a simple loop with guard $dom\ T$ and loop body T. That is, $lim\ T$ does T until it can do so no more, and then stops. Some useful properties of *lim* (that are easily derived from previous laws) are as follows:

$$lim\ T \cdot notdom\ T \;=\; notdom\ T \tag{32}$$
$$lim\ T \cdot dom\ T \;=\; lim\ T \cdot T \tag{33}$$
$$lim\ T \cdot T \;\subseteq\; lim\ T. \tag{34}$$

Also, the rule for least fixpoints gives the following useful property:

$$lim\ T \subseteq Q \quad \Leftarrow \quad notdom\ T \;\cup\; Q \cdot T. \tag{35}$$

We will use the limit operator to model both the specification of feasible solutions to a problem, and the dynamic programming computation of an answer.

Optimization problems expressed in their most general form can be specified relationally as follows:

$$min\, R \cdot \Lambda\, G.$$

In this specification, we call the relation G the *generator* as it generates a single feasible solution from the input. Thus $\Lambda\, G$ generates the set of all possible feasible solutions, and $min\, R$ then selects a best one according to the relation R. We will use limits to generate feasible solutions:

$$min\, R \cdot \Lambda(lim\ T).$$

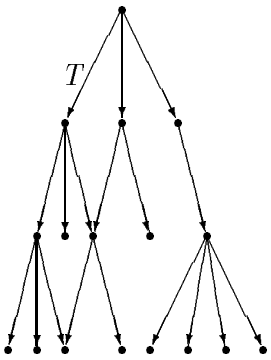
The use of a simple loop to generate feasible solutions is similar to earlier work on automata, although not quite like discrete decision processes, as we are not restricted to the datatype of strings for completed solutions (policies). Helman and Rosenthal (1985) and de Moor (1992) have both considered more general datatypes, trees and

more general initial datatypes, respectively. De Moor uses relational catamorphisms and anamorphisms as generators; these are relational versions of the *fold* operator of functional programming, and their converses. Although this is an elegant theory, the use of non-initial datatypes is not covered. The *lim* operator is not restricted to any particular datatype.

2.3 Searching the Space of Partial Solutions

One simple way to solve the above specification is to execute it directly: that is, by enumerating all possible feasible solutions with $\Lambda(lim\ T)$, then selecting an optimal solution using $min\ R$. This is not at all efficient. Dynamic programming is a strategy that aims to reduce the amount of computation necessary, and we will consider this aim in the context of searching the space of partially-completed feasible solutions.

We first consider how the completed solutions to the problem are generated. The following diagram represents an example of how feasible solutions are built up from the input:



Each node represents a partial solution. The node at the top represents the input, and for each node, the set of its children is the set given by ΛT applied to that node. Thus each arrow represents a possible application of T. The diagram is tree-like, but technically not a tree, as some of the nodes may coincide. The completed solutions are the leaves (that is, nodes with no children), corresponding to partial solutions that are not in the domain of T.

The above tree of partial solutions may also be likened to a diagram of an automaton representing a discrete decision process, with the nodes representing states, and T the transition predicate. The policies (Bellman's terminology for completed solutions) would be the strings corresponding to the paths from the root to the leaves. Policies correspond to the complete history of decisions made, whereas our partial solutions are more general, and can contain this information and more; also they can use more general datatypes than strings and other initial datatypes (Malcolm, 1990).

The straightforward implementation of the specification would involve generating all

the leaves of the tree from the input. In order to consider how we might reduce the amount of computation, we consider how the set of completed solutions might be generated from the input.

We use a relational operator that takes a set of partial solutions and moves one step closer to the set of completed solutions. This operation we will call *sprouting*:

$$sprouts\ T \quad = \quad cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom\ T \times id) \cdot luni^{\circ} \tag{36}$$

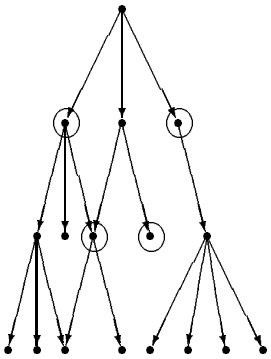
where

$$luni\,(x, y) \quad = \quad x \cup y, \ \text{if } x \neq \{\,\} \ \wedge \ x \cap y = \{\,\} \tag{37}$$

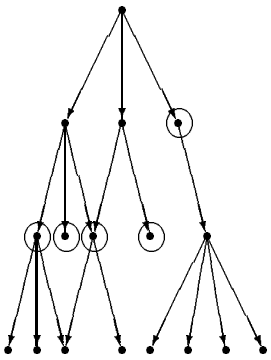
$$cup\,(x, y) \quad = \quad x \cup y \tag{38}$$

Translating the above into English, *sprouts* T takes the input set, takes some uncompleted partial solutions out of the set, applies T in all possible ways to them, then adds these new partial solutions back into the set, doing nothing to the unchosen solutions apart from retaining them.

Consider the following example, to illustrate how it works. Suppose the input set of partial solutions are those circled in this diagram:



then sprouting just the left-most node would result in the set of partial solutions represented by the ringed set here:



Properties of the relations mentioned above that will be used in equational reasoning are the following:

$$\in \cdot \mathit{sprouts}\ T \subseteq T \cdot \in \cup \in \tag{39}$$
$$\in \cdot \mathit{luni} = \in \cdot \mathit{outl} \cup \in \cdot \mathit{outr} \tag{40}$$
$$\mathit{cup} \cdot \mathit{outl}^\circ \subseteq \ni / \ni \tag{41}$$
$$\mathit{cup} \cdot \mathit{outr}^\circ \subseteq \ni / \ni \tag{42}$$

We will also use more particular styles of sprouting. The relation *allsprouts* sprouts every possible uncompleted solution, and leaves completed solutions alone:

$$\mathit{allsprouts}\ T = \mathsf{E}(T \cup \mathit{notdom}\ T) \cdot \mathit{dom}\,(T \cdot \in). \tag{43}$$

The relation $\mathit{sprout}\ T$ sprouts a single uncompleted solution:

$$\mathit{sprout}\ T = \mathit{cup} \cdot (\Lambda T \cdot \mathit{dom}\ T \times \mathit{id}) \cdot \mathit{lcons}^\circ, \tag{44}$$

where

$$\mathit{lcons}\,(x, y) = \{x\} \cup y, \text{ if } x \notin y. \tag{45}$$

Straightforward calculation can be used to show that

$$\mathit{allsprouts}\ T \subseteq \mathit{sprouts}\ T \tag{46}$$
$$\mathit{sprout}\ T \subseteq \mathit{sprouts}\ T. \tag{47}$$

To continue the gardening theme, after sprouting some fresh partial solutions, we want to retain the ones that might lead to a best solution, and remove ones that we know are worthless. The gardening terminology for removing unwanted plants that have sprouted is *thinning*, and that is exactly what we shall be doing.

