Between Dynamic Programming and Greedy: Data Compression

Richard S. Bird and Oege de Moor
Oxford University Programming Research Group
11 Keble Road, Oxford OX1 3QD

September 14, 1995

Abstract

The derivation of certain algorithms can be seen as a hybrid form of dynamic programming and the greedy paradigm. We present a generic theorem about such algorithms, and show how it can be applied to the derivation of an algorithm for data compression.

1 Introduction

Dynamic programming is a technique for solving optimisation problems. A typical dynamic programming algorithm proceeds by decomposing the input in all possible ways, recursively solving the subproblems, and combining optimal solutions to subproblems into an optimal solution for the whole problem. The *greedy paradigm* is also a technique for solving optimisation problems and differs from dynamic programming in that only one decomposition of the input is considered. Such a decomposition is usually chosen to maximise some objective function, and this explains the term 'greedy'. In our earlier work, we have characterised the use of dynamic programming and the greedy paradigm, using the categorical calculus of relations to derive two general theorems [4, 5, 19]. Briefly, these theorems say that dynamic programming is applicable if a certain monotonicity condition is satisfied. If an additional monotonicity property holds, then the choice of decompositions can be narrowed down to a single candidate, thus obtaining a greedy algorithm.

In this paper, we investigate algorithms that are in between dynamic programming and greedy in that they consider multiple (but not necessarily all) decompositions of the input. Our motivating example is a method for data compression that was first suggested by Storer and Szymanski [23]. This example is explained in Section 2. We then proceed to introduce the categorical calculus of relations in

Section 3. In Section 4, we review our earlier results about dynamic programming and greedy algorithms, and then present a theorem about the hybrid case that is the topic of this paper. The hybrid theorem is applied in Section 5 to obtain an efficient algorithm for the data compression example. In Section 6 we briefly discuss some directions for future research.

2 Data compression via textual substitution

The data to be compressed is a finite, possibly empty, sequence of symbols, simply called a *string*. We shall also consider *words*, which are non-empty sequences of symbols. The compressed data is a sequence of *code* elements where a code is either a symbol or a pointer. In a functional programming language, one might define the type of codes as follows:

$$code \quad ::= \quad Sym\ symbol \mid Pointer\ (string, word).$$

A pointer is a pair of sequences that identifies a substring of the string already processed. The second component, which is non-empty, is the substring in question. To make this precise, we describe the process of expanding a sequence of codes. The partial function *decode* takes a sequence of codes $[a_1, a_2, \ldots, a_n]$ and returns a string

$$((([\,] \oplus a_1) \oplus a_2) \oplus \ldots) \oplus a_n,$$

where

$$\begin{array}{rcl} x \oplus (Sym\ a) & = & x \mathbin{+\!\!+} [a] \\ x \oplus (Pointer(y, z)) & = & \begin{cases} x \mathbin{+\!\!+} z, & \text{if } (y \mathbin{+\!\!+} z)\ Prefix\ (x \mathbin{+\!\!+} z) \\ undefined, & \text{otherwise.} \end{cases} \end{array}$$

Here, $\mathbin{+\!\!+}$ denotes concatenation of a string and a word, and *Prefix* denotes the *proper prefix* relation given by

$$x\ Prefix\ z \quad = \quad \exists y : x \mathbin{+\!\!+} y = z$$

(y is a word and therefore non-empty). Note that *decode* is not a total function; in particular, if the first code element is a pointer, then *decode* is undefined since there is no y for which $(y \mathbin{+\!\!+} z)$ is a proper prefix of z.

For later convenience we have chosen to define pointers in terms of strings, but the success of data compression results from representing each pointer (y, z) as a pair of natural numbers (m, n), where m is the length of y and n is the length of z. For this new representation, the decoding of a pointer is given by

$$x \oplus (Pointer(m, n)) \quad = \quad x \otimes (m, n),$$

where

$$\begin{aligned} x \otimes (m, 0) &= x \\ x \otimes (m, n+1) &= (x \mathbin{+\!\!+} [x_m]) \otimes (m+1, n). \end{aligned}$$

Here x_m is the mth element of x (counting from 0). This change of representation yields a compact representation of strings. For instance, we have

$$\mathit{decode}\,[\text{‘a’}, (0, 9)] \;=\; \text{“aaaaaaaaaa”}.$$

A slightly more involved example is

$$\mathit{decode}\,[\text{‘a’}, \text{‘a’}, \text{‘b’}, (1, 3), \text{‘c’}, (1, 2)] \;=\; \text{“aabababacab”}.$$

Bearing the new representation of pointers in mind, we define the *size* of a sequence of codes by

$$\mathit{size}[c_1, c_2, \ldots, c_n] \;=\; (h\, c_1) + (h\, c_2) + \ldots + (h\, c_n),$$

where

$$\begin{aligned} h\,(\mathit{Sym}\ a) &= c \\ h\,(\mathit{Pointer}\,(y, z)) &= p. \end{aligned}$$

Here c is the size of a symbol — typically one byte, and p is the size of a pointer — typically four bytes (three bytes for the position, and one for the length). Note that c and p have to be estimated by the programmer who implements the algorithm on a particular computer system, for in practice the size of a pointer is not constant.

Now we can state our problem, which is to find an algorithm that computes

$$\mathit{compress}\, x \;=\; \mathit{min}(\mathit{size})\{\, y \mid \mathit{decode}\ y = x \,\}.$$

Note that *compress* is not really a function, for x may have multiple compressions of minimum size. For this reason, we shall interpret *compress* as a binary relation between input and output.

The data compression problem explained above might be of some practical interest. A naive implementation of the algorithm derived in this paper reduces the size of a typical ASCII file to 30% of its original size. While this is already quite acceptable, it does even better on files that contain a lot of repetitions: the source of Francis Borceux's macro package for drawing diagrams in LaTeX [6] is reduced to 17% of its original size. By comparison, the Unix utility *compress* only achieves a reduction of 28% on the same file. More detailed figures on the performance of the algorithm can be found in an appendix to this paper.

3 A calculus for solving optimisation problems

The calculus of relations presented here is based on category theory. The role of category theory is primarily one of organisation: it helps to structure the large collection of laws and axioms inherent in any calculus of relations, and also offers an economy of notation that would be difficult to achieve by other means. In the sequel, it will be assumed that the reader has a passing knowledge of categories, functors, natural transformations, products, coproducts and algebras for an endofunctor.

For expository reasons, we present the calculus of relations in set-theoretic terms, but we could also have taken an axiomatic approach. In particular, all the formal proofs in this paper carry through in a locally complete Boolean topos with a natural numbers object. Such topoi can be characterised as categories of relations carrying some additional structure [13].

3.1 Relations

A *relation* is a subset of a cartesian product. The basis of our calculus is the category *Rel* whose objects are sets, and whose arrows are relations. When A and B are sets, and $R \subseteq A \times B$, we write $R : A \leftarrow B$ and say that R is of type "A from B". This notation reflects the order of composition we will adopt. That is, if $R : A \leftarrow B$ and $S : B \leftarrow C$, then their composition is $R \cdot S : A \leftarrow C$. Composition of relations is defined in the usual way. Writing aRb as shorthand for $(a, b) \in R$, we have

$$a(R \cdot S)c \quad = \quad \exists b : aRb \wedge bSc.$$

For every set A, the identity arrow on A is denoted by $1_A : A \leftarrow A$. We will usually omit the subscript.

