Partitions Revisited

S. Curtis
Somerville College, Oxford

April 25th, 1993

Submitted in support of an application for transfer to D.Phil. status.

Oxford University Computing Laboratory
Programming Research Group

Partitions Revisited

S. Curtis

Abstract

Problems involving list partitions are found in many areas of computer science. This paper states theorems about programs that use strategies such as dynamic programming or greedy strategies to solve optimization problems, and applies the theorems to the solving of partition problems. The reasoning is in an equational style, using a calculus of relations and associated laws.

Contents

1 Introduction

Several years ago, Bird and de Moor wrote a paper on the subject of List Partitions [BM92c], which presented theorems applying the strategy of dynamic programming and several greedy strategies to partition problems. The paper was written in the functional programming style of the Bird-Meertens formalism, common in many papers concerning lists at the time. However, there was a growing dissatisfaction with the functional framework for optimization problems, as the optimizing component of such a problem can be better represented as a relation. From this motivation, a relational framework has been created (and is indeed still evolving) especially for use with optimization problems. In several papers, including [BM92d], [BM92b] and [BM92a], this relational calculus is presented, along with several theorems useful in the solving of optimization problems, including theorems using the dynamic programming strategy and greedy strategies.

This paper updates the work of the List Partitions paper [BM92c] into a relational setting, using this calculus. Category theory provides the backdrop for the relational calculus, and concepts such as catamorphisms and datatypes from initial algebras are heavily used. The theorems concerning dynamic programming and greedy algorithms are also in the style of category theory, and from these, equational reasoning is used to apply the concepts to partition problems. Throughout the paper, efficiency is sought after and examples are given at the end of each section.

The rest of this paper is organized as follows. The following subsection introduces and gives examples of partition problems. Section 2 gives an introduction to the calculus of relations, which is used throughout the rest of the paper. Section 3 presents the Dynamic Programming Theorem of de Moor, then shows how this theorem is applicable to partition problems. Section 4 presents the Greedy Theorem of Bird and de Moor, then, similarly to the previous section, applies the theorem to partition problems. Section 5 considers further optimizations which can be made if a partition problem satisfies additional conditions, and shows how to express the solutions to such problems as catamorphisms. Section 6 generalizes the concept of a partition problem, and demonstrates the use of the earlier theorems with the generalizations. Section 7 concludes with some comments on the paper, and suggests other directions for research using the relational calculus.

1.1 Partition Problems

A partition of a list is a division of that list into non-empty contiguous segments. For example, the set of partitions of the list $[5, 6, 7]$ is

$$\{[[5], [6], [7]],\ [[5, 6], [7]],\ [[5], [6, 7]],\ [[5, 6, 7]]\}$$

Partition problems occur frequently in many branches of computer science; here are some examples of partition problems:

Text formatting:
A common problem in word processing programs is to decide how to format a list of words representing a paragraph. The decision of where to put the line breaks in the paragraph can be represented as a partition of the list of words, where each component of the partition represents a line of text. Paragraphs to be considered should satisfy suitable conditions such as the paragraph actually fitting onto the page, and an optimal paragraph is then selected by comparing the "badness" of the possible paragraphs, where such quantities as margin widths and white space are taken into account in calculating "badness". Knuth and Plass give a thorough discussion of the desirable qualities of paragraphs in [KP81].

Sorting by natural merging:
One method to sort a list into ascending order is to first partition the list into contiguous segments, where each segment is in ascending order, and as few segments as possible are in the partition. The components of the partition are then merged together to produce a single list in ascending order. Also see [Wir76].

Coding with respect to a dictionary:
One way of compressing a piece of text is to divide the text into portions, and replace each portion with a pointer into some standard dictionary. An optimal compression for a piece of text may be obtained by choosing to partition the text in such a way as to use the least number of pointers, i.e. by choosing the shortest partition. In [Wag73], such a method is used to minimize the space utilised by a compiler, by replacing common phrases used in error messages by pointers.

The bank cash flow problem:
A bank knows in advance what sequence of cash withdrawals and deposits it is going to experience in the next month, and wishes to make its cash flow arrangements. For any cash withdrawal, there must be enough money in the bank at the time to supply the customer, and at no time must there be too much money in the bank, for security reasons, and therefore this bank uses a van to bring necessary money and take away any surplus. To be as economical as possible, it is desired that the van visit the bank the minimum number of times. This problem can be expressed in terms of partitions, and will be used as an example later on.

The problems above can all be specified in the general form

$$M \quad = \quad \mathit{min}\, R \cdot (\mathit{all}\; p) \triangleleft \cdot \mathit{parts},$$

where *parts* returns the set of all partitions of a list, $(\mathit{all}\; p)\triangleleft$ operating on that set retains just the partitions whose components all satisfy some predicate p, and $\mathit{min}\, R$ selects a minimum partition from those. This specification can be implemented in a functional language, where $\mathit{min}\, R$ returns some partition that is a minimum with respect to R, but such an implementation is not very efficient, as the number of partitions of a list of length $n+1$ is 2^n.

Instead we will manipulate the above expression to find more efficient ways to solve partition problems. However *min* is not a function, because a set may have none, one or several minimum elements; *min* is a relation, and so we need a calculus of relations suitable for manipulating such expressions.

2 A Calculus of Relations

2.1 Relations

We shall be working in two categories: the category *Fun*, which has sets as objects and total functions as arrows, and the category *Rel* which has sets as objects and relations as arrows. Relations will be denoted by upper-case letters R,S,T,..., and relations which are also functions denoted by lower-case letters $f, g, h\ldots$. A relation R with source A and target B (written $R : B \leftarrow A$) is a subset of the cartesian product $(A \times B)$ and we shall write $b\,R\,a$ for $(a, b) \in R$. We denote composition of relations by $\cdot$. If $R : B \leftarrow A$ and $S : C \leftarrow B$ then $S \cdot R : C \leftarrow A$ and we have the usual meaning of composition:

$$c(S \cdot R)a \;\; = \;\; \exists b : c\,S\,b \;\wedge\; b\,R\,a.$$

The identity relation for any set A is denoted by $id_A : A \leftarrow A$ and the subscript is usually omitted, as the set can usually be deduced from the context. The empty relation 0 is the smallest relation of type $B \leftarrow A$, and is a zero of composition:

$$R \cdot 0 = 0 = 0 \cdot R.$$

The largest relation of type $B \leftarrow A$ is denoted by Π, which is also the product $A \times B$. The converse of a relation R is R°, and is defined by:

$$b\,R^\circ\,a \;\; = \;\; a\,R\,b.$$

Indeed, the converse functor which takes sets to themselves and takes relations to their converse is contravariant, because

$$id^\circ = id \quad \text{and} \quad (R \cdot S)^\circ = S^\circ \cdot R^\circ .$$

Also, the converse functor is its own inverse, as $(R^\circ)^\circ = R$. A relation $R : B \leftarrow A$ is said to be *simple* if

$$R \cdot R^\circ \;\; \subseteq \;\; id.$$

Translated, this says that for any $a \in A$, there is at most one $b \in B$ such that $b\,R\,a$. Simple relations are also known as partial functions. R is said to be *entire* (or *total*) if

$$id \;\; \subseteq \;\; R^\circ \cdot R,$$

which translated, says that for any $a \in A$, there is at least one $b \in B$ such that $b\, R\, a$. A relation is a *function* when it is both simple and entire, so that

$$b\, f\, a \quad \equiv \quad f\, a = b.$$

Alternatively, f is a function when for all R, S:

$$R \cdot f^{\circ} \subseteq S \quad \equiv \quad R \subseteq S \cdot f,$$

or equivalently

$$f \cdot R \subseteq S \quad \equiv \quad R \subseteq f^{\circ} \cdot S.$$

Any two relations R, S of the same type have an *intersection* $R \cap S$, which is defined with the equivalence

$$T \subseteq (R \cap S) \quad \equiv \quad (T \subseteq R) \wedge (T \subseteq S).$$

The dual construction is the *union* of two relations of the same type, and is defined with the equivalence

$$(R \cup S) \subseteq T \quad \equiv \quad (R \subseteq T) \wedge (S \subseteq T).$$

There are various laws concerning intersection and union. Intersection distributes over union, and vice-versa; composition with a simple relation distributes over intersection

$$(R \cap T) \cdot S \quad = \quad (R \cdot S) \cap (T \cdot S),$$

but for composition with an arbitrary relation the equality is weakened to $(\subseteq)$. For union however, there is no such restriction on S

$$(R \cup T) \cdot S \quad = \quad (R \cdot S) \cup (T \cdot S).$$

The *Modular Law*, also known as Dedekind's Rule, relates intersection and converse:

$$R \cdot S \cap T \quad \subseteq \quad (R \cap T \cdot S^{\circ}) \cdot S.$$

(Composition binds more tightly than any other binary operator.)

For two relations R and S with a common source their *right quotient* is characterized by the following universal property

$$T \subseteq R/S \equiv T \cdot S \subseteq R.$$

Expressed as a predicate instead, it can be defined

$$c(R/S)b = \forall a : bSa \Rightarrow cRa.$$

Similarly, any two relations with a common target have a *left quotient* which is characterized by the universal property,

$$T \subseteq S\backslash R \equiv S \cdot T \subseteq R,$$

and has the translation into a predicate:

$$b(S\backslash R)c = \forall a : aSb \Rightarrow aRc.$$

A relation S is a *preorder* if it is both reflexive and transitive:

$$id \subseteq S \text{ and } S \cdot S \subseteq S.$$

There is exists a smallest preorder containing a relation R, denoted by R^*, which is the *reflexive transitive closure* of R, and is characterized by the property that for any preorder S,

$$R \subseteq S \equiv R^* \subseteq S.$$

The *transitive closure* of a relation R is denoted by R^+ and can be defined similarly. It has the property that

$$R^+ = R^* \cdot R = R \cdot R^* \cdot$$

The following lemma concerning preorders appears in [BM92a], and we will find it useful later on:

Lemma 1 *Let S be a preorder. Then*

$$R^* \cdot T \subseteq T \cdot S \equiv R \cdot T \subseteq T \cdot S.$$

Proof By the universal property for quotients, it is sufficient to prove that

$$R^* \subseteq (T \cdot S)/T \;\equiv\; R \subseteq (T \cdot S)/T,$$

which follows from the definition of a preorder, given that $(T \cdot S)/T$ itself is a preorder. To establish this, we need to prove reflexivity and transitivity. Reflexivity is proved as follows:

$$\begin{array}{ll}
 & id \subseteq (T \cdot S)/T \\
\equiv & \quad \{\text{universal property of quotient}\} \\
 & T \subseteq T \cdot S \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & id \subseteq S \\
\equiv & \quad \{\text{S is a preorder}\} \\
 & true,
\end{array}$$

and for transitivity:

$$\begin{array}{ll}
 & (T \cdot S)/T \cdot (T \cdot S)/T \subseteq (T \cdot S)/T \\
\equiv & \quad \{\text{universal property of quotient}\} \\
 & (T \cdot S)/T \cdot (T \cdot S)/T \cdot T \subseteq T \cdot S \\
\Leftarrow & \quad \{\text{property of quotient: } P/Q \cdot Q \subseteq P\} \\
 & (T \cdot S)/T \cdot T \cdot S \subseteq T \cdot S \\
\Leftarrow & \quad \{\text{property of quotient as above}\} \\
 & T \cdot S \cdot S \subseteq T \cdot S \\
\Leftarrow & \quad \{\text{S is transitive}\} \\
 & T \cdot S \subseteq T \cdot S \\
\equiv & \quad \{\text{reflexivity of } \subseteq\} \\
 & true.
\end{array}$$

$\Box$.

2.2 Relators

We are interested in the functors of type $Rel \leftarrow Rel$, and in particular what they have to do with functors of type $Fun \leftarrow Fun$. One desirable property for $\mathsf{F} : Rel \leftarrow Rel$ is that of monotonicity. F is monotonic when

$$R \subseteq S \quad \Rrightarrow \quad \mathsf{F}R \subseteq \mathsf{F}S.$$

The following theorem states the important properties of monotonic functors. A detailed proof can be found in [CKW89].

Proposition 1 *Suppose that* $\mathsf{F} : Rel \leftarrow Rel$ *is monotonic. Then*
• F *preserves functions, i.e. for all* f *in Fun,* $\mathsf{F}f$ *is a function.*
• F *preserves converse, i.e.* $\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$.
• F *is characterized by its action on functions, ie if* $\mathsf{G} : Rel \leftarrow Rel$ *is another monotonic functor, then the statement*

$$\mathsf{F}f = \mathsf{G}f \quad \textit{for all } f \textit{ in Fun}$$

is equivalent to $\mathsf{F} = \mathsf{G}$.

So any monotonic functor of type $Rel \leftarrow Rel$ is a unique extension of a functor of type $Fun \leftarrow Fun$. When a functor F on Fun has an extension to Rel, we will call it a *relator*, and the same symbol F will be used for both the functor and its extension.

Many functors of practical use are relators. One example is the product functor. Its extension is given by

$$R \times S \;\; = \;\; outl^\circ \cdot R \cdot outl \;\cap\; outr^\circ \cdot S \cdot outr,$$

where $outl : A \leftarrow (A \times B)$ and $outr : B \leftarrow A \times B$ are the familiar left and right projection functions on pairs. However, the relator $\times$ does *not* fit the categorical definition of product in Rel.

Another relator is the extension of the coproduct functor, which is given by

$$R + S \;\; = \;\; (inl \cdot R \cdot inl^\circ) \cup (inr \cdot S \cdot inr^\circ),$$

where $inl : A + B \leftarrow A$ and $inr : A + B \leftarrow B$ are the familiar coproduct injections. However, here $+$ *does* fit the definition of a categorical coproduct, and since Rel is isomorphic to its own dual, $+$ fits the definition of a categorical product also.