The following relation thins a set with respect to a preorder S:

$$\mathit{thin}\ S = \in \backslash \in \cap\ (\ni \cdot S)/\ni. \tag{48}$$

That is, a subset of the original is returned, so that every member of the original set has something S-ier than it in the subset.

So we will use a relation S to compare partial solutions to decide which are definitely going to result in a better final solution. This *comparison relation* is similar to Helman's notion of *dominance relations*. To prove correctness, we will be using a variation of the Monotonicity Assumption (Mitten, 1964). Often two partial solutions will be incomparable, which is why we do a thinning rather than taking a minimum.

2.4 Dynamic Programming

Using the concepts of thinning and sprouting as explained above, we will model a dynamic programming step by the relation

$$D \;=\; \mathit{thin}\, S \cdot \mathit{sprouts}\, T.$$

Here a set of partial solutions is maintained, and $\mathit{sprouts}\, T$ performs some amount of construction on these, that is, it applies T to some of the partial solutions. The relation S is a comparison relation which can indicate whether a partial solution is worse than another, and the relation $\mathit{thin}\, S$ removes some of the worse partial solutions. The entire algorithm is

$$\mathit{min}\, R \cdot \mathit{lim}\, D \cdot \tau,$$

thus τ puts the input into a singleton set, then $\mathit{lim}\, D$ repeats the dynamic programming step until all the partial solutions are completed, and then an optimum is selected using $\mathit{min}\, R$. Thus we arrive at the following dynamic programming theorem:

Theorem 1
Let

$$\begin{aligned} M &= \mathit{min}\, R \cdot \Lambda\, \mathit{lim}\, T \\ D &\subseteq \in\backslash\in \cdot \mathit{sprouts}\, T, \end{aligned}$$

where R is a preorder on the set of completed solutions represented by $\mathit{notdom}\, T$, and the following conditions are satisfied:

(i) $\quad \mathit{dom}\,(T \cdot \in) \;\subseteq\; \mathit{dom}\, D$

(ii) $\quad \mathit{lim}\, T \cdot \in \cdot D^{\circ} \subseteq\; R \cdot \in \cdot \mathit{lim}\, T.$

Then

$$\mathit{min}\, R \cdot \mathit{lim}\, D \cdot \tau \;\subseteq\; M.$$

Proof. See the appendix. □

The definition of the relation D is slightly different to that mentioned above; D is not given precisely, but instead is merely restricted to containment in $\in\backslash\in \cdot \mathit{sprouts}\, T$, which does include $\mathit{thin}\, S \cdot \mathit{sprouts}\, T$. There is no need to force D to be equal to $\mathit{thin}\, S \cdot \mathit{sprouts}\, T$, as condition (ii) of the theorem restricts S appropriately. Most algorithms that have been derived from this theorem do use $D = \mathit{thin}\, S \cdot \mathit{sprouts}\, T$, although one algorithm suggested by a colleague of mine, Jesús Ravelo, required the more general format.

The condition (i) simply states that it is possible to perform a dynamic programming step D whenever there is still an uncompleted solution within the set of partial solutions maintained: that is, the algorithm does not stop prematurely.

The final condition (ii) is the one ensuring correctness. Translated into English, it says that given a set X of partial solutions so far, then if a partial solution $x \in X$ is completed in some way, way, and perform the dynamic programming step D on X to obtain Y, then there is a partial solution in Y which can be completed to yield a completion at least as good as the completion of x. That is, D does not throw away a partial solution necessary for optimality.

This last condition is a difficult one to prove directly, and so the following lemmas give results which reduce it to a monotonicity condition, making the proofs a lot easier!

(The lemmas are proved in the appendix.)

Lemma 1
If $D \subseteq thin\, S \cdot sprouts\, T$ *and* $S \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R$, *then*

$$D \cdot \ni \cdot (lim\ T)^\circ \subseteq \ni \cdot (lim\ T)^\circ \cdot R.$$

This reduces the necessary proof condition to $S \cdot (lim\ T)^\circ \subseteq (lim\ T)^\circ \cdot R$, which straightforwardly translates to mean that if a partial solution x is worse with respect to S than another y, that is $x \overset{S}{\longleftarrow} y$, then for any completion of x there is always a better completion of y.

This condition can be simplified further to a simple monotonicity condition, which is usually expressed in the form $S \cdot T^\circ \subseteq T^\circ \cdot S$, indicating that a better partial solution at one stage can be constructed one step further using T to result in a better partial solution at the next stage. However not all partial solutions take the same number of steps to completion, and thus suitable conditions on domains result in the following lemma:

Lemma 2
If the following conditions hold,

$$dom\ T \cdot S \cdot T^\circ \subseteq T^\circ \cdot S \tag{49}$$
$$notdom\ T \cdot S \cdot T^\circ \subseteq S \tag{50}$$
$$S \cdot notdom\ T \subseteq (lim\ T)^\circ \cdot R \tag{51}$$

then

$$lim\ T \cdot S^\circ \subseteq R^\circ \cdot lim\ T.$$

The first condition is the monotonicity assumption, for uncompleted solutions. The second says that a completed solution better than a partial solution remains better if

you complete the partial solution. The final condition says that if a partial solution is better than a completed solution, there is a way to complete the partial solution so that it remains better with respect to R.