Inclusion of relations defines a partial order on the homsets of *Rel*. Composition of relations is monotonic with respect to this order:

$$(R_0 \subseteq R_1) \wedge (S_0 \subseteq S_1) \quad \Rightarrow \quad R_0 \cdot S_0 \subseteq R_1 \cdot S_1.$$

In categorical terms, this says that *Rel* is a *poset-enriched* category, a fact which is all-pervasive in our proofs and whose use will often be left implicit.

The category *Rel* is isomorphic to its own opposite. To wit, there exists a monotonic, contravariant isomorphism from *Rel* to itself, called the *converse* functor, and defined by

$$aR^\circ b \quad = \quad bRa.$$

Indeed, this defines a contravariant functor, for we have

$$1^\circ = 1 \quad \text{and} \quad (R \cdot S)^\circ = S^\circ \cdot R^\circ.$$

The converse functor is its own inverse: $(R^\circ)^\circ = R$. For this reason, the converse functor is sometimes called the *involution* on *Rel*.

For given sets A and B there exists a smallest relation of type $A \leftarrow B$, namely the empty relation 0. The empty relation is a zero of composition:

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

There is also a largest relation of type $A \leftarrow B$, written Π. This notation is inspired by the observation that Π is in fact the product $A \times B$.

The partial order on the homsets of *Rel* is a complete distributive lattice. In particular, any two relations of the same type have a *meet*, characterised by the universal property

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

The calculus of relations abounds in universal properties of this type. All such equivalences are instances of the more abstract concept of a *Galois connection*. A particularly readable introduction, which explains the applications to relations, is Aarts' Master's thesis [1]. Here we restrict ourselves to some more examples that will be useful in the sequel.

The *join* of two relations is dual to the meet. It is defined by the universal property

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

Meets distribute over joins, that is

$$(R \cup S) \cap T \;\; = \;\; (R \cap T) \cup (S \cap T).$$

The inclusion $(\supseteq)$ is an immediate consequence of the defining equivalences of meet and join.

An operator which has been largely neglected in the calculus of relations is *implication*. It is defined by

$$T \subseteq (R \Rightarrow S) \;\; \equiv \;\; (T \cap R) \subseteq S \; .$$

In later proofs, we shall refer to this universal property by the hint *Heyting*. Arend Heyting was a Dutch logician who studied algebraic structures that satisfy the above definition of implication.

So far we have defined three binary operators by means of a universal property: meet, join and implication. The only binary operator which did not play a role in such an equivalence was composition itself. The next universal property characterises *division* in terms of composition.

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

As a predicate, R/S can be defined by

$$a(R/S)b \ \equiv\ \forall c : bSc \Rightarrow aRc.$$

At this point, a word about notation is required. Meet, join, implication and division all have the same binding power. Composition, however, binds more tightly than all the other binary operators. For example, monotonicity of composition is expressed by the inequation

$$R \cdot (S \cap T) \ \subseteq\ R \cdot S \cap R \cdot T.$$

This inclusion cannot be strengthened to an equality, which sometimes leads to difficulties in proving that a meet is included in another relation. In such cases one can usually appeal to the so-called *modular law*

$$R \cdot S \cap T \ \subseteq\ (R \cap T \cdot S^\circ) \cdot S$$

to prove the desired inequation. There are many equivalent formulations of the modular law, and a discussion of various alternatives can be found in [3]. For our purposes the above form seems to be the most convenient. We shall discuss an application of the modular law below.

A *preorder* is a relation $R : A \leftarrow A$ that is reflexive and transitive:

$$1 \subseteq R \quad \text{and} \quad R \cdot R \subseteq R.$$

For every relation R, there exists a smallest preorder R^* containing R, called the *reflexive transitive closure* of R. This preorder is characterised by the property that for every preorder S

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

The transitive closure R^+ can be defined analogously and we have

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

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 of division, it suffices to prove

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

But this is immediate from the definition of R^*, provided $(T \cdot S)/T$ is a preorder. Reflexivity is proved as follows:

$$\begin{array}{ll}
 & 1 \subseteq (T \cdot S)/T \\
\equiv & \quad \{\text{universal division}\} \\
 & T \subseteq T \cdot S \\
\Leftarrow & \quad \{\text{monotonicity}\} \\
 & 1 \subseteq S \\
\equiv & \quad \{S \text{ preorder}\} \\
 & \mathit{true}.
\end{array}$$

Transitivity of $(T \cdot S)/T$ is proved in a similar manner.

3.2 Functions

The category of sets and relations has three subcategories of particular interest: the category of entire relations, the category of simple relations, and the category of total functions. Below we examine each of these subcategories in some more detail.

A relation $R : A \leftarrow B$ is said to be *entire* (or *total*) if

$$1 \quad \subseteq \quad R^\circ \cdot R.$$

In words, this means that for each $b \in B$, there exists an $a \in A$ such that aRb. It is easily checked that identity arrows are entire, and that the composition of two entire relations is again entire. Hence, entire relations do indeed form a subcategory of *Rel*.

A relation $R : A \leftarrow B$ is said to be *simple* (or a *partial function*) if

$$R \cdot R^\circ \quad \subseteq \quad 1.$$

Translated into ordinary set theory, this inequation says that for each $b \in B$ there exists at most one $a \in A$ such that aRb. Again it is trivial to check that this defines a subcategory of *Rel*. Partial functions enjoy various algebraic properties not shared by arbitrary relations. For example, composition with partial functions left-distributes through meets: if F is a partial function

$$(R \cap S) \cdot F \quad = \quad R \cdot F \cap S \cdot F.$$

The inclusion $(\subseteq)$ is an instance of monotonicity. Inclusion the other way round follows from the modular law:

$$R \cdot F \cap S \cdot F \;\subseteq\; (R \cap S \cdot F \cdot F^\circ) \cdot F \;\subseteq\; (R \cap S) \cdot F.$$

The last step in this calculation is valid only for partial functions.

A total partial function is simply called a *function*. The subcategory of functions in *Rel* is called *Fun* and we shall denote arrows of *Fun* by lower case identifiers. Functions can be shunted from one side of an inclusion to the other by means of the following equivalence:

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

Using the fact that converse is a monotonic isomorphism, we obtain the dual form of the above equivalence, namely

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

As an application of these equivalences, let us prove the following property of division:

$$(R/S) \cdot f \quad = \quad R/(f^{\circ} \cdot S).$$

We reason

$$\begin{array}{ll} & T \subseteq (R/S) \cdot f \\ \equiv & \quad \{\text{shunting function } f\} \\ & T \cdot f^{\circ} \subseteq R/S \\ \equiv & \quad \{\text{universal division}\} \\ & T \cdot f^{\circ} \cdot S \subseteq R \\ \equiv & \quad \{\text{universal division}\} \\ & T \subseteq R/(f^{\circ} \cdot S). \end{array}$$

The next lemma shows how functions can be characterised without reference to the converse functor and will prove useful below.

Lemma 2 *Let $R : A \leftarrow B$ in Rel. Then R is a function if and only if there exists a relation $S : B \leftarrow A$ such that $R \cdot S \subseteq 1$ and $1 \subseteq S \cdot R$. Furthermore, if S satisfies these inequations, then $S = R^{\circ}$.*

A proof of this lemma may be found in [8]. Another important property is the fact that every relation can be factorised into two functions:

Lemma 3 *Let $R : A \leftarrow B$ in Rel. Then there exists functions $f : A \leftarrow C$ and $g : B \leftarrow C$ such that $R = f \cdot g^{\circ}$.*

The set–theoretic proof of this lemma is easy: take C to be the subset of $A \times B$ characterised by R and define $f(a, b) = a$ and $g(a, b) = b$.