Two other operators that are concerned with products and coproducts are the *split* and *join* operators. The split of two relations $R : B \leftarrow A$ and $S : C \leftarrow A$ is $\langle R, S\rangle : (B \times C) \leftarrow A$ which is defined

$$\langle R, S\rangle \;=\; outl^\circ \cdot R \;\cap\; outr^\circ \cdot S.$$

The join of two relations $R : C \leftarrow A$ and $S : C \leftarrow B$ is $[R, S] : C \leftarrow (A+B)$, which is defined

$$[R, S] \;=\; R \cdot inl^\circ \cup\; S \cdot inr^\circ\,.$$

Other functors that are also relators are the identity functor, the constant functors, and the polynomial functors. The polynomial functors are defined inductively by

- The identity functor I and the constant functors K_A are polynomial.
- If F and G are polynomial, then so too are their composition FG, their sum $\mathsf{F} + \mathsf{G}$ and their product $\mathsf{F} \times \mathsf{G}$, defined by

$$\begin{aligned}(\mathsf{F} + \mathsf{G})f &= \mathsf{F}f + \mathsf{G}f\\ (\mathsf{F} \times \mathsf{G})f &= \mathsf{F}f \times \mathsf{G}f.\end{aligned}$$

2.3 Powersets

Recall that we are working with relations as subsets of cartesian products. An alternative way to define relations is to regard them as set-valued functions. To exploit this isomorphism between relations and set-valued functions, we introduce the *power transpose* operator. Let $R : B \leftarrow A$, and let $\mathsf{P}A$ be the set which in set theory is known as the *powerset* of A. Then the power transpose operator Λ takes a relation of type $B \leftarrow A$ and returns a function of type $\mathsf{P}B \leftarrow A$, and its defining property is that for any such R, and for any function $f : \mathsf{P}B \leftarrow A$ we have

$$f = \Lambda\, R \;\equiv\; \in \cdot f \;=\; R,$$

where $\in$ is the membership relation of type $B \leftarrow \mathsf{P}B$. This property characterizes $(\mathsf{P}A, \Lambda, \in)$ up to isomorphism. The set comprehension for Λ is given by

$$(\Lambda\, R)a = \{\, b \mid b\, R\, a \,\}.$$

Readers familiar with category theory will recognize that the power transpose defines a right adjoint of the inclusion functor from *Fun* to *Rel*.

As we have defined P on sets, by defining

$$(\mathsf{P}f) \;=\; \Lambda\,(f\cdot\in),$$

we get a functor $\mathsf{P} : Fun \leftarrow Fun$. P is actually a relator, and its extension is given by

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

which translates to

$$y(\mathsf{P}R)x \;=\; (\forall b\in y : \exists a\in x : b\;R\;a)\wedge(\forall a\in x : \exists b\in y : b\;R\;a).$$

This corresponds to the *Egli–Milner* ordering induced by R. De Moor's thesis [Moo92] gives a fuller discussion of this relator.

The powerset functor is not the same as the *existential image* functor $\mathsf{E} : Rel \leftarrow Rel$ given by

$$\begin{aligned}\mathsf{E}A &= \mathsf{P}A\\ \mathsf{E}R &= \Lambda\,(R\cdot\in),\end{aligned}$$

because P returns relations and is monotonic, whereas E returns functions and is not monotonic. A law of composition concerning E and Λ is

$$\Lambda\,(R\cdot S) \;=\; \mathsf{E}R\cdot\Lambda\,S.$$

Many other operators can be defined in terms of power transpose and existential image. Filter is one such operator. Given a boolean-valued predicate p, the relation that tests whether p holds is given by

$$b(p?)a \;=\; (p\;a)\;\wedge\;(b=a),$$

and then we can define $p\,\triangleleft : \mathsf{P}A \leftarrow \mathsf{P}A$ by $p\,\triangleleft\, = \mathsf{E}(p?)$.

2.4 F-algebras and catamorphisms

Let $\mathcal{C}$ be a category, and let F be an endofunctor on $\mathcal{C}$. Then an F-algebra is an arrow $f : A \leftarrow \mathsf{F}A$, and A is called the *carrier* of the algebra. Let $g : B \leftarrow \mathsf{F}B$ be another F-algebra. A *homomorphism* from f to g is an arrow $h : B \leftarrow A$ such that

$$h \cdot f \quad = \quad g \cdot \mathsf{F}h.$$

Clearly identity arrows on the carrier sets of F-algebras are homomorphisms, and that composition of two F-homomorphisms is another F-homomorphism follows easily from the above equation and composition of arrows within $\mathcal{C}$. Thus another category can be formed with F-algebras of $\mathcal{C}$ as its objects and F-homomorphisms as its arrows.

For some functors F, this category of F-algebras has an initial object, that is, there exists an F-algebra (α, say) that for any other F-algebra β, there is exactly one F-homomorphism from α to β. This initial algebra is unique up to isomorphism. Initial algebras are used to define data types, and there exist initial F-algebras for many functors, but not all. The functor P does not have an initial algebra, but many functors on *Fun* do, for example polynomial functors on *Fun*. Malcolm in his thesis [Mal90] describes the (fairly weak) conditions on F for it to have an initial algebra. An initial F-algebra is often denoted by α, and α is an isomorphism. The unique homomorphism from an initial F-algebra to another F-algebra β is denoted by $(\![\beta]\!)_{\mathsf{F}}$ (though the F is often omitted where the context is clear) and $(\![\beta]\!)$ is characterized by the universal property

$$h \cdot \alpha = \beta \cdot \mathsf{F}h \quad \equiv \quad h = (\![\beta]\!).$$

These unique homomorphisms $(\![\beta]\!)$ are called *catamorphisms*, from the greek *cata*, meaning "according to" and *morphe*, meaning "shape" or "structure". Catamorphisms have useful calculational properties, e.g. catamorphisms on the data type of lists are the familiar fold operators known to functional programmers (see [BW88]).

Of great importance to us is the existence of these initial algebras in *Rel*. Fortunately we have

Lemma 2 *Let $\mathsf{F} : \mathit{Fun} \leftarrow \mathit{Fun}$ be a relator and have an initial algebra α. Then α is also an initial algebra of $\mathsf{F} : \mathit{Rel} \leftarrow \mathit{Rel}$.*

A proof may be found in [Moo93]. So we have that all polynomial functors have initial algebras in *Rel*, which will be adequate for our purposes.

Also, not only can relations can be compared by testing for equality (as functions are in *Fun*), but relations can also be compared by testing for inclusion. From the universal property for catamorphisms above and the Knaster-Tarski fixpoint theorem (e.g. see [BVvdW91]) we also have

Proposition 2 *If α is an initial* F*-algebra, then*

$$\begin{array}{lcl} R \cdot \alpha = S \cdot \mathsf{F}R & \equiv & R = ([S]) \\ R \cdot \alpha \subseteq S \cdot \mathsf{F}R & \Rightarrow & R \subseteq ([S]) \\ R \cdot \alpha \supseteq S \cdot \mathsf{F}R & \Rightarrow & R \supseteq ([S]). \end{array}$$

A theorem which allows us to combine relations and catamorphisms is the Promotion Theorem (also see [BBM+91]):

Theorem 1 *If α is the initial algebra of the relator* F*, and*

$$U \cdot V \quad = \quad W \cdot \mathsf{F}U \quad (mod\, \mathsf{F}([V])),$$

then

$$U \cdot ([V]) \quad = \quad ([W]).$$

Straightforward application of the above proposition easily proves this theorem, and also yields that the two equality signs can either be replaced with two $\subseteq$ or two $\supseteq$ signs.

2.5 Representing Partitions

In this section we illustrate the idea of F-algebras with some examples that are relevant to partitions, and we introduce some relations that we shall need.

The data-type of *snoc-lists* can be defined in a suitable functional programming language by

$$snoclist \ ::= \ empty \mid snoc\,(snoclist, E),$$

where *empty* stands for the empty list, and E is the type of the elements of the list. Snoc-lists can also be defined as the initial algebra $[empty, snoc]$ of the functor

$$\begin{aligned} \mathsf{F}A &= \top + (A \times E) \\ \mathsf{F}f &= id + (f \times id), \end{aligned}$$

where $\top$ is the terminator of *Fun*. The set of all snoc-lists over E is denoted by E^*.

A similiar data-type is the type of non-empty lists. This can be defined as the initial algebra $[single, snoc]$ of the functor

$$\begin{aligned} \mathsf{F}A &= E + (A \times E) \\ \mathsf{F}f &= id + (f \times id), \end{aligned}$$

and the set of non-empty lists over E is denoted by E^+. Note that the same name *snoc* was used in both definitions. This need not cause confusion, as type-checking will identify which *snoc* was used.

An operator which we will use frequently when considering partition problems is the concatenation operator $cat : E^+ \leftarrow (E^* \times E^+)$, defined by

$$\begin{aligned} cat = {} & snoc \cdot (id \times single^\circ) \\ & \cup\ snoc \cdot (cat \times id) \cdot assoc \cdot (id \times snoc^\circ), \end{aligned}$$

where *assoc* is the natural isomorphism from $A \times (B \times C)$ to $(A \times B) \times C$. This is in accordance with the usual functional programming definition for *cat* as the two equations

$$\begin{aligned} x +\!\!+ [a] &= x +\!\!+\!< a \\ x +\!\!+ (y +\!\!+\!< a) &= (x +\!\!+ y) +\!\!+\!< a, \end{aligned}$$

where *cat* is written as $+\!\!+$, and *snoc* is written as $+\!\!+\!<$.

A partial function on lists which takes a list and removes its last element is *Init*, which is defined by

$$Init = outl \cdot snoc^\circ,$$

and in fact this defines two relations, depending on which *snoc* is used. Using the above definitions we can show that

$$Init \cdot cat \subseteq cat \cdot (id \times Init) \cup outl \cdot (id \times single^\circ).$$

This can be proved in the following way:

$$
\begin{array}{ll}
& \mathit{Init} \cdot \mathit{cat} \\
= & \quad \{\text{definition of } \mathit{Init}\} \\
& \mathit{outl} \cdot \mathit{snoc}^\circ \cdot \mathit{cat} \\
= & \quad \{\text{definition of } \mathit{cat}\text{; composition distributes over union}\} \\
& \mathit{outl} \cdot \mathit{snoc}^\circ \cdot \mathit{snoc} \cdot (\mathit{id} \times \mathit{single}^\circ) \\
& \cup\ \mathit{outl} \cdot \mathit{snoc}^\circ \cdot \mathit{snoc} \cdot (\mathit{cat} \times \mathit{id}) \cdot \mathit{assoc} \cdot (\mathit{id} \times \mathit{snoc}^\circ) \\
\subseteq & \quad \{\mathit{snoc} \text{ is simple}\} \\
& \mathit{outl} \cdot (\mathit{id} \times \mathit{single}^\circ) \\
& \cup\ \mathit{outl} \cdot (\mathit{cat} \times \mathit{id}) \cdot \mathit{assoc} \cdot (\mathit{id} \times \mathit{snoc}^\circ),
\end{array}
$$

and we have that

$$
\begin{array}{ll}
& \mathit{outl} \cdot (\mathit{cat} \times \mathit{id}) \cdot \mathit{assoc} \cdot (\mathit{id} \times \mathit{snoc}^\circ) \\
= & \quad \{\text{naturality of } \mathit{outl}\} \\
& \mathit{cat} \cdot \mathit{outl} \cdot \mathit{assoc} \cdot (\mathit{id} \times \mathit{snoc}^\circ) \\
= & \quad \{\text{property of } \mathit{outl} \text{ and } \mathit{assoc}\} \\
& \mathit{cat} \cdot (\mathit{id} \times \mathit{outl} \cdot \mathit{snoc}^\circ) \\
= & \quad \{\text{definition of Init}\} \\
& \mathit{cat} \cdot (\mathit{id} \times \mathit{Init}).
\end{array}
$$

This fact about *cat* and *Init* will be useful later in this paper. Another useful observation about *Init* is that the relation *Pre*, which relates a list to a prefix of itself, can be defined by

$$\mathit{Pre} \;=\; \mathit{Init}^*.$$

Partitions themselves are a certain variety of lists, and are elements of E^{+*}, that is, partitions are (possibly empty) lists of non-empty lists of elements of some type E. Thus we define the data-type of partitions as the initial algebra $[\mathit{empty}, \mathit{snoc}]$ of the functor

$$
\begin{array}{lll}
\mathsf{F}A & = & \top + (A \times E^+) \\
\mathsf{F}f & = & \mathit{id} + (f \times \mathit{id}).
\end{array}
$$

2.6 Monotonicity

When dealing with F-algebras and minimization, it is useful to know how the two interact together, and the property of monotonicity is often used to express this.

Let $S : A \leftarrow \mathsf{F}A$ be a relation and let F be a relator. S is said to be *monotonic* with respect to a relation $R : A \leftarrow A$ when

$$S \cdot \mathsf{F}R \cdot S^\circ \quad \subseteq \quad R.$$

For an F-algebra $f : A \leftarrow \mathsf{F}A$, as f is a function, this condition can be expressed more symmetrically as:

$$f \cdot \mathsf{F}R \quad \subseteq \quad R \cdot f.$$

For example, take the functor F, where

$$\begin{array}{rcl} \mathsf{F}A & = & A \times A \\ \mathsf{F}f & = & f \times f. \end{array}$$

Addition $plus : \mathbb{N} \leftarrow (\mathbb{N} \times \mathbb{N})$ is an algebra of this functor. Setting f to $plus$, and R to Leq $(\leq)$, the above definition translates to

$$a = a_0 + a_1 \;\wedge\; (a_0 \leq b_0) \;\wedge\; (a_1 \leq b_1) \quad \Rightarrow \quad a \;\leq\; b_0 + b_1.$$

This corresponds to the usual definition of monotonicity of addition on the natural numbers.

2.7 Minimization

In order to select optimal partitions, we need to be able to select minimum elements with respect to some ordering R. To do this, let R be of the type $A \leftarrow A$. The relation $min\,R : A \leftarrow \mathsf{P}A$ relates a set of elements to its minimum elements with respect to R and is defined by

$$min\,R \quad = \quad \in \cap\; (R/\ni).$$

Written at the point level, this translates to

$$a(min\,R)x \quad = \quad a \in x \;\wedge\; (\forall b :\; b \in x \Rightarrow aRb).$$

The definition of *min* is in accordance with the usual definition of minimum from set theory. Also, we are not restricting ourselves by using minimum elements, as $max\, R = min\, R^\circ$.

As the earlier definitions of intersection and quotient have been given in terms of universal properties, these can be used to give a universal property for *min*:

$$T \subseteq min\, R \cdot \Lambda\, S \quad \equiv \quad (T \cdot S^\circ \subseteq R) \wedge (T \subseteq S).$$

This agrees with our earlier definition of *min* when setting S to $\in$ and using the fact that $\Lambda{\in} = id$.