Lemma 3
If the following conditions hold

$$dom\ T \cdot S \cdot T^\circ \subseteq T^\circ \cdot S \tag{52}$$
$$notdom\ T \cdot S \cdot notdom\ T \subseteq R \tag{53}$$
$$S \cdot notdom\ T = notdom\ T \cdot S, \tag{54}$$

then

$$lim\ T \cdot S^\circ \subseteq R^\circ \cdot lim\ T.$$

This is similar to the previous lemma, with conditions slightly altered for those problems which are compared at the same stage of development. Then the second condition says that S respects R on completed solutions, and the third says that if S is comparing one completed solution with another solution, the other is completed too. These are a little easier to prove for suitable problems.

We will use the above lemmas in the following examples, and the examples will be used as motivation to discuss aspects of this particular style of dynamic programming.

3 EXAMPLES AND DISCUSSION

3.1 *0-1 Knapsack Problem*

The well-known 0-1 Knapsack problem concerns a thief ransacking a safe containing various items of given weights and values. The thief has to maximize the total value of the haul, subject to the total weight being less than some maximum capacity C.

We will assume two functions *wgt* and *val* to give the weight and value, respectively, of items in the safe. Partial solutions will be the weight and value of the packing so far, together with the list of items not yet considered. Possible packings can be generated using the following relation:

$$\begin{aligned} (w, v, is) \quad & T \quad (w, v, i : is) \\ (w + wgt\ i, v + val\ i, is) \quad & T \quad (w, v, i : is), \text{ if } w + wgt\ i \leq C, \end{aligned}$$

with the input $(0, 0, ss)$, where ss is the list of items in the safe initially. The second

line corresponds to the selection of item i, and the first corresponds to its rejection. Defining R by

$$(w_1, v_1, [\,])\ R\ (w_2, v_2, [\,]) \quad \equiv \quad v_1 \geq v_2,$$

the problem is specified as $min\, R \cdot \Lambda\, lim\, T$.

To solve this problem with dynamic programming, we need a comparison relation S which says when one partial solution will definitely be better than another. A reasonable S to define is the following

$$(w_1, v_1, is)\ S\ (w_2, v_2, is) \quad \equiv \quad v_1 \geq v_2 \quad \wedge \quad w_1 \leq w_2,$$

which translated into words, says that a partial solution is better if it is more valuable and lighter.

We now use Lemma 3, which is suitable for problems such as this, where completed solutions are all constructed from the same number of T steps.

To check the first condition,

$$dom\ T \cdot S \cdot T^\circ \quad \subseteq \quad T^\circ \cdot S,$$

suppose that

$$(w_1, v_1, i : is) \xleftarrow{S} (w_2, v_2, i : is) \xrightarrow{T} (w_2, v_2, is),$$

where item i is rejected, then

$$(w_1, v_1, i : is) \xrightarrow{T} (w_1, v_1, is) \xleftarrow{S} (w_2, v_2, is).$$

Alternatively, if i is accepted, so that

$$(w_1, v_1, i : is) \xleftarrow{S} (w_2, v_2, i : is) \xrightarrow{T} (w_2 + wgt\ i, v_2 + val\ i, is),$$

then as $w_1 \leq w_2$, $w_1 + wgt\ i \leq w_2 + wgt\ i$ and so

$$(w_1, v_1, i : is) \xrightarrow{T} (w_1 + wgt\ i, v_1 + val\ i, is) \xleftarrow{S} (w_2 + wgt\ i, v_2 + val\ i, is).$$

The second and third conditions follow directly from the definition of S.

Thus we know that dynamic programming is applicable to this problem. We can implement this algorithm as the standard method for solving this problem: the maximum possible thinning is done at each stage (often a good strategy), and the maximum possible sprouting is done using *allsprouts* T at each stage.

Using functional programming the set of partial solutions can be kept as an ordered list of partial solutions, ordered by decreasing value and weight. Then at each stage the sprouting and thinning is implemented by a simple merge and purge operation on two lists (one representing the choice of the next object, one representing the rejection of the next object), that also removes solutions worse with respect to S. Then when the solutions are completed, the most valuable packing is at the head of the list. □

3.2 Use of the Recursive Equation

In the example above, it was shown that dynamic programming was a possible technique to use for solving the problem, but the algorithm given by the theorem is still considerably abstract, and there is still considerable freedom of implementation. This is the trade-off that happens with such a theorem. As we will see, many different sorts of dynamic programming algorithms are covered with this theorem. However, this generality abstracts away from the actual implementation.

The above method is an illustration of the common dynamic programming method that keeps a set of potential candidates. Let us compare this to a recursive algorithm obtained directly from the functional equation corresponding to the problem. Let $P_C(is)$ be the value of the best packing subject to weight capacity C, from a sage with items is. Then the recursion equation is

$$\begin{array}{rcll} P_C([\,]) & = & 0 & \\ P_C(i:is) & = & P_C(is), & \text{if } wgt\ i > C \\ & = & \bigsqcup\{P_C(is), val\ i + P_{C-wgt\ i}(is)\}, & \text{otherwise.} \end{array}$$

If this were to be implemented directly (using memoization, say, to avoid the computation of similar results), the movement of the computation across the space of partial solutions would result in a depth-first search across the tree. This is in contrast to the "candidates" method described above, which performs a breadth-first search, searching all partial solutions at one level at the same step.

3.3 *The Paragraph Formatting Problem*

We will consider a simple form of the sort of paragraph formatting problem considered by Knuth and Plass (1981). Given is a list of words, to be formatted into as neat a paragraph as possible, by inserting suitable line breaks in a list of words.

A paragraph is a list of lines, each line being a list of words. It must also fit onto the paper, and thus has a maximum line width W. For a simple measure of neatness, we define the *whitespace* of a line as its length subtracted from W; the untidiness of a paragraph is then the sum of the squares of the whitespaces of every line except the last, and we wish to minimize this.