3.3 Relators

When we restrict our attention to total functions, many of the basic types that occur in programming can be defined by simple universal properties. For instance, products are defined by the universal property

$$h = \langle f, g \rangle \quad \equiv \quad (outl \cdot h = f) \wedge (outr \cdot h = g).$$

This equivalence defines the triple $(outl, outr, \langle \_, \_ \rangle)$ up to isomorphism. At the point level, the function $\langle f, g \rangle$ could be defined by

$$\langle f, g \rangle \; a \quad = \quad (f \; a, g \; a).$$

There exists an obvious generalisation of this construction to arbitrary relations, namely

$$(b, c) \langle R, S \rangle a \quad = \quad bRa \wedge cSa.$$

Unfortunately, the defining equivalence for products is no longer valid in a relational context: take, for example, R to be the empty relation, and let S be the identity arrow. We have $outr \cdot \langle 0, 1 \rangle = 0$. It might appear that one could weaken the equations in the equivalence to inclusions, but the equivalence then no longer characterises product up to isomorphism. (A counter-example will be discussed below.) Yet, it is important to know in what sense $\langle R, S \rangle$ is unique, and what algebraic identities can be used in reasoning about it.

A nice solution to this problem was proposed by Carboni, Kelly and Wood [9]. They define constructions like products at the functional level, and then show how these can be extended to arbitrary relations in some canonical way. The key step is to show how functors $Fun \leftarrow Fun$ extend to functors $Rel \leftarrow Rel$. (Throughout the paper, functors will be denoted by sans serif identifiers.)

Lemma 4 *Let* $\mathsf{G} : Rel \leftarrow Rel$ *be a monotonic functor. Then* G *preserves converse, i.e.,* $\mathsf{G}(R^\circ) = (\mathsf{G}R)^\circ$.

Proof Since G is monotonic, it follows from lemma 2 that G preserves functions and converses of functions. Hence, by lemma 3, G preserves converses of arbitrary relations:

$$\begin{array}{ll} & \mathsf{G}((h \cdot k^\circ)^\circ) \\ = & \quad \{\text{converse}\} \\ & \mathsf{G}(k \cdot h^\circ) \\ = & \quad \{\mathsf{G} \text{ functor}\} \\ & \mathsf{G}k \cdot \mathsf{G}(h^\circ) \end{array}$$

$$
\begin{array}{cl}
= & \{\mathsf{G} \text{ preserves converses of functions}\} \\
 & \mathsf{G}k \cdot (\mathsf{G}h)^\circ \\
= & \{\text{similar to above}\} \\
 & (\mathsf{G}(h \cdot k^\circ))^\circ.
\end{array}
$$

Lemma 5 *Let* $\mathsf{F} : \mathit{Fun} \leftarrow \mathit{Fun}$*. Then there exists at most one monotonic functor* $\mathit{Rel} \leftarrow \mathit{Rel}$ *which coincides with* F *on functions.*

Proof Let G and G' be monotonic functors from *Rel* to *Rel*, both of which coincide with F on functions. According to lemma 3 there exist h and k such that $R = h \cdot k^\circ$. But now

$$\mathsf{G}R = \mathsf{G}(h \cdot k^\circ) = (\mathsf{G}h) \cdot (\mathsf{G}k)^\circ = (\mathsf{F}h) \cdot (\mathsf{F}k)^\circ = \mathsf{G}'R.$$

When $\mathsf{F} : \mathit{Fun} \leftarrow \mathit{Fun}$ has an extension to relations, it is called a *relator*. Not every functor on functions is a relator, but all examples that arise in programming are. (A theorem to this effect will be stated below.) Analogous results hold for bifunctors $\mathit{Fun} \leftarrow (\mathit{Fun} \times \mathit{Fun})$.

As an example of a relator, consider the product functor $(\times)$. Its extension to relations is given by

$$(R \times S) \;=\; (\mathit{outl}^\circ \cdot R \cdot \mathit{outl}) \cap (\mathit{outr}^\circ \cdot S \cdot \mathit{outr}).$$

The operator $\langle R, S\rangle$ can now be defined by

$$\langle R, S\rangle \;=\; (R \times S) \cdot \langle 1, 1\rangle \;=\; (\mathit{outl}^\circ \cdot R) \cap (\mathit{outr}^\circ \cdot S).$$

Since $\times$ is a relator, this is the canonical generalisation of $\langle\_,\_\rangle$ from functions to relations. Using the universal property of meets and the shunting rule for functions, we obtain that

$$T \subseteq \langle R, S\rangle \;\equiv\; (\mathit{outl} \cdot T \subseteq R) \wedge (\mathit{outr} \cdot T \subseteq S).$$

Note that this is precisely the defining equivalence of products, with equality replaced by inclusion. The situation is not specific to products: it can be shown that any adjunction at the functional level extends to a so-called *local adjunction* between categories of relations [15].

The coproduct functor $(+)$ is also a relator, and its extension is dual to that of the product functor:

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

where *inl* and *inr* are the coproduct injections. Correspondingly, we define

$$[R, S] \;=\; [1,1] \cdot (R + S) \;=\; (R \cdot \mathit{inl}^\circ) \cup (S \cdot \mathit{inr}^\circ)$$

This defines a proper coproduct in *Rel*. That is, we have the usual universal property defining $[R, S]$:

$$T = [R, S] \;\equiv\; (T \cdot \mathit{inl} = R) \wedge (T \cdot \mathit{inr} = S).$$

Because *Rel* is isomorphic to its own opposite, the triple $(\mathit{inl}^\circ, \mathit{inr}^\circ, [(\_)^\circ, (\_)^\circ]^\circ)$ is a product in *Rel*. This triple satisfies the same equivalence as the extended product of *Fun*:

$$T \subseteq [R^\circ, S^\circ]^\circ \;\equiv\; (\mathit{inl}^\circ \cdot T \subseteq R) \wedge (\mathit{inr}^\circ \cdot T \subseteq S),$$

and therefore this equivalence does not characterise the extended product. Backhouse *et al.* [3] give a detailed discussion of the properties of products and coproducts in a calculus of relations similar to ours.

The class of *polynomial* functors *Fun* $\leftarrow$ *Fun* is inductively defined by the following clauses:

- The identity functor $1 : \mathit{Fun} \leftarrow \mathit{Fun}$ is polynomial.
- For every set A, the constant functor K_A (which sends every arrow to the identity on A) is polynomial.
- If F and G are polynomial, then $(\mathsf{F} + \mathsf{G})$ and $(\mathsf{F} \times \mathsf{G})$ defined by

  $$\begin{aligned}(\mathsf{F} + \mathsf{G})h &= \mathsf{F}h + \mathsf{G}h \quad \text{and} \\ (\mathsf{F} \times \mathsf{G})h &= \mathsf{F}h \times \mathsf{G}h\end{aligned}$$

  are polynomial. The composite $\mathsf{F} \cdot \mathsf{G}$ is polynomial as well.

Lemma 6 *All polynomial functors are relators.*

The categorical proof of this lemma consists of a single line: in a topos, polynomial functors preserve pullbacks and epis. The interested reader is referred to [9] for details.