In order for *min* to be computationally useful, we would wish that all non–empty sets of elements have at least one minimum element under R. That is, we want R to be a *well–founded* relation. Well–foundedness of R can be expressed as

$$\in \quad \subseteq \quad R^\circ \cdot min\, R,$$

which translated, says that if $a \in x$, then there is an element b such that $b(min\, R)x$ and $b\, R\, a$.

A common R to choose when dealing with partitions is the relation *Len*, which compares the lengths of two lists, and which is well-founded.

3 Dynamic Programming

In this section, we give a theorem of De Moor [Moo92] which can be used to apply dynamic programming to list partition problems. Throughout this section we will assume that F is a polynomial functor on *Rel*, with the operator of its initial algebra α. It is well-known that dynamic programming is governed by the principle of optimality, where optimal solutions to a problem are composed of optimal solutions to subproblems, and in this theorem, the principle of optimality is expressed as a monotonicity condition on α.

Theorem 2 *Let $M \;=\; min\, R \cdot \; \Lambda\, ([P])^{\circ}$, where R is a preorder. If α is monotonic with respect to R, then the least solution D of the equation*

$$D \;=\; min\, R \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D) \cdot \Lambda\, P^{\circ}$$

satisfies $D \subseteq M$.

The Dynamic Programming Theorem can be interpreted as follows. The $\Lambda\, P^{\circ}$ takes the input and splits it into all possible decompositions. For each of these decompositions the problem is then solved recursively with $\mathsf{F}D$, and combined into a solution for the whole problem with α. An optimal solution is then selected with $min\, R$. Note that this theorem does not require R to be well-founded, but to ensure that we can pick a minimum element with $min\, R$, this is a important property for R to possess.

Now we shall see how this theorem can be applied in the context of partition problems, before looking at specific examples.

3.1 Dynamic Programming for Partitions

We recall that the general specification we had for partition problems was of the form

$$M \;=\; min\, R \cdot (all\ p) \triangleleft \cdot parts,$$

and we wish to express this in a form suitable for applying the dynamic programming theorem, i.e. in the form

$$M \;=\; min\, R \cdot \Lambda\, ([P])^{\circ}.$$

The type of partitions was defined according to the initial algebra of the functor F, where

$$\begin{array}{rcl} \mathsf{F}A & = & \top + (A \times E^+) \\ \mathsf{F}f & = & id + (f \times id), \end{array}$$

and E is the type of elements in the list to be partitioned. As flattening a partition gives back the original list, where

$$\mathit{flatten} \;=\; (\![empty, cat]\!),$$

the way we translate *parts* is

$$parts \;=\; \Lambda\, (\![empty, cat]\!)^\circ.$$

Thus we have

$$\begin{array}{cl} & (all\ p) \triangleleft \cdot parts \\ = & \quad \{\text{by above}\} \\ & (all\ p) \triangleleft \cdot \Lambda\, (\![empty, cat]\!)^\circ \\ = & \quad \{\text{definition of filter}\} \\ & \mathsf{E}((all\ p)?) \cdot \Lambda\, (\![empty, cat]\!)^\circ \\ = & \quad \{\mathsf{E}/\Lambda \text{ law}\} \\ & \Lambda\, ((all\ p)? \cdot (\![empty, cat]\!)^\circ) \\ = & \quad \{\text{converse contravariant; symmetry of } ?\} \\ & \Lambda\, ((\![empty, cat]\!) \cdot (all\ p)?)^\circ\ . \end{array}$$

If $(all\ p)?$ can be expressed as a catamorphism, then we shall be able to express $(\![empty, cat]\!) \cdot (all\ p)?$ as a catamorphism by using the Promotion Theorem (Theorem 1). We can express $(all\ p)?$ as $map\ p?$, where

$$map\ U \;=\; (\![empty, snoc \cdot (id \times U)]\!),$$

and so

$$(all\ p)? \;=\; (\![empty, snoc \cdot (id \times p?)]\!).$$

In order to achieve the condition for the Promotion Theorem, we manipulate $(\![empty, cat]\!) \cdot [empty, snoc \cdot (id \times p?)]$ thus:

$$
\begin{array}{cl}
 & (\![empty, cat]\!) \cdot [\, empty\, ,\ snoc \cdot (id \times p?)] \\
= & \quad \{\text{universal property for catamorphisms; coproduct}\} \\
 & [empty, cat] \cdot \mathsf{F}(\![empty, cat]\!) \cdot \alpha^{\circ} \cdot \alpha \cdot (id + (id \times p?)) \\
= & \quad \{\text{definition of } \mathsf{F};\ \alpha \text{ an isomorphism}\} \\
 & [empty, cat] \cdot (id + ((\![empty, cat]\!) \times id)) \cdot (id + (id \times p?)) \\
= & \quad \{\text{coproduct}\} \\
 & [empty, cat] \cdot (id + (id \times p?)) \cdot (id + ((\![empty, cat]\!) \times id)) \\
= & \quad \{\text{coproduct; definition of } \mathsf{F}\} \\
 & [empty, cat \cdot (id \times p?)] \cdot \mathsf{F}(\![empty, cat]\!),
\end{array}
$$

and so by applying the Promotion Theorem, we obtain

$$(\![empty, cat]\!) \cdot (all\ p)? = (\![\, empty, cat \cdot (id \times p?)]\!).$$

Hence we can write the general specification for partition problems as

$$M = min\ R \cdot \Lambda(\![empty, cat \cdot (id \times p?)]\!)^{\circ}.$$

Aside from choosing suitable preorder for R, there remains the monotonicity condition on α to be checked:

$$
\begin{array}{cl}
 & \alpha \cdot \mathsf{F}R \subseteq R \cdot \alpha \\
\equiv & \quad \{\text{definitions of } \mathsf{F},\ \alpha\} \\
 & [empty, snoc] \cdot (id + (R \times id)) \subseteq R \cdot [empty, snoc] \\
\equiv & \quad \{\text{coproduct}\} \\
 & [empty, snoc \cdot (R \times id)] \subseteq [R \cdot empty, R \cdot snoc] \\
\equiv & \quad \{\text{coproduct; } R \text{ a preorder} \Rightarrow empty \subseteq R \cdot empty\} \\
 & snoc \cdot (R \times id) \subseteq R \cdot snoc.
\end{array}
$$

As R is a preorder and thus transitive, this last equation is equivalent to saying that R is *suffix-stable*, i.e. that

$$y\, R\, x \quad \Rightarrow \quad (y +\!\!+\, z)\, R\, (x +\!\!+\, z).$$

Hence we can apply the Dynamic Programming Theorem to partition problems (as we have specified them) if R is a pre-order and suffix-stable.

It is interesting to compare this Dynamic Programming Theorem of De Moor's with the original theorem in the List Partitions paper ([BM92c]). The original theorem reads

Theorem 3 *If R is a suffix–stable linear order, then*

$$\mathsf{mp} \quad = \quad (\mathsf{null} \rightarrow \mathsf{id},\ \sqcap_R/ \cdot \mathsf{pmp}{*} \cdot q \triangleleft \cdot \mathsf{splits}),$$

where $\mathsf{pmp}(y, z) = \mathsf{mp}\, y +\!\!+ [z]$ *and* $q\,(y, z) = (z \neq [\,]) \wedge p\, z$.

Although the two theorems produce essentially the same algorithm for partition problems, De Moor's theorem has the advantage that it is more general, as it applies to any datatype, rather than just the datatype for partitions.

3.2 Examples

3.2.1 The Equipment Replacement Problem

A common concern in a factory is how to organize the purchase and maintenance of machinery. At the beginning of each year, the factory manager has to decide whether to replace a certain machine, based on the following costs:

- A new machine costs £N.
- Maintenance of a machine in the ith year of its operation costs m_i.
- The salvage value of a machine after i years of operation is s_i.

A purchasing schedule for the next n years can be represented as a partition of the numbers $1 \ldots n$. A partition component $[i \ldots j]$ represents the purchase of a new machine at the beginning of year i, and use of that machine ending at the end of year j. Thus if we define

$$\begin{aligned} \mathit{cost}\ \mathit{xs} \quad &= \quad \mathit{sum}\,(\mathit{map}\ w\ \mathit{xs}) \\ w\ x \quad &= \quad N - s_{\mathit{length}(x)} + \sum_{i=1}^{\mathit{length}(x)} m_i, \end{aligned}$$

we can specify the equipment replacement problem as

$$M \quad = \quad min\,(\leq_{cost}) \cdot parts,$$

where for any function f,

$$xs \leq_f ys \quad \equiv \quad f(xs) \leq f(ys).$$

The equipment replacement problem is an example of a *least weight partition problem*. Such problems can be specified

$$M \quad = \quad min\,(\leq_W) \cdot (all\ p) \triangleleft \cdot parts.$$

In this specification, W is some weighting function, which sometimes takes the form $W = sum \cdot (map\ weight)$, where *weight* is a function returning the weight of an individual partition component. In this case, as in the equipment replacement problem, $\leq_W$ is suffix-stable, and thus the above theory applies to yield a dynamic programming solution to the problem.

We do not give an algorithm for the solution here, as some more work is required to translate the result of the Dynamic Programming Theorem into an executable program. An example of this showing the calculations and the finished program is given in section 6.2.

3.2.2 Text Formatting

The problem of deciding where to put line breaks in a list of words is another example of a least weight partition problem. We use a partition to represent how a list of words is arranged into a paragraph, where each component of a partition represents a line of text.

There are several considerations to be made when forming paragraphs. We wish each line of text to fit on the page, and so define the predicate

$$\mathsf{fits}\,x \quad = \quad width\,x \leq m,$$

where m is the maximum line width. If the words are represented by numbers, then we can define $width = sum$.

We now choose a weighting function W, so that we can specify the text formatting problem as

$$formatting \quad = \quad min\,(\leq_W) \cdot (all\ \mathsf{fits}) \triangleleft \cdot parts.$$

On choosing W, if we define ws to calculate the amount of "white space" in a line of text

$$ws\, x \quad = \quad m \ - \ width\, x,$$

then two possibilities for W are

$$\begin{aligned} W &= sum \cdot (map\ ws) \cdot init \\ W &= max \cdot (map\ ws) \cdot init, \end{aligned}$$

minimising either the total amount of white space or the maximum amount of white space in all lines of the paragraph except the last.

Unfortunately, $\leq_W$ is not suffix-stable for either of these two choices for W. All is not lost though, because $\leq_W$ is prefix-stable. Earlier we chose to represent the datatype of partitions using lists of the "snoc" variety. Instead we choose to use lists of the "cons" variety instead, so that the datatype of partitions is the initial algebra $[empty, cons]$ of the functor

$$\begin{aligned} \mathsf{F}A &= \top + (E^+ \times A) \\ \mathsf{F}f &= id + (id \times f). \end{aligned}$$

When the theory in the previous section is re-worked using this alternative datatype, we find that the condition for the Dynamic Programming Theorem to hold is that R is a prefix-stable preorder. Hence we obtain a dynamic programming algorithm for these two text formatting problems.

When implementing the algorithms, the dynamic programming technique may be made more efficient by using a tabulation scheme. We omit details.

It is also possible to apply the theory in section 4 to the problem of finding the minimum total amount of white space, and this results in a greedy linear-time algorithm for the problem.

4 Greedy Algorithms

One obvious way to improve on the efficiency of the dynamic programming method is to reduce the number of decompositions of the input that are considered. In this greedy theorem, which is due to Bird and De Moor [BM92b], one decomposition of the input is considered, and this decomposition is selected using $min\, S$ for a suitable relation S.

Theorem 4 *Let $M = min\, R \cdot \Lambda ([\![P]\!])^\circ$, where R is a preorder. Suppose α is monotonic with respect to R, and $S \cdot \mathsf{F}([\![P]\!]) \cdot \alpha^\circ \subseteq \mathsf{F}([\![P]\!]) \cdot \alpha^\circ \cdot R$. Then the (unique) solution G of the equation*

$$G \;=\; \alpha \cdot \mathsf{F}G \cdot min\, S \cdot \Lambda\, P^\circ$$

satisfies $G \subseteq M$.

We can interpret the Greedy Theorem similarly to how we interpreted the Dynamic Programming Theorem. Again, $\Lambda\, P^\circ$ takes the input and splits this into all possible decompositions, from which a minimum one with respect to S is selected. Then for this decomposition the problem is solved recursively with $\mathsf{F}G$ and combined into a solution for the whole problem with α.

Now we shall apply the greedy theorem to partition problems in general, before looking at specific examples.

4.1 Greedy Algorithms for Partitions

As before, we have that

$$\begin{aligned} M &= min\, R \cdot \Lambda ([\![P]\!])^\circ, \\ \text{where} \quad P &= [empty, cat \cdot (id \times p?)], \end{aligned}$$

and provided R is a suffix-stable preorder, we have that α is monotonic on R. For partition problems, all that remains is to choose a suitable S to use, which depends on the relation R. In the rest of this section, we will choose to use as an example the length-comparing relation Len for R, which is an ordering frequently used in partition problems. Len is a preorder and suffix-stable, and is defined thus:

$$\begin{aligned} Len &= length^\circ \cdot Leq \cdot length \\ length &= ([\![zero, succ \cdot outl]\!]), \end{aligned}$$

where *zero* is the constant function returning the number 0, and *succ* is the successor function adding 1 to a number.