To formally specify this, we first construct paragraphs using the limit operator. There are several ways by which line breaks can be added to a list of words: one simple way is to add them sequentially starting with the first line. Thus we define

$$(ls +\!\!+ [l], y)\ T\ (ls, l +\!\!+ y), \qquad \text{if } 0 < \mathit{linelength}\ l \leq W$$

and the input will be $([\,], ws)$ where ws is the list of words to be formatted. The above definition assumes that the line length of an empty line is 0, and that non-empty lines have length greater than 0. Note that the function *linelength* is left undefined: the definition of this will depend on the words and font chosen, and this is not important for what follows.

With the following definitions

$$\begin{aligned} \mathit{whitespace}\ l &= W - \mathit{linelength}\ l \\ \mathit{whites}\ ls &= \mathit{sum}\,(\mathit{map}\,(\mathit{square} \cdot \mathit{whitespace})\,ls) \\ \mathit{untidiness}\ (ls +\!\!+ [l]) &= \mathit{whites}\ ls, \end{aligned}$$

we can now define R to be

$$(ls_1, [\,])\,R\,(ls_2, [\,]) \equiv \mathit{untidiness}\ ls_1 \leq \mathit{untidiness}\ ls_2.$$

The problem of formatting paragraphs is now specified as $\mathit{min}\,R \cdot \Lambda\,\mathit{lim}\,T$.

To solve this by dynamic programming, we need to find a comparison relation S to compare partial paragraphs. Once lines at the beginning of the paragraph are chosen, they do not subsequently change, so an obvious choice for S is to compare the untidiness of the already chosen lines. Furthermore, if the paragraphs are not completed, then the last lines of the partial paragraphs can be compared too.

Hence we define

$$\begin{aligned} &(ls_1, y)\,S\,(ls_2, y), && \text{if } \mathit{whites}\ ls_1 \leq \mathit{whites}\ ls_2 \ \wedge\ y \neq [\,] \\ &(ls_1, [\,])\,S\,(ls_2, [\,]), && \text{if } (ls_1, [\,])\,R\,(ls_2, [\,]). \end{aligned}$$

To prove that S can be used, we use Lemma 3. Note that (ls, y) is in the domain of T precisely when y is non-empty. First we need that $\mathit{dom}\ T \cdot S \cdot T^\circ \subseteq T^\circ \cdot S$. If

$$(ls_1, l +\!\!+ y) \xleftarrow{S} (ls_2, l +\!\!+ y) \xrightarrow{T} (ls_2 +\!\!+ [l], y),$$

then from the definitions of T and S, we have that

$$(ls_1, l +\!\!+ y) \xrightarrow{T} (ls_1 +\!\!+ [l], y) \xleftarrow{S} (ls_2 +\!\!+ [l], y).$$

The second and third conditions for Lemma 3 trivially follow from the definition of S.

We now know that S is a suitable relation to thin partial solutions with, but an algorithm is still a long way off, as sprouting and thinning can be done in many possible ways. Usually doing the maximum possible amount of thinning is a good idea, as there is no reason to retain useless partial solutions. However there is a good reason to be careful what sort of sprouting we perform. Suppose there is a "bad" partial solution (one destined to be thinned eventually), and a partial solution that is better with respect to S is not yet available, because it is not yet developed to that stage. Sprouting the bad partial solution is unnecessary computation, and instead sprouting less developed partial solutions is a better strategy.

So for this problem, the partial solutions to be sprouted at each stage are the ones with the most words left to place. Hence at each stage we want to sprout the partial solution (ls, y) for which y is longest. The partial solutions could be kept in a list in descending order of the length of their second component. Then just the head of the list is sprouted at each stage. If the result of the sprout is put into a similarly ordered list, then the thinning can be performed by a linear merging and purging operation on the two lists. If there are n words in the original list, and partial solutions are kept labelled with their white space usage, the result is an $O(n^2)$ algorithm. □

3.4 Sub-problems

Intrinsic to the main idea of dynamic programming is the existence of sub-problems. These arise from the functional equation, as computation of the optimal solution overall requires the recursive computation of optimal solutions to smaller portions of the input. It is the careful planning so that solutions to sub-problems are computed only once that results in unnecessary computation being avoided.

In standard dynamic programming the idea is that if the computation of two distinct problems requires the solution to the same sub-problem, then the sub-problem is solved once, and the result used for the solution of both. Either tabulation or memoization may be used to make sure that solutions are computed at most once.

Within the *lim* model, a partial solution contains information about both the remaining input and the construction so far towards a completed solution. The use of limits means that the notion of a sub-problem is not present explicitly, as it is in the standard style.

In contrast to the standard treatment of sub-problems arising from a recursion equation, in this style of dynamic programming when two partial solutions both require a solution to the same sub-problem, they are compared using the comparison relation S, then the worse is discarded, and the better partial solution may remain for the start of the computation on the rest of the input (the sub-problem).

For example, consider the two partial solutions of the above paragraph formatting problem:

(["`I remember the time I knew what happiness was;`"],"`Let the memory live again.`")
(["`I remember`","`the time I knew what happiness was;`"],"`Let the memory live again.`")

The first is better with respect to the S above, and thus the second may be discarded. The first partial solution (or another better than it) will remain in the current set of partial solutions, and will eventually (unless discarded on account of a better solution) have its remaining word list "`Let the memory live again.`" processed.

3.5 Example: *The String Edit Problem*

The string editing problem concerns the transformation of one string into another, using as few operations as possible. This has many applications: the number of operations required is called the *edit distance* or Levenshtein distance (see Levenshtein, 1966, or Wagner and Fischer, 1974) between the two strings, and this can be used for spell-checking, speech recognition, and comparison of DNA sequences (see Galil and Giancarlo, 1989) for example. Sankoff and Kruskal's book (1983) is the comprehensive reference on the subject.