3.4 Catamorphisms

Let $\mathcal{C}$ be a category, and let $\mathsf{F} : \mathcal{C} \leftarrow \mathcal{C}$ be a functor. An *algebra* for F is an arrow of type $A \leftarrow \mathsf{F}A$. Let $f : A \leftarrow \mathsf{F}A$ and $g : B \leftarrow \mathsf{F}B$ be algebras. A *homomorphism* $f \leftarrow g$ is an arrow $h : A \leftarrow B$ such that

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

Clearly, identity arrows are homomorphisms, and the composition of two homomorphisms yields another homomorphism. It follows that we have a category of algebras, where the arrows are homomorphisms. For many functors F, this category has an initial object, which will be denoted by α. For any other algebra f the unique homomorphism from α to f will be denoted by $(\![f]\!)$. In formulae, we have the universal property

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

Homomorphisms of the form $(\![f]\!)$ are called catamorphisms [17]. In later calculations, we shall refer to their defining property by the hint *cata*.

Lemma 7 *Let* $\mathsf{F} : \mathit{Fun} \leftarrow \mathit{Fun}$ *be a relator with an initial algebra* α*. Then* α *is also an initial algebra of* $\mathsf{F} : \mathit{Rel} \leftarrow \mathit{Rel}$*.*

A proof of this lemma is given in [20].

3.5 Sequences

To illustrate functors and catamorphisms we will consider three kinds of sequences that play a crucial role in the data compression example. The data type of strings can be defined in a functional language by the declaration

$$\mathit{string} \quad ::= \quad \mathit{empty} \mid \mathit{snoc}\,(\mathit{string}, \mathit{symbol}).$$

The constant function *empty* stands for the empty string, and *snoc* is the binary operator that places an element at the end of a string. Formally, the data type of strings is defined as the initial algebra $[\mathit{empty}, \mathit{snoc}]$ of the functor

$$\begin{aligned} \mathsf{F}\,A &= \top + (A \times E) \\ \mathsf{F}\,h &= 1 + (h \times 1), \end{aligned}$$

where $\top$ is the terminator of *Fun*, and E is the type of symbols. The set of all strings over E is denoted by E^*.

Words are non-empty sequences of symbols. The data type of words is defined as the initial algebra $[one, snoc]$ of the functor

$$\begin{aligned} \mathsf{W}\,A &= E + (A \times E) \\ \mathsf{W}\,h &= 1 + (h \times 1). \end{aligned}$$

The set of all words over E is denoted by E^+. Note that we use the same name for *snoc* on strings and symbols; the precise meaning can always be deduced from type considerations.

Strings and words can be concatenated. The binary operator $cat : E^* \leftarrow (E^* \times E^+)$ is defined as the unique solution of

$$\begin{aligned} cat \;=\;& snoc \cdot (1 \times one^\circ) \cup \\ & snoc \cdot (cat \times 1) \cdot assoc \cdot (1 \times snoc^\circ), \end{aligned}$$

where *assoc* is the natural isomorphism $(A \times B) \times C \leftarrow A \times (B \times C)$. The defining equation for *cat* can also be stated as two separate equations

$$\begin{aligned} cat \cdot (1 \times one) &= snoc \\ cat \cdot (1 \times snoc) &= snoc \cdot (cat \times 1) \cdot assoc. \end{aligned}$$

This formulation mirrors the usual definition of concatenation in functional programming. To see this, write $[a]$ for $(one\ a)$, $(+\!\!+)$ for *cat* and $(+\!\!+\!\!<)$ for *snoc*. We have

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

which is just a restatement of the relational combinator code.

The fact that the above equation has a unique solution is not immediate from the definition of strings as an initial algebra. The proof makes use of concepts that are beyond the scope of this paper. It is, in fact, similar to the proof of Lawvere's recursion theorem for natural numbers in a cartesian closed category. More details can be found in [16].

The partial function *Init* takes a string and removes the last element. It is defined by

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

In fact, this defines two functions: one for strings and one for words. Using this definition we can show

$$\begin{aligned} Init \cdot cat &= cat \cdot (1 \times Init) \cup outl \cdot (1 \times one^o) \\ Init^\circ \cdot cat &= cat \cdot (1 \times Init^\circ). \end{aligned}$$

For example, here is a proof of the second identity:

$$\begin{array}{cl}
 & cat \cdot (1 \times Init^\circ) \\
= & \quad \{\text{def. } Init\} \\
 & cat \cdot (1 \times snoc \cdot outl^\circ) \\
= & \quad \{\text{product}\} \\
 & cat \cdot (1 \times snoc) \cdot (1 \times outl^\circ) \\
= & \quad \{\text{def. } cat\} \\
 & snoc \cdot (cat \times 1) \cdot assoc \cdot (1 \times outl^\circ) \\
= & \quad \{\text{since } assoc \cdot (1 \times outl^\circ) = outl^\circ\} \\
 & snoc \cdot (cat \times 1) \cdot outl^\circ \\
= & \quad \{\text{naturality of } outl\} \\
 & snoc \cdot outl^\circ \cdot cat \\
= & \quad \{\text{def. } Init\} \\
 & Init^\circ \cdot cat.
\end{array}$$

The proof of the other identity is similar and is omitted. Both these facts about *cat* will be used below in the formal treatment of data compression.

We can also define code sequences as instances of sequences, but it will turn out to be more convenient to distribute the two kinds of code element into the definition of the data type. We can define codes in a functional language by the declaration

$$codes \quad ::= \quad empty \mid snocs\,(codes, symbol) \mid snocp(codes, (string, word)).$$

Formally, the data type of codes is defined as the initial algebra $[empty, snocs, snocp]$ of the functor

$$\begin{array}{rcl}
\mathsf{F}\,A & = & \top + (A \times E) + (A \times (E^* \times E^+)) \\
\mathsf{F}\,h & = & 1 + (h \times 1) + (h \times 1).
\end{array}$$

The function *Init* can also be defined on codes; we have

$$Init \quad = \quad outl \cdot (snocs \cup snocp)^\circ.$$

3.6 Monotonicity

In the study of optimisation problems, the interaction between algebras and preorders is often expressed in terms of *monotonicity* properties. Consider for example the fact that addition of natural numbers is monotonic:

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

Addition is an algebra for the *squaring* functor, which sends every arrow k to $(k \times k)$. Writing *plus* instead of $+$ and *Leq* instead of $\leq$, one can translate the above implication into relational combinators:

$$Leq \times Leq \quad \subseteq \quad plus^\circ \cdot Leq \cdot plus.$$

This motivates the following definition. Let $R : A \leftarrow A$ and let F be a relator. An algebra f of F is *monotonic* with respect to R if

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

Because f is a function, the above inclusion can also be written in the more symmetrical form

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

Alternatively, both occurrences of f can be shunted to the right–hand side:

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

3.7 Powersets

Recall that we decided to define relations as subsets of a cartesian product. This was a more or less arbitrary decision, since there are many other ways of representing relations in set theory, for example as boolean-valued functions of two arguments, or as set-valued functions. In this section, we shall show how the notion of powersets may be *defined* by exploiting the isomorphism between relations and set-valued functions.