Further on in this section, we will be assuming that p is prefix-closed. A predicate p is *prefix-closed* when

$$Init \cdot p? \ \subseteq \ p? \cdot Init,$$

which translated, says that

$$p\,(x \mathbin{+\!\!\!+\!\!<} a) \ \Rightarrow \ p\,x.$$

Now we must choose a suitable S, so that $min\,S$ may select the best decomposition of the input. $\Lambda\,P^{\circ}$ takes the input list, and for a non-empty list, returns a set of candidate solutions all of the form (x, y), where $x \in E^*$, $y \in E^+$, $x \mathbin{+\!\!\!+} y$ is the original input, and $p\,y$ is true. As R is Len, the best overall partition is the shortest one, so a reasonable guess as to the best of these candidate solutions generated by $\Lambda\,P^{\circ}$ is the one that has "soaked up" the most input, ie (x_1, y_1) is better than (x_2, y_2) when y_1 is longer than y_2, or alternatively, this can be expressed as x_1 being a prefix of x_2. This motivates our choice of S to be $V\,Pre$, where

$$V\,X \ = \ id \ + \ outl^{\circ} \cdot X \cdot outl$$

We will need the following fact about V, the so-called *swapping rule*, which says that when $X \cdot Y \subseteq Y \cdot Z$, then $V\,X \cdot \mathsf{F}Y \subseteq \mathsf{F}Y \cdot V\,Z$. Furthermore, this rule also holds with $\subseteq$ replaced by either $=$ or $\supseteq$. The proof is as follows:

$$\begin{array}{ll}
 & V\,X \cdot \mathsf{F}Y \\
= & \quad \{\text{definitions of } V, \mathsf{F}\} \\
 & (id \ + \ outl^{\circ} \cdot X \cdot outl) \cdot (id \ + \ (Y \times id)) \\
= & \quad \{\text{property of coproducts}\} \\
 & id \ + \ outl^{\circ} \cdot X \cdot outl \cdot (Y \times id) \\
= & \quad \{\text{naturality of } outl\} \\
 & id \ + \ outl^{\circ} \cdot X \cdot Y \cdot outl \\
\subseteq & \quad \{\text{assumption}\} \\
 & id \ + \ outl^{\circ} \cdot Y \cdot Z \cdot outl \\
= & \quad \{\text{naturality of } outl\}
\end{array}$$

$$
\begin{aligned}
& id + (Y \times id) \cdot outl^\circ \cdot Z \cdot outl \\
= & \quad \{\text{property of coproducts}\} \\
& (id + (Y \times id)) \cdot (id + outl^\circ \cdot Z \cdot outl) \\
= & \quad \{\text{definitions of } V, \mathsf{F}\} \\
& \mathsf{F}Y \cdot V\, Z.
\end{aligned}
$$

Having chosen our S, we need to verify the condition in the Greedy Theorem, i.e. that $S \cdot \mathsf{F}(\![P]\!) \cdot \alpha^\circ \subseteq \mathsf{F}(\![P]\!) \cdot \alpha^\circ \cdot Len$. We reason

$$
\begin{aligned}
& S \cdot \mathsf{F}(\![P]\!) \cdot \alpha^\circ \\
= & \quad \{\text{definition of } S\} \\
& V\, Pre \cdot \mathsf{F}(\![P]\!) \cdot \alpha^\circ \\
\subseteq & \quad \{\text{swapping rule and claim: } Pre \cdot (\![P]\!) \subseteq (\![P]\!) \cdot Len\} \\
& \mathsf{F}(\![P]\!) \cdot V\, Len \cdot \alpha^\circ \\
\subseteq & \quad \{\text{claim: } V\, Len \cdot \alpha^\circ \subseteq \alpha^\circ \cdot Len\} \\
& \mathsf{F}(\![P]\!) \cdot \alpha^\circ \cdot Len.
\end{aligned}
$$

The second of the claims made above is equivalent (by instantiating the definition of Len and using properties of functions) to the inequality

$$
length \cdot \alpha \cdot V\, Len \subseteq Leq \cdot length \cdot \alpha,
$$

and is verified thus:

$$
\begin{aligned}
& length \cdot \alpha \cdot V\, Len \\
= & \quad \{\text{universal property for catamorphisms}\} \\
& [zero, succ \cdot outl] \cdot \mathsf{F}length \cdot V\, Len \\
\subseteq & \quad \{\text{swapping rule using } length \cdot Len \subseteq Leq \cdot length\} \\
& [zero, succ \cdot outl] \cdot V\, Leq \cdot \mathsf{F}length \\
= & \quad \{\text{definition of } V\text{; property of coproducts}\} \\
& [zero, succ \cdot outl \cdot outl^\circ \cdot Leq \cdot outl] \cdot \mathsf{F}length \\
= & \quad \{outl \cdot outl^\circ = id\} \\
& [zero, succ \cdot Leq \cdot outl] \cdot \mathsf{F}length \\
\subseteq & \quad \{succ \cdot Len \subseteq Len \cdot succ\}
\end{aligned}
$$

$$
\begin{array}{ll}
& [zero, Leq \cdot succ \cdot outl] \cdot \mathsf{F}length \\
\subseteq & \quad \{id \subseteq Leq; \text{ property of coproduct}\} \\
& Leq \cdot [zero, succ \cdot outl] \cdot \mathsf{F}length \\
= & \quad \{\text{universal property of catamorphisms}\} \\
& Leq \cdot length \cdot \alpha.
\end{array}
$$

The first claim is more difficult to prove. By Lemma 1, we only need show $Init \cdot (\![P]\!) \subseteq (\![P]\!) \cdot Len$, since $Pre = Init^*$. We start proving this as follows:

$$
\begin{array}{ll}
& Init \cdot (\![P]\!) \\
= & \quad \{\text{universal property of catamorphisms}\} \\
& Init \cdot P \cdot \mathsf{F}(\![P]\!) \cdot \alpha^\circ \\
= & \quad \{\text{definition of P; property of coproducts}\} \\
& [Init \cdot empty, Init \cdot cat \cdot (id \times p?)] \cdot \mathsf{F}(\![P]\!) \cdot \alpha^\circ \\
= & \quad \{Init \cdot empty = 0; \text{ definition of } \mathsf{F}\} \\
& [0, Init \cdot cat \cdot ((\![P]\!) \times p?)] \cdot \alpha^\circ \\
= & \quad \{\text{definition of } \alpha; \;\; [S, T] \cdot [U, V]^\circ = S \cdot U^\circ \cup T \cdot V^\circ\} \\
& 0 \cdot empty^\circ \cup \; Init \cdot cat \cdot ((\![P]\!) \times p?) \cdot snoc^\circ \\
\subseteq & \quad \{0 \text{ identity of } \cup; \text{ property of } Init \cdot cat\} \\
& (cat \cdot (id \times Init) \cup outl \cdot (id \times single^\circ)) \cdot ((\![P]\!) \times p?) \cdot snoc^\circ \\
= & \quad \{\text{distribution of composition over union; product}\} \\
& cat \cdot ((\![P]\!) \times Init \cdot p?) \cdot snoc^\circ \cup outl \cdot ((\![P]\!) \times single^\circ \cdot p?) \cdot snoc^\circ \\
\subseteq & \quad \{\text{assumption: } p \text{ is prefix-closed; naturality of outl}\} \\
& cat \cdot ((\![P]\!) \times p? \cdot Init) \cdot snoc^\circ \cup \; (\![P]\!) \cdot outl \cdot snoc^\circ \\
= & \quad \{\text{definition of } Init\} \\
& cat \cdot ((\![P]\!) \times p? \cdot Init) \cdot snoc^\circ \cup \; (\![P]\!) \cdot Init \\
= & \quad \{\text{claims: } cat \cdot ((\![P]\!) \times p? \cdot Init) \cdot snoc^\circ \subseteq \; (\![P]\!) \cdot Len \\
& \text{and that } Init \subseteq Len\} \\
& (\![P]\!) \cdot Len.
\end{array}
$$

The second of these claims is equivalent to $length \cdot outl \subseteq Leq \cdot length \cdot snoc$, (by instantiating the definitions of Len and $Init$), and is verified as follows:

$$
Leq \cdot length \cdot snoc
$$

$$
\begin{array}{cl}
= & \{\text{universal property of catamorphisms}\} \\
& Leq \cdot [zero, succ \cdot outl] \cdot \mathsf{F}length \cdot \alpha^\circ \cdot snoc \\
= & \{\text{definition of } \mathsf{F}\text{; property of coproduct}\} \\
& Leq \cdot [zero, succ \cdot outl \cdot (length \times id)] \cdot \alpha^\circ \cdot snoc \\
= & \{\text{definition of } \alpha;\ \ [S,T] \cdot [U,V]^\circ = S \cdot U^\circ \cup T \cdot V^\circ\} \\
& Leq \cdot (zero \cdot empty^\circ \cup succ \cdot length \cdot outl \cdot snoc^\circ) \cdot snoc \\
= & \{\text{composition distributes over } \cup;\ empty^\circ \cdot snoc = 0\} \\
& Leq \cdot succ \cdot length \cdot outl \\
\supseteq & \{Leq \cdot succ \supseteq id\} \\
& length \cdot outl.
\end{array}
$$

It remains to show that $cat \cdot ((\![P]\!) \times p? \cdot Init) \cdot snoc^\circ \subseteq (\![P]\!) \cdot Len$. We proceed thus

$$
\begin{array}{cl}
& cat \cdot ((\![P]\!) \times p? \cdot Init) \cdot snoc^\circ \\
= & \{\text{product}\} \\
& cat \cdot (id \times p?) \cdot ((\![P]\!) \times id) \cdot (id \times Init) \cdot snoc^\circ \\
= & \{\text{definition of } P\text{; characterization of catamorphisms}\} \\
& (\![P]\!) \cdot snoc \cdot (id \times Init) \cdot snoc^\circ .
\end{array}
$$

It now suffices to prove $(id \times Init) \subseteq snoc^\circ \cdot Len \cdot snoc$, since $snoc$ is a function.

$$
\begin{array}{cl}
& snoc^\circ \cdot Len \cdot snoc \\
= & \{\text{definition of } length\text{; characterization of catamorphisms}\} \\
& (succ \cdot outl \cdot (length \times id))^\circ \cdot Leq \cdot succ \cdot outl \cdot (length \times id) \\
\supseteq & \{id \subseteq Leq\text{; converse}\} \\
& (length^\circ \times id) \cdot outl^\circ \cdot succ^\circ \cdot succ \cdot outl \cdot (length \times id) \\
= & \{\text{naturality of } outl\} \\
& outl^\circ \cdot length^\circ \cdot succ^\circ \cdot succ \cdot length \cdot outl \\
\supseteq & \{succ^\circ \cdot succ = id;\ length^\circ \cdot length \subseteq id\} \\
& outl^\circ \cdot outl \\
= & \{\text{property of outl}\}
\end{array}
$$

$$
\begin{array}{ll}
 & id \times \Pi \\
\supseteq & \quad \{Init \subseteq \Pi\} \\
 & id \times Init.
\end{array}
$$

Thus we have that for Len as our choice of R, and with p prefix-closed, we can apply the Greedy Theorem to our partition problems, using $V\ Pre$ as our choice of S.

4.2 Examples

4.2.1 Coding With Respect to a Dictionary

As introduced earlier, this problem is to prepare a piece of text for compression by dividing it into substrings, each of which is a prefix of some word in the dictionary. The compression is then achieved by replacing each substring with a pointer into the dictionary. To ensure the feasibility of the problem, a dictionary has the property that every possible character begins some word in the dictionary. Thus if ws is the dictionary of words, the predicate dict is defined

$$\mathsf{dict}\ x \quad = \quad (\exists w \in ws \ : x\ Pre\ w),$$

and thus the problem of preparing a piece of text for compression can be specified

$$prepare \quad = \quad min\ Len \cdot (all\ \mathsf{dict}) \triangleleft \cdot parts.$$

Clearly dict is prefix-closed, and so the above theory can be applied to this problem, yielding a greedy algorithm. In order to implement the recursion equation that is the result of the Greedy Theorem, we note that the relation $min\, S$ (where $S = id + outl^{\circ} \cdot Pre \cdot outl$) yields a unique minimum on the range of $\Lambda[empty, cat \cdot (id \times \mathsf{dict}?)]^{\circ}$. This results in the following algorithm, which can be translated into a suitable programming language:

$$
\begin{array}{lll}
prepare\ text & = [\,], & \text{if } text = [] \\
 & = (prepare\ initial) + \!\!\! + final, & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\text{where } initial + \!\!\! + final & = & text \\
final & = & (\sqcup_{\#}/ \cdot \mathsf{dict} \triangleleft \cdot tails)\ text.
\end{array}
$$

The complexity of this algorithm depends on the complexity of the calculation $(\sqcup_{\#}/ \cdot \mathsf{dict} \triangleleft \cdot \mathit{tails})$. This can be done in amortized linear time, from a generalization of the KMP algorithm as discussed in [BGJ89]. (We omit details of this.)

Thus an amortized linear-time algorithm for the complete problem is obtained.

4.2.2 Motorway Driving

Another example where a greedy algorithm is applicable is in the case of motorway driving. A motorist has a car with fuel capacity sufficient to drive m miles, and the route along the motorway has several service stations at which petrol may be obtained. Obviously the motorist does not wish to run out of petrol, however stopping at a service station takes time, and so it is desired to make as few stops as possible. The problem is to find the best plan of which stations to stop at.

Suppose the motorist starts off at the point s_0 with a full tank. There are service stations $1, 2 \ldots, n-1$ at points $s_1, s_2, \ldots, s_{n-1}$ along the route, and the end of the journey is at point s_n. The distance from s_i to s_{i+1} is d_i, and we assume that for all i, $d_i \leq m$. A plan of which stations the motorist stops at can be represented by a partition of $[d_0, d_1, \ldots d_{n-1}]$. A partition component $[d_i, \ldots d_{j-1}]$ represents the motorist stopping at the point s_i to fill up the petrol tank of the car, but then not stopping again before the point s_j.