We will consider the following edit operations on strings: adding, deleting, or retaining a character. If the editing is carried out from left to right of the given word, then a partial solution can be represented by a triple (es, u, v), where es is the list of edits done so far, u is the remainder of the word that we are transforming, and v is the remainder of the word required. An edit step can be performed by the relation T:

$$\begin{array}{lll} (es +\!\!+ [Ret\ c], u, v) & T & (es, c : u, c : v) \\ (es +\!\!+ [Add\ c], u, v) & T & (es, u, c : v) \\ (es +\!\!+ [Del\ c], u, v) & T & (es, c : u, v). \end{array}$$

As it is desired to find the shortest list of edit operations possible, we define

$$(es_1, [\,], [\,])\, R\, (es_2, [\,], [\,]) \quad \equiv \quad \#es_1 \leq \#es_2.$$

The problem can now be specified as $min\, R \cdot \Lambda\, lim\ T$, with the input $([\,], w_1, w_2)$, with w_1 being the given word, and w_2 being the required word.

In order to use dynamic programming for this problem, we need a comparison relation S to determine when one partial solution is better than another. One obvious choice for S is that if two partial solutions are at the same stage with respect to the remaining input, then the one which has used fewest edit operations so far must be better. Hence we define

$$(es_1, u, v)\, S\, (es_2, u, v), \quad \text{if} \quad \#es_1 \leq \#es_2$$

We need to prove that this is a suitable choice of S. Using Lemma 3, the first condition required is that $dom\ T \cdot S \cdot T^\circ \subseteq T^\circ \cdot S$, and we must consider all three possibilities that T° might perform on the left hand side. If

$$(es_1, c : u, c : v) \xleftarrow{S} (es_2, c : u, c : v) \xrightarrow{T} (es_2 +\!\!+ [Ret\ c], u, v),$$

then from the definitions of T and S,

$$(es_1, c : u, c : v) \xrightarrow{T} (es_1 +\!\!+ [Ret\ c], u, v) \xleftarrow{S} (es_2 +\!\!+ [Ret\ c], u, v).$$

For the second possibility, if

$$(es_1, c : u, v) \xleftarrow{S} (es_2, c : u, v) \xrightarrow{T} (es_2 +\!\!+ [Del\ c], u, v),$$

then from the definitions of T and S, we have that

$$(es_1, c : u, v) \xrightarrow{T} (es_1 +\!\!+ [Del\ c], u, v) \xleftarrow{S} (es_2 +\!\!+ [Del\ c], u, v),$$

and the third case is symmetrical to this one. The second and third conditions of Lemma 3 follow trivially from the definition of S.

We have yet to decide on an actual algorithm. Again, the consideration that least-developed partial solutions should be sprouted first applies.

Hence at each stage we want to sprout the partial solutions for which there is most unprocessed input. Thus we do not wish to sprout a partial solution (es_1, u, v) if there is another partial solution $(es_2, u' +\!\!+ u, v' +\!\!+ v)$ in the set, with at least one of u' and v' non-empty. One way to implement this strategy is to at each stage sprout the partial solutions with minimal $\#u + \#v$. □

3.6 Tabulation

The tabulation of results from the solving of sub-problems is one of the most important techniques in dynamic programming. However, the use of limits to construct feasible solutions has resulted in an abstraction away from the structures of the optimization problems, so there is no longer a notion of sub-problem.

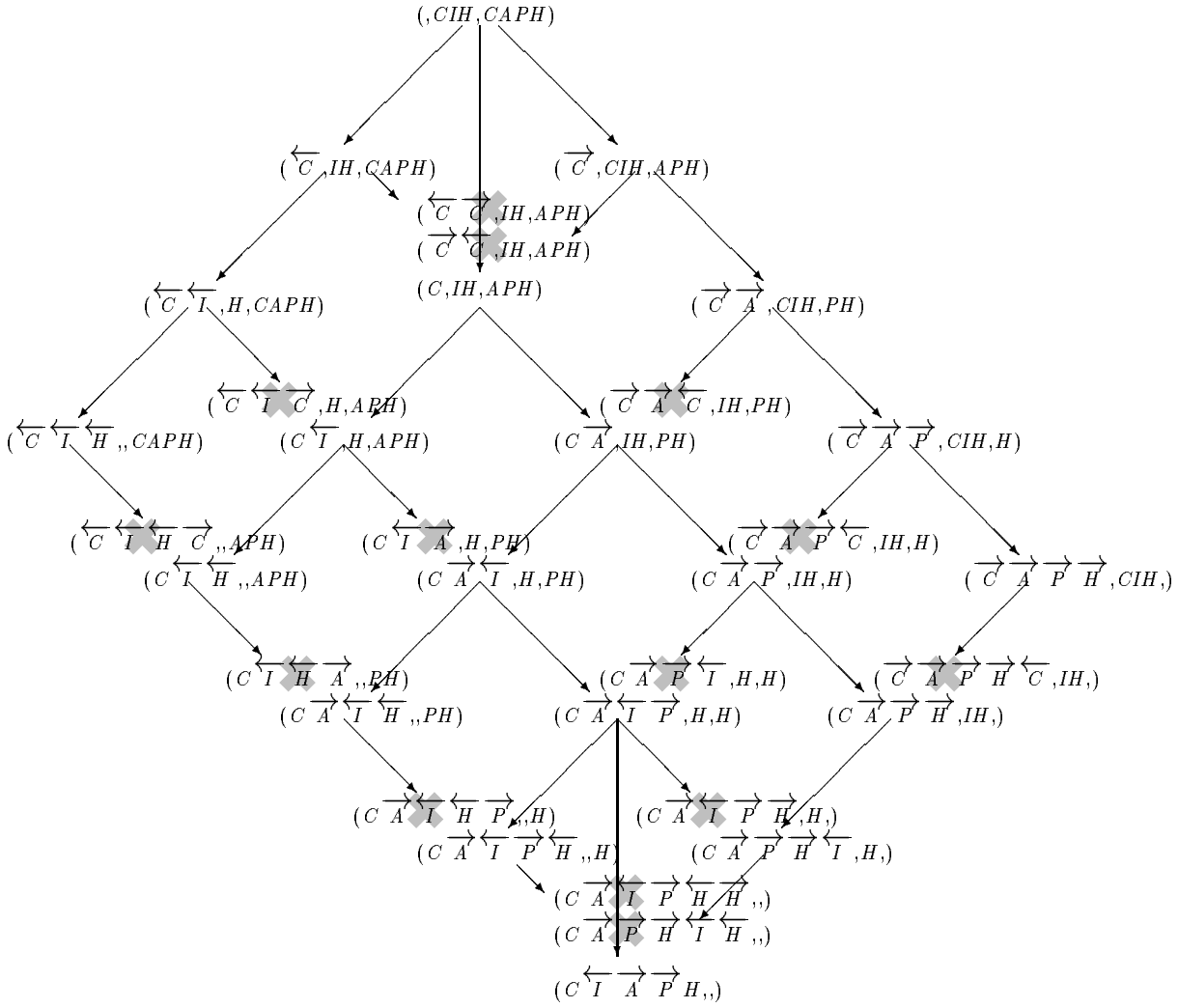
However, the table can be still seen in this method of dynamic programming. The table is an embedding into the partial space of solutions, and often the computation of the dynamic programming algorithm will mimic the steps taken to construct the table.