Let A and B be sets. Below we shall give a universal property which defines (up to isomorphism) a three-tuple $(\mathsf{P}A, \Lambda, \in)$. Here $\mathsf{P}A$ is a set, called the *powerset* of A. The operator Λ takes a relation of type $A \leftarrow B$, and produces a function of type $\mathsf{P}A \leftarrow B$. Finally, $\in : A \leftarrow \mathsf{P}A$ is called the *membership* relation on A. The defining property is this: for all $R : A \leftarrow B$ and $f : \mathsf{P}A \leftarrow B$ we have

$$f = \Lambda R \quad \equiv \quad \in \cdot f = R.$$

If you are familiar with adjunctions, you will notice that this says that the embedding *Rel* $\leftarrow$ *Fun* has a right adjoint.

Under the usual interpretation of powersets, the operator Λ is given by the set comprehension

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

Indeed, one might say that the definition of Λ is merely a restatement of the comprehension principle in Zermelo-Fraenkel set theory.

Let us now see how the universal property of Λ can be used to reason about simple identities in set theory. The following calculation shows how R cancels ΛR; in later proofs we shall refer to this result as Λ-*cancellation*.

$$\begin{array}{ll} & R \cdot (\Lambda R)^{\circ} \subseteq \in \\ \equiv & \quad \{\Lambda R \text{ function}\} \\ & R \subseteq \in \cdot \Lambda R \\ \equiv & \quad \{\text{universal } \Lambda\} \\ & R \subseteq R \\ \equiv & \quad \{\text{inclusion is reflexive}\} \\ & \mathit{true}. \end{array}$$

It is a general principle in category theory that any suitable operator on objects can be extended to a functor. Since we have just introduced the operator P on objects, it is natural to look for a corresponding functor. The *powerset* functor $\mathsf{P} : \mathit{Fun} \leftarrow \mathit{Fun}$ is defined by

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

This functor is a relator, and its extension is given by

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

A more thorough discussion of P and its algebraic properties can be found in De Moor's thesis [19].

3.8 Minimums and Minimals

Let R be a relation of type $A \leftarrow A$. The *minimum* of R is a relation of type $A \leftarrow \mathsf{P}A$ defined by

$$\mathit{min}\,R \quad = \quad \in \cap\, (R/\ni),$$

where $\ni$ denotes the converse of $\in$. This very concise formula is a restatement of the familiar definition in set theory: a is a minimum element of x if a is an element of x, and a is a lower bound of x. In terms of predicates, one could write this as follows:

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

The definition of *min* and its algebraic properties were first studied in the context of categorical relations by Brook [7].

The above rule for $min\, R$ gives an expression in terms of meet and division, both of which have been defined by means of a universal property. The following rule exploits that fact, and it gives a universal property for reasoning about *min* itself:

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

Note that when $S = \in$, this is a restatement of our earlier definition of *min*, because $\Lambda\!\in = 1$.

When informally explaining the formulae in our calculus, it is sometimes helpful to attach a loose operational interpretation to some of the expressions. For example, the composite $min\, R \cdot \Lambda\, S$ can be read as follows. The function $\Lambda\, S$ generates a set of combinatorial objects. Subsequently, the relation $min\, R$ selects a minimum element from this set of candidates, possibly in a nondeterministic fashion.

For useful computation, it is important that non-empty sets do contain minimum elements under R. That is, we require that R be a *well-founded* relation. One way of formulating this requirement is by the condition

$$\in \;\;\subseteq\;\; R^{\circ} \cdot min\, R.$$

Translated, this condition says that if there is some element $a \in x$, then there is a b such that $b\; R\; a$ and $b\; (min\, R)\; x$. The length order on sequences is well-founded, but the prefix order is not. The concept of a well-founded relation is to be contrasted with the concept of a *well-supported* relation, defined below.

In the sequel, we shall often encounter subexpressions of the form $min\, R \cdot \mathsf{P}\, S$. This composite can be viewed as a nondeterministic mapping that takes a set as its argument. The relation $\mathsf{P}S$ applies the mapping S to each element of that set, and finally $minR$ selects a minimum element from the set that has been generated in this way. The equation below is sometimes helpful in reasoning about the pattern $min\, R \cdot \mathsf{P}\, S$:

$$min\, R \cdot \mathsf{P}\, S \;\;=\;\; ((R \cdot S)/\ni) \cap (S \cdot \in), \quad \text{provided } R \text{ is reflexive.}$$

At the time of writing, we have been unable to find a snappy proof of this equation. In later calculations, we shall refer to it by the hint *min*-P *fusion*.

Let $R : A \leftarrow A$. The *minimal* of R is a relation of type $A \leftarrow \mathsf{P}A$ defined by

$$mnl\, R \;\;=\;\; min(R^{\circ} \Rightarrow R).$$

Again this reflects the usual definition: $a(mnl\, R)x$ holds if a is an element of x, and whenever bRa we also have aRb. To appreciate the difference between minimum and minimal, consider the case that R is the empty relation 0. We have $min\, 0 = 0$, but $mnl\, 0 = \in$.

Say that R is *connected* when

$$R \cup R^\circ \;\; = \;\; \Pi.$$

An example of a connected relation is the length ordering on finite sequences. The prefix ordering on finite sequences is not connected. When R is connected, $min\, R = mnl\, R$. Clearly, it suffices to prove that $R = (R^\circ \Rightarrow R)$. The inclusion $(\subseteq)$ is obvious, and the reverse inclusion may be proved as follows:

$$\begin{array}{ll}
 & R^\circ \Rightarrow R \\
= & \quad \{\text{identity of meet}\} \\
 & (R^\circ \Rightarrow R) \cap \Pi \\
= & \quad \{\text{assumption: } R \text{ connected}\} \\
 & (R^\circ \Rightarrow R) \cap (R \cup R^\circ) \\
= & \quad \{\text{distributive lattice}\} \\
 & ((R^\circ \Rightarrow R) \cap R) \cup ((R^\circ \Rightarrow R) \cap R^\circ) \\
\subseteq & \quad \{\text{monotonicity}\} \\
 & R \cup ((R^\circ \Rightarrow R) \cap R^\circ) \\
\subseteq & \quad \{\text{Heyting}\} \\
 & R \cup R \\
= & \quad \{\text{join idempotent}\} \\
 & R.
\end{array}$$

A relation is said to be *well-supported* if for every set, each element is above a minimal element. That is,

$$\in \;\; \subseteq \;\; R^\circ \cdot mnl\, R.$$

The length order on sequences is well-supported, and so is the prefix order. The empty relation is not well-supported, nor is the usual order on the integers. Unlike well-foundedness, well-supportedness is remarkably stable under common constructions on preorders.

Lemma 8 *If R and S are well-supported preorders, then so are $R \cap S$ and $R \cap (R^\circ \Rightarrow S)$. Furthermore, for any function f, the conjugation of R by f (*i.e.* $f^\circ \cdot R \cdot f$) is a well-supported preorder if R is. Finally, if F is polynomial, then $\mathsf{F}R$ is well-supported if R is.*

The construction $T = (R \cap (R^\circ \Rightarrow S))$ is perhaps slightly unfamiliar. Informally, aTb holds provided aRb and whenever a and b are equivalent in R, we have aSb.