We wish partition components to satisfy the condition that the motorist does not run out of petrol during the distance represented by that component. So we define the following predicate, which represents that the amount of petrol put in the tank is sufficient to drive that distance:

$$\mathsf{sufficient}\, x \quad = \quad \mathit{sum}\, x \;\leq\; m.$$

Now the motorway driving problem can be specified as

$$\mathit{plan} \quad = \quad \mathit{min}\, \mathit{Len} \cdot (\mathit{all}\; \mathsf{sufficient}) \triangleleft \cdot \mathit{parts}.$$

The predicate sufficient is clearly prefix-closed, and so the theory above can be applied. Similarly to the previous example, the greedy algorithm obtained from applying the Greedy Theorem relies on the calculation of

$(\sqcup_{\#}/ \cdot \textsf{sufficient} \triangleleft \cdot \mathit{tails})$. This can be computed in linear time with a right reduction by using an extra variable to remember the sum of the distances considered so far in the current partition component. Using this, the algorithm to the whole problem can be expressed using a right reduction, and it runs in linear time:

$$\begin{array}{lcl} \mathit{plan}\ \ [\,] & = & [\,] \\ \mathit{plan}\ \ (ds +\!\!+ [d]) & = & \mathit{first}\ \ (\oplus \not\leftarrow ([[d]], d)\ ds) \end{array}$$

$$\begin{array}{ll} \text{where} & d \oplus ([x] +\!\!+ xs, sum) \\ & \quad = \ ([[d] +\!\!+ x] +\!\!+ xs, sum + d), \quad \text{if}\,(sum + d) \leq m \\ & \quad = \ ([[d]] +\!\!+ [x] +\!\!+ xs, d), \qquad\quad \text{otherwise.} \end{array}$$

5 Further Optimization

Using the greedy theorem, we have seen how efficient algorithms for partition problems may be obtained. A simpler (and possibly more efficient) algorithm may be obtained under certain conditions by expressing it as a catamorphism. We have that

$$M \ = \ min\, R \cdot \ \Lambda\, ([P])^{\circ}.$$

where $P \ = \ [empty, cat \cdot (id \ \times \ p?)]$, and it happens to be the case that we will find it easier to express M as a catamorphism if we first express $([P])^{\circ}$ as a catamorphism, i.e. we want to find a Q such that

$$([P])^{\circ}{}_{\mathsf{F}_{Partitions}} \quad = \quad ([Q])_{\mathsf{F}_{E*}} \, .$$

Intuitively, this means that instead of doing $([P])^{\circ}$, which is the opposite of flattening a partition into a list, $([Q])$ will construct a partition directly from the original list. From the universal property of catamorphisms, the above equation is equivalent to

$$([P])^{\circ} \ \cdot \ \alpha \quad = \quad Q \ \cdot \ \mathsf{F}([P])^{\circ}.$$

Manipulating the left hand side of the above equation until the form of the right hand side is obtained, we have

$$\begin{array}{cl}
 & ([P])^{\circ} \ \cdot \ \alpha \\
= & \quad \{\text{universal property of catamorphisms; converse}\} \\
 & \alpha \ \cdot \ \mathsf{F}([P])^{\circ} \ \cdot \ P^{\circ} \ \cdot \ \alpha \\
= & \quad \{\text{claim: } P^{\circ} \cdot \alpha \ = \ (id \ + \ (id \times p?) \cdot cat^{\circ} \cdot snoc)\} \\
 & \alpha \ \cdot \ \mathsf{F}([P])^{\circ} \ \cdot \ (id \ + \ (id \times p?) \cdot cat^{\circ} \cdot snoc) \\
= & \quad \{\text{definition of } \mathsf{F}\text{; coproduct}\} \\
 & \alpha \ \cdot \ (id \ + \ (([P])^{\circ} \ \times \ p?) \ \cdot \ cat^{\circ} \ \cdot snoc).
\end{array}$$

Since we have that

$$\begin{array}{cl}
 & cat^{\circ} \ \cdot snoc \\
= & \quad \{\text{definition of } cat\text{; converse}\} \\
 & ((id \times single) \ \cdot \ snoc^{\circ} \cup \ (id \times snoc) \ \cdot \ assoc^{\circ} \ \cdot (cat^{\circ} \times id) \ \cdot \ snoc^{\circ}) \ \cdot \ snoc \\
= & \quad \{snoc^{\circ} \ \cdot snoc \ = \ id_{E^* \times E}\} \\
 & (id \times single) \ \cup \ (id \times snoc) \ \cdot \ assoc^{\circ} \ \cdot (cat^{\circ} \times id),
\end{array}$$

referring back to a sub-expression of the previous result, we obtain

$$
\begin{array}{ll}
 & ((\![P]\!)^\circ \times p?) \cdot cat^\circ \cdot snoc \\
= & \quad \{\text{from above}\} \\
 & ((\![P]\!)^\circ \times p?) \cdot ((id \times single) \cup (id \times snoc) \cdot assoc^\circ \cdot (cat^\circ \times id)) \\
= & \quad \{\text{distribution of } \cdot \text{ over } \cup\} \\
 & ((\![P]\!)^\circ \times p?) \cdot (id \times single) \qquad\qquad\qquad\qquad \text{(i)} \\
 & \cup\ ((\![P]\!)^\circ \times p?) \cdot (id \times snoc) \cdot assoc^\circ \cdot (cat^\circ \times id) \ \text{(ii)}.
\end{array}
$$

Looking at these sub-expressions separately, the first simplifies to

$$
\begin{array}{ll}
 & \text{(i)} \\
= & \quad \{\times;\ p \text{ prefix-closed implies that } p? \cdot single = single\} \\
 & (id \times single) \cdot ((\![P]\!)^\circ \times id),
\end{array}
$$

and the second can be manipulated

$$
\begin{array}{ll}
 & \text{(ii)} \\
= & \quad \{\times;\ p \text{ prefix-closed with suffix-derivative } q \text{ implies that} \\
 & \qquad\qquad p? \cdot snoc = snoc \cdot q? \cdot (p? \times id)\} \\
 & (id \times snoc \cdot q?) \cdot ((\![P]\!)^\circ \times (p? \times id)) \cdot assoc^\circ \cdot (cat^\circ \times id) \\
= & \quad \{\text{property of } assoc;\ \times\} \\
 & (id \times snoc \cdot q?) \cdot assoc^\circ \cdot (((\![P]\!)^\circ \times p?) \cdot cat^\circ \times id) \\
= & \quad \{\text{claim: } ((\![P]\!)^\circ \times p?) \cdot cat^\circ = snoc^\circ \cdot (\![P]\!)^\circ;\ \times\} \\
 & (id \times snoc \cdot q?) \cdot assoc^\circ \cdot (snoc^\circ \times id) \cdot ((\![P]\!)^\circ \times id).
\end{array}
$$

Hence from the above we have

$$
\begin{array}{ll}
 & (\![P]\!)^\circ \cdot \alpha \\
= & \quad \{\text{from above}\} \\
 & \alpha \cdot (id + (id \times single) \cdot ((\![P]\!)^\circ \times id) \\
 & \qquad\quad \cup\ (id \times snoc \cdot q?) \cdot assoc^\circ \cdot (snoc^\circ \times id) \cdot ((\![P]\!)^\circ \times id)) \\
= & \quad \{\text{definition of } \alpha;\ \text{distribution of } \cdot \text{ over } \cup\} \\
 & [empty, \\
 & snoc \cdot ((id \times single) \cup (id \times snoc \cdot q?) \cdot assoc^\circ \cdot (snoc^\circ \times id)) \cdot ((\![P]\!)^\circ \times id)]
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{coproduct; definition of } \mathsf{F}\text{; defining} \\
& \quad new \;=\; snoc \cdot (id \times single) \\
& \quad glue \;=\; snoc \cdot (id \times snoc \cdot q?) \cdot assoc^{\circ} \cdot (snoc^{\circ} \times id)\} \\
& [empty\,,\; new \cup glue] \cdot \mathsf{F}(\!|P|\!)^{\circ}.
\end{array}
$$

We now have two claims to verify.
The first is that $P^{\circ} \cdot \alpha \;=\; id + (id \times p?) \cdot cat^{\circ} \cdot snoc$:

$$
\begin{array}{ll}
& P^{\circ} \cdot \alpha \\
= & \quad \{\text{definitions of } P \text{ and } \alpha\} \\
& [empty, cat \cdot (id \times p?)]^{\circ} \cdot [empty, snoc] \\
= & \quad \{\text{definition of join; converse}\} \\
& (inl \cdot empty^{\circ} \cup inr \cdot (id \times p?) \cdot cat^{\circ}) \\
& \cdot (empty \cdot inl^{\circ} \cup snoc \cdot inr^{\circ}) \\
= & \quad \{empty \cdot snoc = 0 \text{ and } cat^{\circ} \cdot empty = 0\} \\
& inl \cdot empty^{\circ} \cdot empty \cdot inl^{\circ} \\
& \cup\; inr \cdot (id \times p?) \cdot cat^{\circ} \cdot snoc \cdot inr^{\circ} \\
= & \quad \{\text{definition of coproduct}\} \\
& id + (id \times p?) \cdot cat^{\circ} \cdot snoc,
\end{array}
$$

and the second claim is that $snoc^{\circ} \cdot (\!|P|\!)^{\circ} \;=\; ((\!|P|\!)^{\circ} \times p?) \cdot cat^{\circ}$, which is verified thus:

$$
\begin{array}{ll}
& snoc^{\circ} \cdot (\!|P|\!)^{\circ} \\
= & \quad \{\text{converse}\} \\
& ((\!|P|\!) \cdot snoc)^{\circ} \\
= & \quad \{\text{characterization of catamorphisms}\} \\
& (cat \cdot (id \times p?) \cdot ((\!|P|\!) \times id))^{\circ} \\
= & \quad \{\text{converse; } \times\} \\
& ((\!|P|\!)^{\circ} \times p?) \cdot cat^{\circ}.
\end{array}
$$

Now we can express our partition problem as

$$
M \;=\; min\, R \cdot \Lambda(\!|Q|\!),
$$

where $Q = [empty, new \cup glue]$. In fact, *new* is a total function and *glue* a partial function, and so they can also be written

$$\begin{array}{rcll} new(xs, a) & = & xs \mathbin{+\!\!+\!\!<} [a] & \\ glue(xs \mathbin{+\!\!+\!\!<} x, a) & = & xs \mathbin{+\!\!+\!\!<} (x \mathbin{+\!\!+\!\!<} a), & \text{if } q(x, a). \end{array}$$

This definition of Q seems reasonable when thought about in an informal way, as when considering the next element of the list to be partitioned, the element either starts a *new* partition, or is *glue*d onto the end of the last component of the partition so far if p holds.

Note that an alternative method of finding Q would have been to first consider the expression $(all\ p?) \triangleleft \cdot \Lambda\, flatten^{\circ}$, metamorphose the $flatten^{\circ}$ into a catamorphism using the same method above, and then promote the filter into the catamorphism using the Promotion Theorem. However, as the above method uses only two steps explicitly involving p, the proof given above is shorter (although possibly less elegant).

5.1 Finding a Functional Catamorphism

To use a catamorphism to solve a partition problem efficiently, we wish to find a function f such that

$$(\![f]\!) \subseteq min\, R \cdot \Lambda (\![Q]\!).$$

Searching for a suitable definition for f we manipulate this as follows

$$\begin{array}{cll}
 & (\![f]\!) \subseteq min\, R \cdot \Lambda(\![Q]\!) & \\
\Leftarrow & \quad \{\text{Proposition 2}\} & \\
 & f \cdot \mathsf{F}(min\, R \cdot \Lambda(\![Q]\!)) \subseteq min\, R \cdot \Lambda(\![Q]\!) \cdot \alpha & \\
\equiv & \quad \{\Lambda\text{-composition; characterization of catamorphisms}\} & \\
 & f \cdot \mathsf{F}(min\, R \cdot \Lambda(\![Q]\!)) \subseteq min\, R \cdot \Lambda\,(Q \cdot \mathsf{F}(\![Q]\!)) & \\
\equiv & \quad \{\text{universal property for } min\} & \\
 & f \cdot \mathsf{F}(min\, R \cdot \Lambda(\![Q]\!)) \subseteq Q \cdot \mathsf{F}(\![Q]\!) & \text{(i)} \\
 & f \cdot \mathsf{F}(min\, R \cdot \Lambda(\![Q]\!)) \cdot (Q \cdot \mathsf{F}(\![Q]\!))^{\circ} \subseteq R & \text{(ii).}
\end{array}$$

With the first inequality we can reason

$$
\begin{array}{ll}
 & \text{(i)} \\
\Leftarrow & \{min\,R \subseteq \in;\ \text{definition of } \Lambda\} \\
 & f \cdot \mathsf{F}(\!|Q|\!) \subseteq Q \cdot \mathsf{F}(\!|Q|\!) \\
\Leftarrow & \{\text{monotonicity}\} \\
 & f \subseteq Q.
\end{array}
$$

Recall that $Q = [empty, new \cup glue]$. new is a total function on its domain, while $glue$ is a simple relation (or a partial function) on the same set. If it is the shortest partition that we require, then when forming our partition using $(\!|f|\!)$, we wish to do as much $glue$ing as possible, rather than starting a new partition at each step. Thus a sensible choice for the second component of f is to be $glue$ wherever possible, and new otherwise. This leads to the definition $f = [empty\,,\ start \cup glue \cup add]$, where

$$
\begin{array}{rcll}
start\,([\,], a) & = & [[a]] & \\
glue(xs \mathbin{+\!\!\!+\!\!<} x, a) & = & xs \mathbin{+\!\!\!+\!\!<} (x \mathbin{+\!\!\!+\!\!<} a), & \textit{if}\ q(x,a) \\
add(xs \mathbin{+\!\!\!+\!\!<} x, a) & = & (xs \mathbin{+\!\!\!+\!\!<} x) \mathbin{+\!\!\!+\!\!<} a, & \textit{if}\ \neg q(x,a).
\end{array}
$$

Now we need to check that our definition of f satisfies the second inequation. Rearranging (ii) slightly, we have the condition

$$
f \cdot \mathsf{F}(min\,R \cdot \Lambda(\!|Q|\!) \cdot (\!|Q|\!)^{\circ}) \cdot Q^{\circ} \subseteq R
$$

Unfortunately choosing Len for R does not make (ii) valid. If we wish to choose the shortest partition, we need instead a preorder that respects Len. The preorder $SemiLex$ is defined in the following way:

$$
xs\ SemiLex\ ys \ \equiv\ xs = ys \ \vee\ xs <_{SL} ys
$$
$$
\text{where}\ \ xs <_{SL} ys \equiv (\exists k : xs \uparrow k = ys \uparrow k \ \wedge\ (ys.k)\,Pre\,(xs.k)).
$$

On partitions of the same sequence, $SemiLex$ is a total order (so there is a unique minimum partion). A minimum partition under $SemiLex$ is not usually a minimum partition under Len, however under certain conditions, $SemiLex$ does respect Len, and these conditions will be discussed later in this section. Now we wish to prove that $SemiLex$ satisfies (ii). Denoting $min\,R \cdot \Lambda(\!|Q|\!) \cdot (\!|Q|\!)^{\circ}$ by T, we have that

$$
\begin{array}{ll}
 & \text{(ii)} \\
\equiv & \quad \{\text{definitions of } \mathsf{F} \text{ and } Q\} \\
 & f \cdot (id + T \times id) \cdot [empty, new \cup glue]^{\circ} \subseteq SemiLex \\
\equiv & \quad \{\text{definition of } f\text{; properties of coproduct}\} \\
 & empty \cdot id \cdot empty^{\circ} \subseteq SemiLex \\
 & \wedge \; start \cup glue \cup add \cdot (T \times id) \cdot new^{\circ} \cup glue^{\circ} \subseteq SemiLex.
\end{array}
$$

The first inequality is true as $SemiLex$ is a preorder. The second inequality breaks down into

$$
\begin{array}{rcl}
start \cdot (T \times id) \cdot (new^{\circ} \cup glue^{\circ}) & \subseteq & SemiLex \\
glue \cdot (T \times id) \cdot new^{\circ} & \subseteq & SemiLex \\
glue \cdot (T \times id) \cdot glue^{\circ} & \subseteq & SemiLex \\
add \cdot (T \times id) \cdot new^{\circ} & \subseteq & SemiLex \\
add \cdot (T \times id) \cdot glue^{\circ} & \subseteq & SemiLex.
\end{array}
$$

Now as $T \times id$ relates $([\,], a)$ only to $([\,], a)$, and as $([\,], a)$ is not in the domain of $glue$, the first of these inequations is true as $SemiLex$ is a preorder.