To illustrate this embedding, we consider an example of a computation for the above String Edit problem. The word Cih[*] is to be changed into Caph[†], and this diagram shows the partial solutions that are considered during the execution of the algorithm. The arrows represent applications of T, $Ret\ x$ is abbreviated by x, $Add\ x$ by $\overrightarrow{x}$ and

[*] γ Casseiopeiae

[†] β Casseiopeiae

Del x by $\overleftarrow{x}$. The partial solutions marked with a cross are those which get discarded from the thinning process.



The embedding of a rectangular structure can clearly be seen here in the above diagram of partial solutions. This structure corresponds to that of the table used to solve this problem with the standard dynamic programming algorithm.

One other consideration that is vital in the formal development of imperative programs is the termination of loops, and this has not yet been addressed.

When performing a loop *lim* P, it might be thought that a suitable requirement would be that *lim* P is total. Certainly this is necessary, but it is not sufficient. The totality of a relation only says that it is possible to produce a result, but does not guarantee that a result will be produced. The calculus of relations does not model termination accurately.

In practice, for each individual example, it has always been straightforward to check that the loop terminates, by using the well-known method of variants.

4 CONCLUSIONS

In this paper, we have seen a fresh way of looking at dynamic programming: while other views take the functional equation approach, with the Principle of Optimality, here we have chosen to use a monotonicity condition. The choice of which computation to avoid has been based on "which is better so far", not "what have we left to do".

The theorem presented here gives a very abstract view of a dynamic programming algorithm, best viewed as a search through the space of partial solutions, eliminating some branches from consideration. Using the traditional recursive method in this way results in a depth-first search of the space, while the method of keeping potential candidates results in a breadth-first search.

The high level abstraction of the algorithm gives freedom to do either of these search approaches, or a different search strategy entirely. Many possibilities can be explored, even with different approaches to the same problem. In the examples given here, simple search strategies of "sprout all" or "sprout the least developed" have been used, and the comparison relations chosen have been very straightforward, the obvious naive choices. The exploration of more sophisticated comparison relations is a fun pastime, stretching partway into the theory of branch-and-bound algorithms, resulting in many different algorithms, even variations on well-known algorithms for common problems.

The huge variety of possibilities for dynamic programming algorithms is not, I think, seen as clearly in the traditional approach based on the functional equation, nor even in recent approaches which focus on using the data structure inherent in the problem. Concentrating on data structures can lead to elegant algorithms, but this sometimes misses the possibility of alternative algorithms inspired by lateral thinking.

The work presented here is by no means practical: the theory is still very abstract and the actual program is a long way off. There is still a great deal of freedom of implementation, and the theory does not even begin to address efficiency considerations, although the use of data refinement is touched upon in Curtis (1996).

Nevertheless, this is a fresh approach, which reveals insight into this interesting subject, and it opens up possibilities for exploration of different dynamic programming algorithms for optimization problems.

REFERENCES

R. Bellman, H. Kagiwada, and R. Kalaba. Dynamic programming and an inverse problem in transport theory. *Computing*, 2:5–16, 1967.

Richard E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61, 1993.

P. Bonzon. Necessary and sufficient conditions for dynamic programming of combinatorial type. *Journal of the ACM*, 17(4):675–682, October 1970.

Sharon Curtis. A relational approach to optimization problems. D.Phil. thesis. Technical Monograph PRG-122, Computing Laboratory, Oxford, 1996.

Joseph G. Ecker and Michael Kupferschmid. *Introduction to Operations Research.* John Wiley and Sons, 1988.

Salah E. Elmaghraby. The concept of "state" in discrete dynamic programming. *Journal of Mathematical Analysis and Applications*, 29:523–557, 1970.

Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

Michael Held and Richard Karp. A dynamic programming approach to sequencing problems. *SIAM Journal for Applied Mathematics*, 10:196–210, 1962.

Paul Helman. The principle of optimality in the design of efficient algorithms. *Journal of Mathematical Analysis and Applications*, 119:97–127, 1986.

Paul Helman. A common schema for dynamic programming and branch and bound algorithms. *Journal of the ACM*, 36(1):97–128, January 1989.

Paul Helman and Arnon Rosenthal. A comprehensive model of dynamic programming. *SIAM Journal on Algebraic and Discrete Methods*, 6(2), April 1985.

T. Ibaraki. Solvable classes of discrete dynamic programming. *Journal of Mathematical Analysis and Applications*, 43:642–693, 1973.

Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal of Applied Mathematics*, 15(3):693–718, May 1967.

D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice and experience*, pages 1119 – 1184, 1981.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics-Doklady*, 10(8):707–710, February 1966.

G. R. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Groningen University, 1990.

L. G. Mitten. Composition principles for the synthesis of optimal multi-stage processes. *Operations Research*, 12, 1964.

O. de Moor. Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, 1992.

T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

David Sankoff and Joseph B. Kruskal, editors. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading, Mass., 1983.