As an example, take R to be the length order on finite sequences, and define S to be the sum order. We have

$$\begin{array}{lll}[1,2,3] & (mnl(R \cap (R^\circ \Rightarrow S))) & \{\, [1,2,3],[1,1,1,1], \\ & & \phantom{\{\,}[4,5,6,7], \\ & & \phantom{\{\,}[3,4,5],[5,6,8]\,\}.\end{array}$$

4 Theorems for paradigms

As explained in the introduction, our aim is to provide concise theorems that capture the use of common paradigms in algorithm design. This pursuit is not new; it is one of the driving forces in combinatorial optimisation [14] and operations research [12]. Our approach differs from earlier work in the style of presentation in that we seek to express and derive results in a formal calculus. The research effort is still very much in its infancy, and so far we have only studied two common paradigms, namely dynamic programming and greedy algorithms. This section summarises two theorems about these techniques, and it presents the hybrid theorem that is the topic of this paper. None of the theorems is expressed in quite the most general terms, but they will be sufficient for our purposes.

4.1 Dynamic programming

It is well known that the use of dynamic programming is governed by the *principle of optimality*. Roughly speaking, the principle of optimality is satisfied if an optimal solution is composed of optimal solutions to subproblems. It should be noted that the principle of optimality is not universally true, and it might be better to speak of a *property* instead of a *principle*. In the theorem below, the optimality property is expressed as a monotonicity condition.

Theorem 1 *[19] Let $T = \mathit{min}\, R \cdot \Lambda\, (\![P]\!)^\circ$, where R is a preorder. If α is monotonic with respect to R, then the least solution D of the equation*

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

satisfies $D \subseteq T$.

The conclusion of the Dynamic Programming theorem may be interpreted as follows. The function $\Lambda\, P^\circ$ splits the argument in all possible ways, and this yields a set of decompositions. For each of these decompositions, $\mathsf{F}\, D$ solves the subproblems recursively, and the optimal solutions to subproblems are composed into a solution for the whole problem with α. Hence, the composite

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

generates a set of candidates, and $\mathit{min}\, R$ chooses an optimal element from this set. The dynamic programming theorem does not require R to be well-founded, but if the least solution D is to be non-empty, then well-foundedness is important.

4.2 Greedy algorithms

There exists an obvious way to improve the efficiency of the dynamic programming scheme sketched in the preceding section: rather than considering *all* decompositions of the argument, we might cleverly choose a particular one, known to lead to an optimal solution. This is the idea underlying the next result, which is called the *Greedy Theorem.*

Theorem 2 *[4] Let $T = \mathit{min}\, R \cdot \Lambda(\llbracket P \rrbracket)^{\circ}$, where R is a preorder. Suppose that α is monotonic with respect to R, and $Q = \mathsf{F}\llbracket P \rrbracket \cdot \alpha^{\circ}$ satisfies $S \cdot Q \subseteq Q \cdot R$ for some S. Then the (unique) solution G of the equation*

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

satisfies $G \subseteq T$.

In analogy with the Dynamic Programming theorem, the conclusion of the Greedy Theorem can be read in an operational way. The composite $\mathit{min}\, S \cdot \Lambda\, P^{\circ}$ returns a single decomposition of the input. This decomposition consists of subproblems, which are recursively solved by $\mathsf{F} G$. Finally, the solutions to these subproblems are composed into a solution for the whole problem by α. Again, the statement and proof of the Greedy Theorem does not require R and S to be well-founded, but for computational use these conditions are necessary.

4.3 Between dynamic programming and greedy

The Dynamic Programming and Greedy Theorems represent extreme cases: either one considers all decompositions, or one considers only a single decomposition. What about the case where an algorithm considers multiple decompositions, but not necessarily all of them? It was this question that motivated the theorem stated below. For want of a better name, we shall call it the *Hybrid Theorem.*

Theorem 3 *Let $T = \mathit{min}\, R \cdot \Lambda(\llbracket P \rrbracket)^{\circ}$, where R is a preorder. Suppose that α is monotonic with respect to R, and $Q = \mathsf{F}\llbracket P \rrbracket \cdot \alpha^{\circ}$ satisfies $S \cdot Q \subseteq Q \cdot R$ for some S. Furthermore, suppose that S is well-supported. Then the least solution H of*

$$H \;=\; \mathit{min}\, R \cdot \mathsf{P}\,(\alpha \cdot \mathsf{F}\, H) \cdot \Lambda(\mathit{mnl}\, S) \cdot \Lambda\, P^{\circ}$$

satisfies $H \subseteq T$.

It may come as a surprise that the hybrid case of dynamic programming and greedy algorithms seems to require stronger conditions than the Greedy Theorem itself. Why consider a hybrid algorithm when apparently weaker conditions guarantee the applicability of the greedy theorem? The brief answer is that well-foundedness is a stronger condition than well-supportedness, and we need well-foundedness in order to use the greedy theorem in practice. In other words, unless S is well-founded the unique solution G of

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

does not have the same domain as $T = \mathit{min}\, R \cdot \Lambda\, (\![P]\!)^{\circ}$. When the domain of G is strictly smaller than the domain of T, we cannot use G to obtain a computer program that implements T. The situation is analogous to the use of 'miracles' in the refinement calculus studied by Back, Morgan and others [2, 21].

Before turning to the proof of this theorem, let us remark that the Dynamic Programming Theorem is a special case of the Hybrid Theorem. If we take $S = 1$, then S is well-supported, $S \cdot Q \subseteq Q \cdot R$, and $\Lambda(\mathit{mnl}\, S) = \Lambda \in = 1$. So the Dynamic Programming Theorem follows.

Turning to the proof of the Hybrid Theorem, we aim to show

$$T \quad \supseteq \quad \mathit{min}\, R \cdot \mathsf{P}(\alpha \cdot \mathsf{F}\, T) \cdot V$$

where $V = \Lambda\, \mathit{mnl}\, S \cdot \Lambda\, P^{\circ}$. By the universal min rule, this inequation can be split into two separate proof obligations:

$$\mathit{min}\, R \cdot \mathsf{P}\, (\alpha \cdot \mathsf{F}\, T) \cdot V \quad \subseteq \quad (\![P]\!)^{\circ} \tag{1}$$

and

$$\mathit{min}\, R \cdot \mathsf{P}\, (\alpha \cdot \mathsf{F}\, T) \cdot V \cdot (\![P]\!) \quad \subseteq \quad R. \tag{2}$$

The first proof obligation can be discharged as follows:

$$\begin{array}{ll}
 & \mathit{min}\, R \cdot \mathsf{P}\, (\alpha \cdot \mathsf{F}\, T) \cdot V \\
\subseteq & \quad \{\text{min-}\mathsf{P}\text{ fusion}\} \\
 & \alpha \cdot \mathsf{F}\, T \cdot \in \cdot\, V \\
= & \quad \{\text{def. } V\text{, universal } \Lambda\} \\
 & \alpha \cdot \mathsf{F}\, T \cdot \mathit{mnl}\, S \cdot \Lambda\, P^{\circ} \\
\subseteq & \quad \{\text{def. } T\text{, universal min (twice)}\} \\
 & \alpha \cdot \mathsf{F}\, (\![P]\!)^{\circ} \cdot P^{\circ} \\
= & \quad \{\text{cata}\} \\
 & (\![P]\!)^{\circ}.
\end{array}$$