For the remainder we need to consider $(xs +\!\!+ [x], a) \; T \times id \; (ys +\!\!+ [y], a)$. For any sequence s, and partitions of that sequence $xs +\!\!+ [x]$, $ys +\!\!+ [y]$ in $\Lambda\,(\![Q]\!)(s)$, $(xs +\!\!+ [x], a) \; T \times id \; (ys +\!\!+ [y], a)$ when $xs +\!\!+ [x]$ is the unique minimum partition of $\Lambda\,(\![Q]\!)(s)$ under $SemiLex$. For such $xs +\!\!+ [x]$ and $ys +\!\!+ [y]$, the remaining four inequations are equivalent to

$$
\begin{array}{rcll}
p\,(x +\!\!+ [a]) & \Rightarrow & xs +\!\!+ [x +\!\!+ [a]] \; SemiLex \; ys +\!\!+ [y] +\!\!+ [[a]] & \text{(iii)} \\
p\,(x +\!\!+ [a]) \wedge p\,(y +\!\!+ [a]) & \Rightarrow & xs +\!\!+ [x +\!\!+ [a]] \; SemiLex \; ys +\!\!+ [y +\!\!+ [a]] & \text{(iv)} \\
\neg p\,(x +\!\!+ [a]) & \Rightarrow & xs +\!\!+ [x] +\!\!+ [[a]] \; SemiLex \; ys +\!\!+ [y] +\!\!+ [[a]] & \text{(v)} \\
\neg p\,(x +\!\!+ [a]) \wedge p(y +\!\!+ [a]) & \Rightarrow & xs +\!\!+ [x] +\!\!+ [[a]] \; SemiLex \; ys +\!\!+ [y +\!\!+ [a]] & \text{(vi).}
\end{array}
$$

To verify these, we will need three properties of $SemiLex$:

Property 1: (The suffix property) For any us, vs, ws, zs,

$$us <_{SL} vs \;\Rightarrow\; (us +\!\!+ ws) <_{SL} (vs +\!\!+ zs).$$

This property follows directly from the definition of $<_{SL}$.

Property 2: For any u, v, us, vs,

$$us \mathbin{+\!\!+} [u] \; SemiLex \; vs \quad \Rightarrow \quad us \mathbin{+\!\!+} [u \mathbin{+\!\!+} v] <_{SL} vs.$$

This property also follows directly from the definition of $<_{SL}$.

Property 3: If $us \mathbin{+\!\!+} [u]$ and $vs \mathbin{+\!\!+} [v]$ are distinct partitions of the same sequence, then

$$us \mathbin{+\!\!+} [u] <_{SL} vs \mathbin{+\!\!+} [v] \quad \Rightarrow \quad us \mathbin{+\!\!+} [u] <_{SL} vs.$$

This property may be proved as follows:

Proof

From the definition of $SemiLex$,

$$us \mathbin{+\!\!+} [u] <_{SL} vs \mathbin{+\!\!+} [v] \equiv$$
$$(\exists k : (us \mathbin{+\!\!+} [u]) \uparrow k = (vs \mathbin{+\!\!+} [v]) \uparrow k \;\wedge\; (vs \mathbin{+\!\!+} [v]).k \; Pre \, (us \mathbin{+\!\!+} [u]).k).$$

We use a case analysis on the value of k. If $k < length\ vs$, then we have

$$\begin{array}{ll}
& (us \mathbin{+\!\!+} [u]) \uparrow k = (vs \mathbin{+\!\!+} [v]) \uparrow k \;\wedge\; (vs \mathbin{+\!\!+} [v]).k \; Pre \, (us \mathbin{+\!\!+} [u]).k \\
\Rrightarrow & \quad \{k < length\ vs\} \\
& (us \mathbin{+\!\!+} [u]) \uparrow k = vs \uparrow k \;\wedge\; vs.k \; Pre \, (us \mathbin{+\!\!+} [u]).k \\
\Rrightarrow & \quad \{\text{definition of } <_{SL}\} \\
& (us \mathbin{+\!\!+} [u]) <_{SL} vs,
\end{array}$$

and if $k = length\ vs$ then

$$\begin{array}{ll}
& (us \mathbin{+\!\!+} [u]) \uparrow k = (vs \mathbin{+\!\!+} [v]) \uparrow k \;\wedge\; (vs \mathbin{+\!\!+} [v]).k \; Pre \, (us \mathbin{+\!\!+} [u]).k \\
\Rrightarrow & \quad \{k = length\ vs\} \\
& (us \mathbin{+\!\!+} [u]) \uparrow k = vs \;\wedge\; v \; Pre \, (us \mathbin{+\!\!+} [u]).k \\
\Rrightarrow & \quad \{\text{property of } \uparrow\} \\
& \exists zs : \; vs \mathbin{+\!\!+} zs = us \mathbin{+\!\!+} [u] \;\wedge\; v \; Pre \; zs.0 \\
\Rrightarrow & \quad \{us \mathbin{+\!\!+} [u], vs \mathbin{+\!\!+} [v] \text{ partitions of same sequence}\} \\
& \exists zs : \; flatten\ zs = v \;\wedge\; v \; Pre \; zs.0 \\
\Rrightarrow & \quad \{flatten, Pre\} \\
& False
\end{array}$$

$\Box$.

Now for the verification of (iii)–(vi).
Recall that we can assume that $xs + [x], ys + [y] \in \Lambda([Q])(s)$, for some sequence s, and that $(xs + [x])\, SemiLex\, (ys + [y])$.
For (iii), we have that

$$
\begin{array}{ll}
 & xs + [x]\ SemiLex\ ys + [y] \\
\Rightarrow & \quad \{\text{Property 2}\} \\
 & xs + [x + [a]]\ <_{SL}\ ys + [y] \\
\Rightarrow & \quad \{\text{Property 1; definition of } SemiLex\} \\
 & xs + [x + [a]]\ SemiLex\ ys + [y] + [[a]].
\end{array}
$$

For (iv), we have that if $xs + [x] = ys + [y]$ then as $SemiLex$ is a preorder, $(xs + [x + [a]])\, SemiLex\, (ys + [y + [a]])$. Otherwise,

$$
\begin{array}{ll}
 & xs + [x]\ <_{SL}\ ys + [y] \\
\Rightarrow & \quad \{\text{Property 3}\} \\
 & xs + [x]\ <_{SL}\ ys \\
\Rightarrow & \quad \{\text{Property 2}\} \\
 & xs + [x + [a]]\ <_{SL}\ ys \\
\Rightarrow & \quad \{\text{Property 1; definition of } SemiLex\} \\
 & xs + [x + [a]]\ SemiLex\ ys + [y + [a]].
\end{array}
$$

For (v), again if $xs + [x] = ys + [y]$ then as $SemiLex$ is a preorder, we have $xs + [x] + [[a]]\, SemiLex\, ys + [y] + [[a]]$. Otherwise, if $xs + [x]\ <_{SL}\ ys + [y]$, then we get the same result from Property 1.
For (vi), as we have $\neg p\,(x + [a])\ \wedge\ p(y + [a])$, it cannot be the case that $xs + [x] = ys + [y]$. So we have

$$
\begin{array}{ll}
 & xs + [x]\ <_{SL}\ ys + [y] \\
\Rightarrow & \quad \{\text{Property 3}\} \\
 & xs + [x]\ <_{SL}\ ys \\
\Rightarrow & \quad \{\text{Property 1; definition of } SemiLex\} \\
 & xs + [x] + [[a]]\ SemiLex\ ys + [y + [a]].
\end{array}
$$

Having proved that $SemiLex$ is an suitable ordering, we want conditions under which $SemiLex$ respects Len in the following way:

$$min\, SemiLex \cdot \Lambda([Q]) \quad \subseteq \quad min\, Len \cdot \Lambda([Q]).$$

This says that the minimum partition of the partitions formed by $\Lambda\,([Q])$ under *SemiLex* is also one of the shortest of those partitions. A sufficient condition for this inequation to hold is that p be *segment-closed*, that is for non-empty y,

$$p\,(x \mathbin{+\!\!+} y \mathbin{+\!\!+} z) \quad\Rightarrow\quad p\,(x \mathbin{+\!\!+} y) \;\wedge\; p\,(y \mathbin{+\!\!+} z).$$

This is also the same as saying that p is prefix-closed and suffix-closed. We now prove this:

Proof
The proof is by contradiction. Suppose that

$$min\;SemiLex \cdot\; \Lambda\,([Q]) \quad\not\subseteq\quad min\;Len \cdot\; \Lambda\,([Q]).$$

Then there exist partitions $xs, ys\; \in \Lambda\,([Q])(s)$ for some sequence s, such that xs is the (unique) minimum partition of $\Lambda\,([Q])(s)$ under *SemiLex*, and ys is one of those partitions shorter than xs.
Define

$$m \;=\; (\sqcap\, k \;:\; \mathit{flatten}\,(xs \uparrow k)\;\mathit{Pre}\;\mathit{flatten}\,(ys \uparrow k)).$$

Such an m certainly exists, as $\mathit{flatten}\,(ys \uparrow \mathit{length}(ys))$ is the sequence s, and as xs is a partition with more components than ys, $\mathit{flatten}\,(xs \uparrow \mathit{length}(ys))$ is some prefix of s.
With this m, we define the partition zs to be

$$\begin{aligned} zs \;=\;& xs \uparrow (m-1) \mathbin{+\!\!+} [xs.(m-1) \mathbin{+\!\!+} y] \mathbin{+\!\!+} ys \downarrow m \\ & \text{where}\;\; x \mathbin{+\!\!+} xs.(m-1) \mathbin{+\!\!+} y \;=\; ys.(m-1). \end{aligned}$$

We now claim that $zs\; \in \Lambda\,([Q])(s)$ and $zs\;\mathit{SemiLex}\;xs$.

We can see that zs is a partition of s, by observing that $\mathit{flatten}(xs \uparrow (m-1))$ is a prefix of s, that $\mathit{flatten}(ys \downarrow m)$ is a suffix of s, and that the definition of m ensures that $xs.(m-1) \mathbin{+\!\!+} y$ contains the elements in between.

To see that the components of zs satisfy p, we note that the components of $xs \uparrow (m-1)$ and $ys \downarrow m$ satisfy p because $xs, ys\; \in \Lambda\,([Q])(s)$, and that $p\;(xs.(m-1) \mathbin{+\!\!+} y)$ because $p\;(ys.(m-1))$ and p is segment-closed.

That $zs\ SemiLex\ xs$ can be seen from the definition of $SemiLex$ and from the truth of $zs \uparrow (m-1) \ = \ xs \uparrow (m-1) \ \wedge \ xs.(m-1)\ Pre\ zs.(m-1)$. This is true because from the definition of m, y is non-empty.

So we have that $zs \in \Lambda\ ([Q])(s)$ and $zs\ SemiLex\ xs$. This is impossible, because we defined xs to be the unique minimum partition of $\Lambda([Q])(s)$ under $SemiLex$. So we conclude that

$$min\ SemiLex \cdot \ \Lambda\,([Q]) \ \ \subseteq \ \ min\ Len \cdot \ \Lambda\,([Q]).$$

$\Box$.

Another sufficient condition for the above inequation to hold is that p be *robust*, that is for non-empty y,

$$p\,(x +\!\!\!+\, y) \ \wedge \ p\,(y +\!\!\!+\, z) \ \ \Rightarrow \ \ p\,(x +\!\!\!+\, y +\!\!\!+\, z).$$

A proof of this is as follows:

Proof

First we define what is meant by a *sub-division*.

A sub-division of a partition $xs \ = \ [x_0, x_1, \ldots x_{n-1}]$ of a sequence is another partition $ys_0 +\!\!\!+\, ys_1 +\!\!\!+ \ldots ys_{n-1}$ of the same sequence, such that for all i in the range $0 \ldots n-1$, ys_i is a partition of x_i.

We now show for any sequence s, there is a unique shortest partition of $\Lambda\,([Q])(s)$, and that all the other partitions are sub-divisions of it.

Take a partition $xs = \ [x_0, x_1, \ldots x_{n-1}]$ such that $xs\,(min\ Len \cdot \ \Lambda\,([Q]))\,s$. Any other partition of $\Lambda\ ([Q])(s)$ that is not a subdivision of xs, must have one component of the form $v +\!\!\!+\, x_i +\!\!\!+ \ldots +\!\!\!+\, x_{i+k} +\!\!\!+\, w$, where v is a suffix of x_{i-1}, and w is a prefix of x_{i+k+1}. However, as p is robust, and the components of these partitions satisfy p,

$$\begin{aligned} p\ x_{i-1} \ \ \wedge \ \ p\ (v +\!\!\!+\, x_i +\!\!\!+ \ldots +\!\!\!+\, x_{i+k} +\!\!\!+\, w) \ \ \wedge \ \ p\ x_{i+k+1} \\ \Rightarrow \ \ p\ (x_{i-1} +\!\!\!+ \ldots +\!\!\!+\, x_{i+k+1}), \end{aligned}$$

which contradicts the assumption that the shortest partition of $\Lambda\,([Q])(s)$ is xs. Therefore any other of the partitions is a sub-division of xs, which also implies that xs is the unique shortest of the partitions.