Yu. A. Shreider. Automata and the problem of dynamic programming. *Problems of Cybernetics*, 5:31–48, 1961.

Robert A. Wagner and Michael J. Fisher. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, January 1974.

APPENDIX: PROOFS

Proof. of Theorem 1 (using definitions as in the theorem):

Let $M' = min\, R \cdot \mathsf{E}\, lim\, T$. Then

$$
\begin{array}{ll}
 & min\, R \cdot lim\, D \cdot \tau \subseteq M \\
\equiv & \quad \{\text{definition, (19)}\} \\
 & min\, R \cdot lim\, D \cdot \tau \subseteq min\, R \cdot \mathsf{E}\, lim\, T \cdot \tau \\
\Leftarrow & \quad \{\text{monotonicity, definition}\} \\
 & min\, R \cdot lim\, D \subseteq M' \\
\equiv & \quad \{(5)\} \\
 & lim\, D \subseteq min\, R \setminus M' \\
\Leftarrow & \quad \{(31), (35)\} \\
 & notdom\, D \cup min\, R \setminus M' \cdot D \subseteq min\, R \setminus M' \\
\equiv & \quad \{(1), (5)\} \\
 & min\, R \cdot notdom\, D \subseteq M' \\
 & \quad \wedge \; min\, R \cdot min\, R \setminus M' \cdot D \subseteq M' \\
\Leftarrow & \quad \{\text{assumption, (27)}\} \\
 & min\, R \cdot notdom\, (T \cdot \in) \subseteq M' \\
 & \quad \wedge \; min\, R \cdot min\, R \setminus M' \cdot D \subseteq M' \\
\Leftarrow & \quad \{(6)\} \\
 & min\, R \cdot notdom\, (T \cdot \in) \subseteq M' \\
 & \quad \wedge \; M' \cdot D \subseteq M' \\
\equiv & \quad \{(17), (30)\ \} \\
 & min\, R \cdot notdom\, (T \cdot \in) \subseteq lim\, T \cdot \in \\
 & \quad \wedge \; min\, R \cdot notdom\, (T \cdot \in) \cdot \ni \cdot (lim\, T)^\circ \subseteq R \\
 & \quad \wedge \; M' \cdot D \subseteq lim\, T \cdot \in \\
 & \quad \wedge \; M' \cdot D \cdot \ni \cdot (lim\, T)^\circ \subseteq R
\end{array}
$$

The first of these inequalities can be shown as follows:

$$
\begin{array}{ll}
 & min\, R \cdot notdom\, (T \cdot \in) \\
\subseteq & \quad \{(28\} \\
 & \in \cdot notdom\, (T \cdot \in) \\
\subseteq & \quad \{(24)\} \\
 & notdom\, T \cdot \in \\
\subseteq & \quad \{(31)\} \\
 & lim\, T \cdot \in .
\end{array}
$$

The second inequality proceeds thus

$$
\begin{array}{ll}
 & \mathit{min}\, R \cdot \mathit{notdom}\,(T \cdot \in) \cdot \ni \cdot (\mathit{lim}\ T)^\circ \\
\subseteq & \quad \{(25)\} \\
 & \mathit{min}\, R \cdot \ni \cdot \mathit{notdom}\ T \cdot (\mathit{lim}\ T)^\circ \\
= & \quad \{\text{reflexivity, } (29)\} \\
 & R \cdot \mathit{notdom}\ T \cdot (\mathit{lim}\ T)^\circ \\
= & \quad \{(32)\} \\
 & R \cdot \mathit{notdom}\ T \\
\subseteq & \quad \{(22)\} \\
 & R.
\end{array}
$$

The third inequality is shown as follows

$$
\begin{array}{ll}
 & M' \cdot D \\
= & \quad \{\text{definition}\} \\
 & \mathit{min}\, R \cdot \mathsf{E}\, \mathit{lim}\ T \cdot D \\
\subseteq & \quad \{28\} \\
 & \in \cdot \mathsf{E}\, \mathit{lim}\ T \cdot D \\
= & \quad \{(18)\} \\
 & \mathit{lim}\ T \cdot \in \cdot D \\
\subseteq & \quad \{\text{definition of } D\} \\
 & \mathit{lim}\ T \cdot \in \cdot \in\backslash\in \cdot \mathit{sprouts}\ T \\
\subseteq & \quad \{(6)\} \\
 & \mathit{lim}\ T \cdot \in \cdot \mathit{sprouts}\ T \\
\subseteq & \quad \{(39)\} \\
 & \mathit{lim}\ T \cdot (T \cdot \in\ \cup\ \in) \\
= & \quad \{\text{distribution of union over composition}\} \\
 & \mathit{lim}\ T \cdot T \cdot \in\ \cup\ \mathit{lim}\ T \cdot \in \\
\subseteq & \quad \{(34), \text{idempotence of union}\} \\
 & \mathit{lim}\ T \cdot \in,
\end{array}
$$

and the final inequality can be proved as follows:

$$
\begin{array}{ll}
 & M' \cdot D \cdot \ni \cdot (\mathit{lim}\ T)^\circ \\
\subseteq & \quad \{\text{assumption,}(3)\} \\
 & M' \cdot \ni \cdot (\mathit{lim}\ T)^\circ \cdot R \\
= & \quad \{\text{definition, } (17), (3)\} \\
 & \mathit{min}\, R \cdot \Lambda(\mathit{lim}\ T \cdot \in) \cdot (\mathit{lim}\ T \cdot \in)^\circ \cdot R
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \{(15)\} \\
& min\, R \cdot \ni \cdot R \\
\subseteq & \{(29), \text{transitivity of preorders}\} \\
& R.
\end{array}
$$