To prove inequation (2), we reason

$$\begin{array}{ll}
& \mathit{min}\, R \cdot \mathsf{P}\,(\alpha \cdot \mathsf{F}\,T) \cdot V \cdot (\!|P|\!) \\
\subseteq & \quad \{\text{min-}\mathsf{P}\text{ fusion}\} \\
& (R \cdot \alpha \cdot \mathsf{F}\,T)/\ni \cdot V \cdot (\!|P|\!) \\
= & \quad \{\text{def. } V\text{, fun-division, universal } \Lambda\} \\
& (R \cdot \alpha \cdot \mathsf{F}\,T)/(\mathit{mnl}\,S)^\circ \cdot \Lambda\,P^\circ \cdot (\!|P|\!) \\
= & \quad \{\text{cata, def. } Q\} \\
& (R \cdot \alpha \cdot \mathsf{F}\,T)/(\mathit{mnl}\,S)^\circ \cdot \Lambda\,P^\circ \cdot P \cdot Q \\
\subseteq & \quad \{\Lambda\text{-cancellation}\} \\
& (R \cdot \alpha \cdot \mathsf{F}\,T)/(\mathit{mnl}\,S)^\circ \cdot \ni \cdot Q \\
\subseteq & \quad \{S\text{ well-supported}\} \\
& (R \cdot \alpha \cdot \mathsf{F}\,T)/(\mathit{mnl}\,S)^\circ \cdot (\mathit{mnl}\,S)^\circ \cdot S \cdot Q \\
\subseteq & \quad \{\text{universal division}\} \\
& R \cdot \alpha \cdot \mathsf{F}\,T \cdot S \cdot Q \\
\subseteq & \quad \{\text{greedy condition}\} \\
& R \cdot \alpha \cdot \mathsf{F}\,T \cdot Q \cdot R \\
= & \quad \{\text{def. } Q\text{, } F\text{ functor}\} \\
& R \cdot \alpha \cdot \mathsf{F}\,(T \cdot (\!|P|\!)) \cdot \alpha^\circ \cdot R \\
\subseteq & \quad \{\text{def. } T\text{, universal min, } \Lambda\text{-cancellation}\} \\
& R \cdot \alpha \cdot \mathsf{F}\,R \cdot \alpha^\circ \cdot R \\
\subseteq & \quad \{\alpha\text{ monotonic}\} \\
& R \cdot R \cdot R \\
\subseteq & \quad \{R\text{ transitive}\} \\
& R.
\end{array}$$

This completes the proof of the theorem.

5 An algorithm for data compression

Recall the data compression problem that formed the motivation for this paper. In this section, we first rephrase the specification in the calculus of relations. The specification of the problem does not include a choice for the relation S. Having defined S, we then verify the conditions of the Hybrid Theorem. Finally, it is indicated how the result leads to an efficient program in an imperative programming language like Modula-2.

5.1 Specification

Let E denote the set of symbols. Sequences over E are defined as in section 3.5. Recall that the data type of code sequences is defined slightly differently as the initial algebra of the functor

$$\begin{aligned} \mathsf{F}\,A &= \top + (A \times E) + A \times (E^* \times E^+) \\ \mathsf{F}\,h &= 1 + (h \times 1) + h \times 1. \end{aligned}$$

Codes are decoded by

$$\mathit{Decode} = (\![\mathit{empty}, \mathit{snoc}, \mathit{append} \cap \mathit{Ok}]\!),$$

where

$$\begin{aligned} \mathit{append} &= \mathit{cat} \cdot (1 \times \mathit{outr}) \\ \mathit{Ok} &= (\mathit{Init}^+)^\circ \cdot \mathit{cat} \cdot \mathit{outr}. \end{aligned}$$

The relation $\mathit{append} \cap \mathit{Ok}$ is a partial function which takes a pair $(x, (y, z))$, and returns $x \mathbin{+\!\!+} z$ provided $y \mathbin{+\!\!+} z$ is a proper prefix of $x \mathbin{+\!\!+} z$.

The size of a sequence of codes is also given by a catamorphism

$$\mathit{size} = (\![\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p)]\!),$$

where zero is the constant function returning 0, and c and p are given constant functions.

The function size induces a preorder R, defined by

$$R = \mathit{size}^\circ \cdot \mathit{Leq} \cdot \mathit{size},$$

where $\mathit{Leq} = \mathit{outl} \cdot \mathit{plus}^\circ$ is the usual order on natural numbers.

The problem that we seek to solve is

$$\mathit{Compress} = \mathit{min}\, R \cdot \Lambda\, \mathit{Decode}^\circ.$$

5.2 Choosing S

For general c and p it is not possible to determine at each stage whether it is better to pick a symbol or a pointer. In other words, we cannot hope for a greedy algorithm. On the other hand, it is possible to choose between pointers: a pointer (y, z) is better than (y', z') whenever z is longer than z' because this means a longer portion of the input is consumed at each step. More precisely, if $(x, (y, z))$ and $(x', (y', z'))$ are splittings of the same string, then $x \mathbin{+\!\!+} z = x' \mathbin{+\!\!+} z'$, so if z is longer than z', then x is shorter than x'. Indeed, we can define S in part by saying

$(x,(y,z))S(x',(y',z'))$ if x is a prefix of x'. If $x = x'$ we can arbitrarily prefer shorter values of y. This leads to the specification $S = 1 + 1 + W$, where

$$\begin{array}{rcl} W & = & Z \cap (Z^\circ \Rightarrow \mathit{outr}^\circ \cdot Z \cdot \mathit{outr}) \\ Z & = & \mathit{outl}^\circ \cdot \mathit{Init}^* \cdot \mathit{outl}. \end{array}$$

In words, $(x,(y,z))\,W(x',(y',z'))$ if x is a prefix of x', and y is a prefix of y' whenever $x = x'$. The relation S is a well-supported preorder, since Init^* has these properties, and the construction of W preserves them (lemma 8).

For the purposes of verifying the conditions of the theorem, the rather complicated definition of S above can be simplified; we require only that $S \subseteq V\,\mathit{Init}^*$, where we introduce the abbreviation

$$V\,X \quad = \quad 1 + 1 + \mathit{outl}^\circ \cdot X \cdot \mathit{outl}.$$

We have $1 \subseteq V\,1$ and $V\,X \subseteq V\,Y$ whenever $X \subseteq Y$. More important is the following fact about V, which we will refer to below as the *swapping rule*:

$$X \cdot Y \subseteq Y \cdot Z \quad \text{implies} \quad V\,X \cdot \mathsf{F}\,Y \subseteq \mathsf{F}\,Y \cdot V\,Z.$$

Moreover, the same implication holds when $\subseteq$ is replaced by $\supseteq$. The proof is as follows:

$$\begin{array}{cl} & V\,X \cdot \mathsf{F}\,Y \\ = & \quad \{\text{definitions}\} \\ & (1 + 1 + \mathit{outl}^\circ \cdot X \cdot \mathit{outl}) \cdot (1 + (Y \times 1) + (Y \times 1)) \\ = & \quad \{\text{coproduct}\} \\ & 1 + (Y \times 1) + \mathit{outl}^\circ \cdot X \cdot \mathit{outl} \cdot (Y \times 1) \\ = & \quad \{\text{naturality } \mathit{outl}\} \\ & 1 + (Y \times 1) + \mathit{outl}^\circ \cdot X \cdot Y \cdot \mathit{outl} \\ \subseteq & \quad \{\text{assumption}\} \\ & 1 + (Y \times 1) + \mathit{outl}^\circ \cdot Y \cdot Z \cdot \mathit{outl} \\ = & \quad \{\text{naturality } \mathit{outl}^\circ \text{ and coproduct}\} \\ & (1 + (Y \times 1) + (Y \times 1)) \cdot (1 + 1 + \mathit{outl}^\circ \cdot Z \cdot \mathit{outl}) \\ = & \quad \{\text{definitions}\} \\ & \mathsf{F}\,Y \cdot V\,Z. \end{array}$$