Now we just need to show that the unique minimum partition under Len is also the unique minimum partition under $SemiLex$. It suffices to show that for any if ys is a subdivision of xs, then $xs\, SemiLex\, ys$.

If $ys = xs$, then as $SemiLex$ is a preorder, $xs\, SemiLex\, ys$.
Otherwise, there must be some i for which $ys.i \neq [xs.i]$. Take the minimum of those i. Then we have that $xs \uparrow i = ys \uparrow i$, and that as ys is a subdivision of xs, $ys.i\, Pre\, xs.i$. From the definition of $SemiLex$ we now have that $xs\, SemiLex\, ys$.

As the unique minimum partition under $SemiLex$ is also the (unique) minimum partition under Len, we have that

$$min\, SemiLex \cdot \Lambda([Q]) \quad \subseteq \quad min\, Len \cdot \Lambda([Q]).$$

$\square$.

The conclusion of this section is that

$$([empty, start \cup glue \cup add]) \quad \subseteq \quad min\, Len \cdot \Lambda([Q]),$$

provided that p is segment-closed or robust. As $([empty, start \cup glue \cup add])$ is a function, this catamorphism can easily be implemented in a suitable programming language.

5.2 Examples

5.2.1 Sorting by Natural Merging

As stated earlier, the first stage of sorting using the natural merging method is to construct a shortest partition of the original list so that all the components of the partition are ascending. The predicate expressing that a sequence is ascending is up, so the problem can be specified

$$stage1 \quad = \quad min\, Len \cdot (all\ \mathsf{up}) \triangleleft \cdot parts.$$

As up is segment-closed and robust, and therefore the above theory can be applied. Furthermore, by remembering the previous value in the list, the suffix derivative of up can be computed in constant time, and the following algorithm results:

$$
\begin{array}{lll}
stage1 \ \ [\,] & = & [\,] \\
stage1 \ \ (a:x) & = & \mathit{first}\ \ (\oplus \not\leftarrow ([[a]], a)\ \ x)
\end{array}
$$

$$
\begin{array}{llll}
\text{where} \ \ (xs +\!\!+ [x], m) \oplus a & = & (xs +\!\!+ [x +\!\!+ [a]], a), & \text{if } m \leq a \\
 & = & (xs +\!\!+ [x] +\!\!+ [[a]], a), & \text{otherwise.}
\end{array}
$$

5.2.2 The Bank Cash Flow Problem

The bank cash flow problem, which was introduced earlier, is another example of the application of the above theory. Recall that a bank knows in advance a sequence of withdrawals or deposits, which can be represented as a sequence of negative or positive numbers. The bank must always have sufficient money to cover withdrawals but not more than N, for security reasons. A van can bring or take away the money that the bank needs, and the problem is to minimize the number of visits that the van makes.

We shall represent the sequence of visits by the van as a partition of the list of withdrawals and deposits. Each component of the partition represents one visit by the van, and that component must satisfy a suitable predicate. We say that a sequence of numbers is ok if at no time during those transactions does the bank either exhaust its supply of money or have too much, and so define

$$
\mathsf{ok}\, x \ = \ (\mathit{max}\ y \ - \ \mathit{min}\ y) \leq N, \quad \mathit{where}\ \ y \ = \ \mathit{map\, sum}\, (\mathit{inits}\ x),
$$

and thus the problem can be specified

$$
\mathit{cashflow} \ = \ \mathit{min\, Len} \cdot (\mathit{all}\ \mathsf{ok}) \triangleleft \cdot \mathit{parts}.
$$

Although ok is not robust, it is segment-closed, and thus the above theory applies. By remembering the minimum, maximum and current intial sums of the current partition component, the suffix derivative of ok can be computed in constant time, and so a simple algorithm with linear complexity is achieved for this problem:

$$
\begin{array}{lll}
\mathit{cashflow} \ \ [\,] & = & [\,] \\
\mathit{cashflow} \ \ (a:x) & = & \mathit{first}\ \ (\oplus \not\leftarrow ([[a]], a, a, a)\ \ x),
\end{array}
$$

where $(xs ⧺ [x], mx, mn, sum) \oplus a$

$$\begin{array}{lll} = & (xs ⧺ [x ⧺ [a]], m, n, s), & \text{if } (m - n) \leq N \\ = & (xs ⧺ [x] ⧺ [[a]], a, a, a), & \text{otherwise.} \end{array}$$

$$\begin{array}{llll} \text{where} & s & = & sum + a \\ & m & = & max\,(mx, s) \\ & n & = & min\,(mn, s) \end{array}$$

6 Further Partition Problems

So far, the problems we have considered have all fit into the form

$$M \;=\; min\,R \cdot (all\,p) \triangleleft \cdot parts.$$

However, not all partition problems can be expressed in this way. One generalization that can be made is to look at partition problems of the form

$$M \;=\; min\,R \cdot p \triangleleft \cdot parts.$$

In order to attempt to apply the dynamic programming and greedy theorems, as before, we should absorb the filter into the generation of the partitions. (It is of course possible to absorb the filter into the relation R, but it is more efficient to generate only the partitions that satisfy p.) As before, taking the definition of $parts$ to be $\Lambda\,(\![empty, cat]\!)^\circ$, we have

$$\begin{array}{ll} & p \triangleleft \cdot \Lambda\,(\![empty, cat]\!)^\circ \\ = & \quad \{\text{definition of filter}\} \\ & \mathsf{E}(p?) \cdot \Lambda\,(\![empty, cat]\!)^\circ \\ = & \quad \{\mathsf{E}/\Lambda \text{ law}\} \\ & \Lambda\,(p? \cdot (\![empty, cat]\!)^\circ) \\ = & \quad \{\text{converse contravariant; symmetry of ?}\} \\ & \Lambda\,((\![empty, cat]\!) \cdot p?)^\circ\,. \end{array}$$

and again, we need to express $p?$ as a catamorphism, so that we can apply the Promotion Theorem. If it is the case that p has a suffix-derivative, q say, then we can do so:

$$p? \;=\; (\![empty, snoc \cdot q?]\!).$$

As before, we now manipulate $(\![empty, cat]\!) \cdot [empty, snoc \cdot q?]$ to achieve the condition for the Promotion Theorem:

$$\begin{array}{ll} & (\![empty, cat]\!) \cdot [empty, snoc \cdot q?] \\ = & \quad \{\text{universal property for catamorphisms; coproduct}\} \\ & [empty, cat] \cdot \mathsf{F}(\![empty, cat]\!) \cdot \alpha^\circ \cdot \alpha \cdot (id + q?) \end{array}$$

$$
\begin{array}{ll}
= & \{\text{definition of } \mathsf{F};\ \alpha \text{ an isomorphism}\} \\
& [\mathit{empty}, \mathit{cat}] \cdot (\mathit{id} + ((\![\mathit{empty}, \mathit{cat}]\!) \times \mathit{id})) \cdot (\mathit{id} + q?) \\
= & \{\text{assumption: } ((\![\mathit{empty}, \mathit{cat}]\!) \times \mathit{id}) \cdot q? = r? \cdot ((\![\mathit{empty}, \mathit{cat}]\!) \times \mathit{id})\} \\
& [\mathit{empty}, \mathit{cat}] \cdot (\mathit{id} + r?) \cdot (\mathit{id} + ((\![\mathit{empty}, \mathit{cat}]\!) \times \mathit{id})) \\
= & \{\text{coproduct; definition of } \mathsf{F}\} \\
& [\mathit{empty}, \mathit{cat} \cdot r?] \cdot \mathsf{F}(\![\mathit{empty}, \mathit{cat}]\!),
\end{array}
$$

and so by applying the Promotion Theorem, we obtain

$$(\![\mathit{empty}, \mathit{cat}]\!) \cdot p? \ = \ (\![\mathit{empty}, \mathit{cat} \cdot r?]\!),$$

to get

$$M \ = \ \mathit{min}\, R \cdot \Lambda (\![\mathit{empty}, \mathit{cat} \cdot r?]\!)^{\circ}.$$

Once we have the partition problem in this form, the dynamic programming or greedy theorems can now be applied. The assumption we made is equivalent to saying that $q\,(xs, a)$ can be expressed in the form $r\,(\mathit{flatten}(xs), a)$. That is, q depends only on the elements of the partition so far, not on the actual partitioning of the elements so far.

6.1 N-partitions

We have increased the number of partition problems that we can solve, but the above reasoning is not applicable to a certain common form of problem. Some partition problems have a restriction on the number of components allowed in a partition. For example, the length of a partition may be restricted to exactly N or at most N components. (We call partitions that are bounded in length *N-partitions*.) A predicate expressing such a length constraint has no suffix derivative in the form used above, so the above method cannot be used to generate N-partitions by using a power transpose of the converse of a catamorphism. Instead, we construct a catamorphism that includes the length constraint directly.

In the Dynamic Programming section, we noted that *flatten*ing a partition reconstructs the original list, and thus wrote *flatten* as a catamorphism to obtain

$$\mathit{parts} \ = \Lambda\, \mathit{flatten}^{\circ} = \Lambda (\![\mathit{empty}, \mathit{cat}]\!)^{\circ}.$$

Let us take the example where we want an upper bound for the length of the partition. We note that $\langle \mathit{flatten}, \mathit{Geq} \cdot \mathit{length}\rangle$ (where Geq is $\geq$) inputs a partition and outputs a pair consisting of the original list and an upper bound on the length of the partition, so we can generate all the partitions of a list with length at most a certain value using $\Lambda\,\langle \mathit{flatten}, \mathit{Geq} \cdot \mathit{length}\rangle^{\circ}$.

Now we need to express $\langle \mathit{flatten}, \mathit{Geq} \cdot \mathit{length}\rangle$ as a catamorphism. To do this we will first express $\mathit{Geq} \cdot \mathit{length}$ as a catamorphism. As we have that $\mathit{length} = ([\mathit{zero}, \mathit{succ} \cdot \mathit{outl}])$, we can use the Promotion Theorem to do this. Manipulating $\mathit{Geq} \cdot [\mathit{zero}, \mathit{succ} \cdot \mathit{outl}]$ we have the following

$$
\begin{array}{ll}
 & \mathit{Geq} \cdot [\mathit{zero}, \mathit{succ} \cdot \mathit{outl}] \\
= & \quad \{\text{coproduct}\} \\
 & [\mathit{Geq} \cdot \mathit{zero}, \mathit{Geq} \cdot \mathit{succ} \cdot \mathit{outl}] \\
= & \quad \{\mathit{Geq} \cdot \mathit{succ} = \mathit{succ} \cdot \mathit{Geq}\} \\
 & [\mathit{Geq} \cdot \mathit{zero}, \mathit{succ} \cdot \mathit{Geq} \cdot \mathit{outl}] \\
= & \quad \{\text{naturality of } \mathit{outl}\} \\
 & [\mathit{Geq} \cdot \mathit{zero}, \mathit{succ} \cdot \mathit{outl} \cdot (\mathit{Geq} \times \mathit{id})] \\
= & \quad \{\text{definition of } \mathsf{F}\} \\
 & [\mathit{Geq} \cdot \mathit{zero}, \mathit{succ} \cdot \mathit{outl}] \cdot \mathsf{F}\mathit{Geq}.
\end{array}
$$

Thus we now have that $\mathit{Geq} \cdot \mathit{length} = ([\mathit{Geq} \cdot \mathit{zero}, \mathit{succ} \cdot \mathit{outl}])$, by the Promotion Theorem. Because for this particular functor F we can express the split of two catamorphisms thus

$$
\langle ([U_1, U_2]), ([V_1, V_2])\rangle = ([\langle U_1, V_1\rangle, \langle U_2 \cdot (\mathit{outl} \times \mathit{id}), V_2 \cdot (\mathit{outr} \times \mathit{id})\rangle]),
$$

we use the catamorphism expressions for $\mathit{flatten}$ and $\mathit{Geq} \cdot \mathit{length}$ to obtain

$$
\begin{array}{ll}
 & \langle \mathit{flatten}, \mathit{Geq} \cdot \mathit{length}\rangle \\
= & \langle ([\mathit{empty}, \mathit{cat}]), ([\mathit{Geq} \cdot \mathit{zero}, \mathit{succ} \cdot \mathit{outl}])\rangle \\
= & ([\langle \mathit{empty}, \mathit{Geq} \cdot \mathit{zero}\rangle, \langle \mathit{cat} \cdot (\mathit{outl} \times \mathit{id}), \mathit{succ} \cdot \mathit{outl} \cdot (\mathit{outr} \times \mathit{id})\rangle]).
\end{array}
$$

$\Lambda\,\langle \mathit{flatten}, \mathit{Geq} \cdot \mathit{length}\rangle^{\circ}$ is now almost suitable for use in the Dynamic Programming or Greedy theorems, but for one detail. At each stage of an algorithm produced by one of the theorems is the step $\Lambda\, P^{\circ}$, which produces partial solutions. For the algorithm to work properly, we must ensure that the partial solutions produced are feasible. For this example, a partial solution $((y, n), z)$ is feasible if we can partition y into at most n partitions, and so we define the predicate

$$\mathsf{fsb}\,((y, n), z) \quad = \quad y \neq [\,] \;\Rightarrow\; n > 0.$$

We can now define the generator *atmost-n-parts* to be

$$\begin{aligned} &atmost\text{-}n\text{-}parts \;= \Lambda\,([P])^{\circ}, \\ &\text{where } P \;= \\ &\qquad [\langle empty, Geq \cdot zero \rangle, \langle cat \cdot (outl \times id), succ \cdot outl \cdot (outr \times id) \rangle \;\cdot\; \mathsf{fsb}?] \end{aligned}$$

Now that we have expressed *atmost-n-parts* in the form $\Lambda\,([P])^{\circ}$, the Dynamic Programming and Greedy Theorems may be applied to problems involving *atmost-n-parts*.