□

Proof. of Lemma 1:

$$
\begin{array}{ll}
& thin\, S \cdot sprouts\, T \cdot \ni \cdot (lim\, T)^\circ \\
\subseteq & \{\text{claim}\} \\
& thin\, S \cdot (\ni \ \cup \ \ni/\ni \cdot \Lambda T \cdot dom\, T) \cdot (lim\, T)^\circ \\
\subseteq & \{(48)\} \\
& (\ni \cdot S)/\ni \cdot (\ni \ \cup \ \ni/\ni \cdot \Lambda T \cdot dom\, T) \cdot (lim\, T)^\circ \\
\subseteq & \{\text{union, (6), (8)}\} \\
& \ni \cdot S \cdot (lim\, T)^\circ \\
& \ \cup \ (\ni \cdot S)/\ni \cdot \Lambda T \cdot dom\, T \cdot (lim\, T)^\circ \\
= & \{(31), (33)\} \\
& \ni \cdot S \cdot (lim\, T)^\circ \\
& \ \cup \ (\ni \cdot S)/\ni \cdot \Lambda T \cdot T^\circ \cdot (lim\, T)^\circ \\
= & \{(15),(6)\} \\
& \ni \cdot S \cdot (lim\, T)^\circ \ \cup \ \ni \cdot S \cdot (lim\, T)^\circ \\
= & \{\text{idempotence, assumption}\} \\
& \ni \cdot (lim\, T)^\circ \cdot R
\end{array}
$$

The claim is that $sprouts\, T \cdot \ni \ \subseteq \ \ni \ \cup \ \ni/\ni \cdot \Lambda T \cdot dom\, T$ and is proved

$$
\begin{array}{ll}
& sprouts\, T \cdot \ni \\
= & \{(36)\} \\
& cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom T \times id) \cdot luni^\circ \cdot \ni \\
= & \{(40);\ (3)\} \\
& cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom T \times id) \cdot (outr^\circ \cdot \ni \ \cup \ outl^\circ \cdot \ni) \\
= & \{\text{distribution of union}\} \\
& cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom T \times id) \cdot outr^\circ \cdot \ni \\
& \ \cup \ cup \cdot (\mathsf{E}\, T \cdot \mathsf{P}\, dom T \times id) \cdot outl^\circ \cdot \ni \\
\subseteq & \{(12), (13)\} \\
& cup \cdot outr^\circ \cdot \ni \ \cup \ cup \cdot outl^\circ \cdot \mathsf{E}\, T \cdot \mathsf{P}\, dom T \cdot \ni \\
\subseteq & \{(41), (42)\} \\
& \ni/\ni \cdot \ni \ \cup \ \ni/\ni \cdot \mathsf{E}\, T \cdot \mathsf{P}\, dom T \cdot \ni
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \quad \{(6), (20), (5)\} \\
& \ni \ \cup \ \ni / \ni \cdot \mathsf{E}\, T \cdot \ni \cdot \mathit{dom}\ T \\
= & \quad \{(9), (3), (18)\} \\
& \ni \ \cup \ \ni / (\ni \cdot T^\circ) \cdot \ni \cdot \mathit{dom}\ T \\
\subseteq & \quad \{(7)\} \\
& \ni \ \cup \ \ni / T^\circ \cdot \mathit{dom}\ T \\
= & \quad \{(15), (3), (9)\} \\
& \ni \ \cup \ \ni / \ni \cdot \Lambda T \cdot \mathit{dom}\ T
\end{array}
$$

□

Proof. of Lemma 2

$$
\begin{array}{ll}
& \mathit{lim}\ T \cdot S^\circ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\equiv & \quad \{(4)\} \\
& \mathit{lim}\ T \ \subseteq \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \\
\Leftarrow & \quad \{(31), (35)\} \\
& \mathit{notdom}\ T \ \cup \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \ \subseteq \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \\
\equiv & \quad \{(1)\} \\
& \mathit{notdom}\ T \ \subseteq \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \ \subseteq \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \\
\equiv & \quad \{(4), (3)\} \\
& S \cdot \mathit{notdom}\ T \ \subseteq \ (\mathit{lim}\ T)^\circ \cdot R \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \cdot S^\circ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\equiv & \quad \{\text{assumption}\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \cdot S^\circ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\equiv & \quad \{(23), \text{distribution of union}\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \cdot S^\circ \cdot \mathit{dom}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot T \cdot S^\circ \cdot \mathit{notdom}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \quad \{\text{assumptions}, (3)\} \\
& (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot S^\circ \cdot T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ (R^\circ \cdot \mathit{lim}\ T)/S^\circ \cdot S^\circ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \quad \{(6)\} \\
& R^\circ \cdot \mathit{lim}\ T \cdot T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
& \ \wedge \ R^\circ \cdot \mathit{lim}\ T \ \subseteq \ R^\circ \cdot \mathit{lim}\ T \\
\Leftarrow & \quad \{\text{monotonicity}; (31), (35)\} \\
& \mathit{true}.
\end{array}
$$

□

Proof. of Lemma 3: this is a corollary of Lemma 2. For the second condition,

$$
\begin{array}{ll}
 & \mathit{notdom}\ T \cdot S \cdot T^\circ \\
= & \quad \{\text{assumption}\} \\
 & S \cdot \mathit{notdom}\ T \cdot T^\circ \\
= & \quad \{(26)\} \\
 & \emptyset \\
\subseteq & \quad \{\text{empty relation}\} \\
 & S.
\end{array}
$$

For the third condition,

$$
\begin{array}{ll}
 & S \cdot \mathit{notdom}\ T \\
= & \quad \{\text{property of coreflexives}\} \\
 & S \cdot \mathit{notdom}\ T \cdot \mathit{notdom}\ T \\
= & \quad \{\text{assumption}\} \\
 & \mathit{notdom}\ T \cdot S \cdot \mathit{notdom}\ T \\
= & \quad \{\text{property of coreflexives}\} \\
 & \mathit{notdom}\ T \cdot \mathit{notdom}\ T \cdot S \cdot \mathit{notdom}\ T \\
\subseteq & \quad \{\text{assumption}\} \\
 & \mathit{notdom}\ T \cdot R \\
\subseteq & \quad \{(31)\} \\
 & (\mathit{lim}\ T)^\circ \cdot R.
\end{array}
$$

□