Furthermore, this can be generalized to find generators for partitions with exact or lower bound constraints. Using the same idea, we have that $\Lambda\,\langle flatten, length \rangle^{\circ}$ takes a list and a value and returns all partitions of length that value. Also, $\Lambda\,\langle flatten, Leq \cdot length \rangle^{\circ}$ takes a list and a value and returns all partitions of length at least that value. As before, we can express the halves of the splits as catamorphisms

$$\begin{aligned} length \quad &= \quad ([zero, succ \cdot outl]) \\ Leq \cdot length \quad &= \quad ([zero, (succ \,\cup\, id) \cdot outl]), \end{aligned}$$

and then with the addition of suitable predicates as before to prevent infeasible partial solutions, the two generators *n-parts* and *atleast-n-parts* can also be expressed in the form $\Lambda\,([P])^{\circ}$.

Thus we have the three generators *atmost-n-parts*, *n-parts* and *atleast-n-parts* that can be used in the Dynamic Programming and Greedy Theorems. Note that for any particular problem, further work may be needed before the theorems can be applied, as in the specification of a problem there may be a predicate that the N-partitions are filtered through, and so the predicate will have to be absorbed into the generator using the Promotion Theorem, as before.

The above also demonstrates the versatility of the new theorems as compared to the old theorems in the List Partitions paper [BM92c]. The old theorems were concerned only with the standard generator of partitions, *parts*, and could not have accommodated length-restricted partitions.

6.2 Example: The Packing Boxes Problem

The Packing Boxes problem concerns several items which have to be packed in individual boxes for the purposes of transportation. The boxes are to be constructed, and can be made in sizes to exactly fit items. The larger the size of the item, the greater the cost of constructing a box to fit that item exactly.

The constraint is that the box builder does not wish to make more than n templates for boxes, so that at most n sizes of box can be used. The builder does not mind which sizes are chosen. This means that some items may have to be packed in boxes bigger than their size. It is wished to decide how many boxes of which sizes to make so that the total cost is minimized.

This can be represented as a partition problem in the following way. Recall that the ordering of the items by size is the same as the ordering of the items by the costs of the boxes that fit them. Then we label the items $1 \dots m$ so that if c_i is the cost of making a box to fit the ith item, $c_1 \geq c_2 \dots \geq c_m$. The decision of how many boxes of which sizes to make can be represented as a partition of the list $[c_1, c_2, \dots c_m]$ with at most n components. A component $[c_i \dots c_j]$ of the partition represents items from i up to and including j all being packed in boxes fitting the ith item.

We now choose R to be $\leq_{cost}$, where

$$\begin{aligned} cost\ xs &= sum\,(map\ w\ xs) \\ w\ x &= hd\ x \times length\ x, \end{aligned}$$

and thus the problem can be specified as

$$M = min(\leq_{cost}) \cdot atmost\text{-}n\text{-}parts.$$

In order to use the Dynamic Programming Theorem on this problem, we first express the problem in the form

$$\begin{aligned} & M && = min\,(\leq_{cost}) \cdot \Lambda\,(\![P]\!)^{\circ} \\ where\ & P && = [\langle empty, Geq \cdot zero\rangle, \langle cat \cdot (outl \times id), succ \cdot outl \cdot (outr \times id)\rangle], \end{aligned}$$

and the remaining condition for the theorem is that α be monotonic with respect to $\leq_{cost}$. The α is the same as before, so this equates to the condition that $\leq_{cost}$ be suffix-stable. Again, this is a least weight partition problem, and cost functions of the form $sum \cdot (map\, w)$ are suffix-stable. Thus the Dynamic Programming Theorem applies, giving the recursive algorithm

$$D \;=\; min\,(\leq_{cost}) \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D) \cdot \Lambda\, P^{\circ}\,.$$

However, this recursive equation involving relations is still far away from a real program. As promised in the Dynamic Programming section, we shall now aim to reach the point where it is possible to implement this algorithm in a functional language, and to do this we shall introduce some operations involving lists. The list functor $\mathsf{L} : Rel \leftarrow Rel$ is defined

$$[x_1, \ldots, x_m]\, \mathsf{L}R\, [y_1, \ldots, y_n] \;\equiv\; n = m \;\wedge\; (\forall 1 \leq i \leq n :\; x_i R y_i)$$

so that in fact L is the familiar functional programming operator *map*. The relation $\in_{\mathsf{L}}$ is similar to the usual $\in$, except that it acts on lists rather than sets. A property of $\in_{\mathsf{L}}$ is that for any relation S,

$$\in_{\mathsf{L}} \cdot \mathsf{L}S \;\subseteq\; S \cdot \in_{\mathsf{L}}\,.$$

Now we work on the recursion equation that we have. Firstly we consider the generator $\Lambda\, ([P])^{\circ}$. This takes a pair consisting of a list and value $(x, n{+}1)$ to a set of partial solutions, either of the form $[\,]$ if x is the empty list, or otherwise of the form $((y, n), z)$, where $x\; cat\; (y, z)$ and $\mathsf{fsb}\,((y, n), z)$. Sets are not easy to implement in a functional language, but it is straightforward to construct a function *partials* that returns the list of partial solutions so that $\Lambda\, P^{\circ} = \Lambda{\in_{\mathsf{L}}} \cdot partials$.

Next we look at P, and as P is not in general implementable as a function we will use the following lemma to remove it

Lemma 3 *For any relation S,*

$$\Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \;\subseteq\; \mathsf{P}S \cdot \Lambda{\in_{\mathsf{L}}}\,.$$

Proof This can be proved as follows:

$$\begin{array}{ll}
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \;\subseteq\; \mathsf{P}S \cdot \Lambda{\in_{\mathsf{L}}} \\
\equiv & \quad \{\text{definition of } \mathsf{P}\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \;\subseteq\; (\in \backslash (S \cdot \in) \;\cap\; (\ni \cdot S)/\ni) \cdot \Lambda{\in_{\mathsf{L}}} \\
\equiv & \quad \{\Lambda{\in_{\mathsf{L}}} \text{ is a function; characterization of } \cap\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \;\subseteq \in \backslash (S \cdot \in) \cdot \Lambda{\in_{\mathsf{L}}} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \;\subseteq\; (\ni \cdot S)/\ni \;\cdot\; \Lambda{\in_{\mathsf{L}}}
\end{array}$$

The first inequation is proved

$$
\begin{aligned}
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \ \subseteq \ \in \backslash (S\cdot \in)\cdot \Lambda{\in_{\mathsf{L}}} \\
\equiv \quad & \{\text{quotient; } \Lambda{\in_{\mathsf{L}}} \text{ is a function}\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \ \subseteq \ \in \backslash (S\cdot \in \cdot \Lambda{\in_{\mathsf{L}}}) \\
\equiv \quad & \{\text{quotient}\} \\
& \in \cdot \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \ \subseteq \ S\cdot \in \cdot \Lambda{\in_{\mathsf{L}}} \\
\equiv \quad & \{\in / \Lambda \text{ cancellation property}\} \\
& \in_{\mathsf{L}} \cdot \mathsf{L}S \ \subseteq \ S\cdot \in_{\mathsf{L}} \\
\Leftarrow \quad & \{\text{property of } \in_{\mathsf{L}}\} \\
& \mathit{true},
\end{aligned}
$$

and the second inequation is proved

$$
\begin{aligned}
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \ \subseteq \ (\ni \cdot S)/\ni \cdot \Lambda{\in_{\mathsf{L}}} \\
\equiv \quad & \{\Lambda{\in_{\mathsf{L}}} \text{ is a function}\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \cdot (\Lambda{\in_{\mathsf{L}}})^{\circ} \subseteq \ (\ni \cdot S)/\ni \\
\equiv \quad & \{\text{quotient}\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \cdot (\Lambda{\in_{\mathsf{L}}})^{\circ} \cdot \ni \subseteq \ni \cdot S) \\
\equiv \quad & \{\text{converse; } \in / \Lambda \text{ cancellation property}\} \\
& \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}S \cdot \ni_{\mathsf{L}} \subseteq \ni \cdot S \\
\equiv \quad & \{\Lambda{\in_{\mathsf{L}}} \text{ is a function}\} \\
& \mathsf{L}S \cdot \ni_{\mathsf{L}} \subseteq \ (\Lambda{\in_{\mathsf{L}}})^{\circ} \cdot \ni \cdot S \\
\equiv \quad & \{\text{converse; } \in / \Lambda \text{ cancellation property}\} \\
& \mathsf{L}S \cdot \ni_{\mathsf{L}} \subseteq \ni_{el} \cdot S \\
\Leftarrow \quad & \{\text{converse; property of } \in_{\mathsf{L}}\} \\
& \mathit{true}
\end{aligned}
$$

□.

What we have so far is that

$$
\begin{aligned}
& \mathit{min}\,(\leq_{cost}) \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D)\cdot \Lambda\, P^{\circ} \\
= \quad & \{\text{expression for } P^{\circ}\} \\
& \mathit{min}\,(\leq_{cost}) \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D)\cdot \Lambda{\in_{\mathsf{L}}} \cdot \mathit{partials}
\end{aligned}
$$

$$\begin{aligned} \supseteq \quad & \{\text{Lemma } 3\} \\ & min\,(\leq_{cost}) \cdot \Lambda{\in_{\mathsf{L}}} \cdot \mathsf{L}(\alpha \cdot \mathsf{F}D) \cdot partials. \end{aligned}$$

Expressing $min\,(\leq_{cost}) \cdot \Lambda{\in_{\mathsf{L}}}$ as a catamorphism seems like a good idea, so we first express $\Lambda{\in_{\mathsf{L}}}$ thus

$$\Lambda{\in_{\mathsf{L}}} \quad = \quad (\![empty_{\{\}}, union \cdot (id \times single_{\{\}})]\!),$$

where *union* takes the union of two sets, $empty_{\{\}}$ returns the empty set, and $single_{\{\}}$ is the function returning a singleton set. Applying the Promotion Theorem (Theorem 1) to get the *min* inside the catamorphism then yields that

$$(\![\infty, leftmin_{cost}]\!) \quad \subseteq \quad min\,(\leq_{cost}) \cdot \Lambda{\in_{\mathsf{L}}},$$

where $leftmin_{cost}$ is the function that takes a pair of values and returns either the one of least cost, or the left-hand argument if both have equal cost.

We now have that

$$\begin{aligned} & min\,(\leq_{cost}) \cdot \mathsf{P}(\alpha \cdot \mathsf{F}D) \cdot \Lambda\, P^{\circ} \supseteq \\ & \quad (\![\infty, leftmin_{cost}]\!) \cdot \mathsf{L}(\alpha \cdot \mathsf{F}D) \cdot partials. \end{aligned}$$

Fortunately, the Knaster-Tarski fixpoint theorem tells us that a least solution of the recursion equation using the second expression is a subset of the least solution of the recursion equation given us by the Dynamic Programming Theorem. This means that we can now finally implement the algorithm using a functional programming language.

Also, in our implementation, a further optimization can be made that is specific to the packing boxes problem. If at any point the input to the algorithm is (x, n) with $n \geq length\, x$, the minimum cost of packing the items in x is to construct an individual box for each item.

Figure 1 gives an implementation of the algorithm in the functional programming language Orwell.

```
> dynamic ([],n)    = []
> dynamic ((a:x),n) = map single (a:x),          if #(a:x)<=n
>                   = minimum (map recurse
>                          (partials ((a:x),n)), otherwise

> single a = [a]

> minimum [x] = x
> minimum (x1:x2:xs) = x1, if cost(m)>=cost(x1)
>                    = m, otherwise
>                    where m = minimum (x2:xs)

> cost = sum . (map w)
> w x  = (hd x) * (#x)

> recurse ((y,n),z) = (dynamic (y,n)) ++ [z]

> partials (x,1)   = [(([],0),x)]
> partials (x,n+2) = zip (map (insert (n+2)) (inits x), tails x)

> insert (n+1) x = (x,n)

> inits [a]     = [[]]
> inits (a:b:x) = []: map (a:) (inits (b:x))

> tails [a]     = [[a]]
> tails (a:b:x) = (a:b:x) : (tails (b:x))
```

Figure 1: An Orwell program to solve the Packing Boxes problem.

7 Conclusion

The earlier work of Bird and de Moor in their paper on List Partitions [BM92c] has now been updated to the context of relations. In many ways the theory using relations has a more elegant appearance than the old squiggol style. Comparing the statements of the Dynamic Programming and Greedy Theorems with the previous corresponding theorems in squiggol shows that the relational versions are more concisely stated, and as section 6 shows, they are also are applicable to many data types, rather than just to the type of partitions. The proofs in a calculational style are also in a straightforward manner. At each step there is usually an obvious rule to apply, thus helping to eliminate the need for clever ideas resembling magicians pulling rabbits out of hats. Using relations seems to have many advantages over a purely functional style, because as well as the elegance and generalization which they provide, none of the original functional style is lost.

Possible continuations of this work include exploring ways of making such algorithms more efficient. Not all problems have a straightforward greedy algorithm solution, but many can be solved more efficiently than standard dyamic programming. Such a strategy is used in *nub theory*, where the number of partial solutions kept at each stage is kept to a minimum.

References

[BBM+91] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, 1991.

[BGJ89] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, July 1989.

[BM92a] R.S. Bird and O. de Moor. Between dynamic programming and greedy: Data compression. Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1992.

[BM92b] R.S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In *State-of-the-Art Seminar on Formal Program Development*, 1992.

[BM92c] R.S. Bird and O. de Moor. List partitions. *Formal Aspects of Computing*, 1992.

[BM92d] R.S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction*, 1992.

[BVvdW91] R.C. Backhouse, E. Voermans, and J.C.S.P. van der Woude. A relational theory of datatypes. In *EURICS Workshop on Calculational Theories of Program Structure*, 1991.

[BW88] R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[CKW89] A. Carboni, G.M. Kelly, and R.J. Wood. A 2–categorical approach to geometric morphisms. Research report 89–19, Department of Pure Mathematics, The University of Sydney, NSW 2006, Australia, 1989.

[KP81] D.E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice and experience*, pages 1119 – 1184, 1981.

[Mal90] G.R. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Groningen University, 1990.

[Moo92] O. de Moor. Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, 1992.

[Moo93] O. de Moor. Inductive types for predicate transformers. To appear in Information Processing Letters, 1993.

[Wag73] R.A. Wagner. Common phrases and minimum-space text storage. *Communications of the ACM*, 1973.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976. IBSN 0-13-022418-9.