5.3 Checking the conditions

Transitivity of R is immediate and we shall prove monotonicity of α below. It remains to verify the hybrid condition

$$S \cdot \mathsf{F}\,\mathit{Decode} \cdot \alpha^\circ \quad \subseteq \quad \mathsf{F}\,\mathit{Decode} \cdot \alpha^\circ \cdot R.$$

But

$$\begin{array}{cl}
& S \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \\
\subseteq & \quad \{\text{assumption on } S\} \\
& V\, \mathit{Init}^* \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \\
\subseteq & \quad \{\text{swapping rule and claim: } \mathit{Init}^* \cdot \mathit{Decode} \subseteq \mathit{Decode} \cdot R\} \\
& \mathsf{F}\, \mathit{Decode} \cdot V\, R \cdot \alpha^\circ \\
\subseteq & \quad \{\text{claim: } V\, R \cdot \alpha^\circ \subseteq \alpha^\circ \cdot R\} \\
& \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \cdot R.
\end{array}$$

There are two claims in the above calculation. Let us first establish the second. Given that $R = \mathit{size}^\circ \cdot \mathit{Leq} \cdot \mathit{size}$, the second claim is equivalent – by shunting – to

$$\mathit{size} \cdot \alpha \cdot V\, R \subseteq \mathit{Leq} \cdot \mathit{size} \cdot \alpha.$$

The proof is as follows:

$$\begin{array}{cl}
& \mathit{size} \cdot \alpha \cdot V\, R \\
= & \quad \{\text{cata}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p)] \cdot \mathsf{F}\, \mathit{size} \cdot V\, R \\
\subseteq & \quad \{\text{swapping rule, since } \mathit{size} \cdot R \subseteq \mathit{Leq} \cdot \mathit{size}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p)] \cdot V\, \mathit{Leq} \cdot \mathsf{F}\, \mathit{size} \\
= & \quad \{\text{definition of } V \text{ and coproduct}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p) \cdot \mathit{outl}^\circ \cdot \mathit{Leq} \cdot \mathit{outl}] \cdot \mathsf{F}\, \mathit{size} \\
= & \quad \{\text{naturality of } \mathit{outl}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p) \cdot (\mathit{Leq} \times 1) \cdot \mathit{outl}^\circ \cdot \mathit{outl}] \cdot \mathsf{F}\, \mathit{size} \\
= & \quad \{\text{product and } p \text{ is constant}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (\mathit{Leq} \times 1) \cdot (1 \times p)] \cdot \mathsf{F}\, \mathit{size} \\
\subseteq & \quad \{\mathit{plus} \text{ is monotonic on } \mathit{Leq}\} \\
& [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{Leq} \cdot \mathit{plus} \cdot (1 \times p)] \cdot \mathsf{F}\, \mathit{size} \\
\subseteq & \quad \{1 \subseteq \mathit{Leq} \text{ and coproduct}\} \\
& \mathit{Leq} \cdot [\mathit{zero}, \mathit{plus} \cdot (1 \times c), \mathit{plus} \cdot (1 \times p)] \cdot \mathsf{F}\, \mathit{size} \\
= & \quad \{\text{cata}\} \\
& \mathit{Leq} \cdot \mathit{size} \cdot \alpha.
\end{array}$$

This result also establishes that α is monotonic with respect to R, i.e. $\alpha \cdot \mathsf{F}\, R \subseteq R \cdot \alpha$, for we have

$$\mathsf{F}\, R \subseteq V\, 1 \cdot \mathsf{F}\, R = \mathsf{F}\, 1 \cdot V\, R = V\, R.$$

We are now left with showing $\mathit{Init}^* \cdot \mathit{Decode} \subseteq \mathit{Decode} \cdot R$. Using Lemma 1 this is equivalent to $\mathit{Init} \cdot \mathit{Decode} \subseteq \mathit{Decode} \cdot R$. This is the most difficult part of the proof.

We begin

$$\begin{array}{cl}
 & \mathit{Init} \cdot \mathit{Decode} \\
= & \quad \{\text{cata}\} \\
 & \mathit{Init} \cdot [\mathit{empty}, \mathit{snoc}, \mathit{append} \cap \mathit{Ok}] \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \\
= & \quad \{\text{coproduct}\} \\
 & [\mathit{Init} \cdot \mathit{empty}, \mathit{Init} \cdot \mathit{snoc}, \mathit{Init} \cdot (\mathit{append} \cap \mathit{Ok})] \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ.
\end{array}$$

Now, $\mathit{Init} \cdot \mathit{empty} = 0$ and $\mathit{Init} \cdot \mathit{snoc} = \mathit{outl}$. For the final term $\mathit{Init} \cdot (\mathit{append} \cap \mathit{Ok})$, we reason

$$\begin{array}{cl}
 & \mathit{Init} \cdot \mathit{append} \\
= & \quad \{\text{definition of } \mathit{append}\} \\
 & \mathit{Init} \cdot \mathit{cat} \cdot (1 \times \mathit{outr}) \\
= & \quad \{\text{property of } \mathit{Init} \cdot \mathit{cat}\} \\
 & (\mathit{cat} \cdot (1 \times \mathit{Init}) \cup \mathit{outl} \cdot (1 \times \mathit{one}^\circ)) \cdot (1 \times \mathit{outr}) \\
= & \quad \{\text{distributing } (1 \times \mathit{outr})\} \\
 & \mathit{cat} \cdot (1 \times \mathit{Init}) \cdot (1 \times \mathit{outr}) \cup \mathit{outl} \cdot (1 \times \mathit{one}^\circ) \cdot (1 \times \mathit{outr}) \\
\subseteq & \quad \{\text{product, introducing } J = 1 \times (1 \times \mathit{Init})\} \\
 & \mathit{cat} \cdot (1 \times \mathit{outr}) \cdot J \cup \mathit{outl} \\
= & \quad \{\text{definition of } \mathit{append}\} \\
 & \mathit{append} \cdot J \cup \mathit{outl}.
\end{array}$$

Hence

$$\begin{array}{cl}
 & \mathit{Init} \cdot (\mathit{append} \cap \mathit{Ok}) \\
\subseteq & \quad \{\text{distributing } \mathit{Init} \text{ and above}\} \\
 & (\mathit{append} \cdot J \cup \mathit{outl}) \cap \mathit{Init} \cdot \mathit{Ok} \\
\subseteq & \quad \{(\cap, \cup) \text{ distributive lattice}\} \\
 & (\mathit{append} \cdot J \cap \mathit{Init} \cdot \mathit{Ok}) \cup \mathit{outl} \\
\subseteq & \quad \{\text{modular law}\} \\
 & (\mathit{append} \cap \mathit{Init} \cdot \mathit{Ok} \cdot J^\circ) \cdot J \cup \mathit{outl} \\
= & \quad \{\text{see below}\} \\
 & (\mathit{append} \cap \mathit{Ok}) \cdot J \cup \mathit{outl}
\end{array}$$

In the last step we used

$$\mathit{Init} \cdot \mathit{Ok} \cdot J^\circ \quad = \quad \mathit{Ok}.$$

This equation is proved as follows:

$$\begin{array}{ll}
 & \mathit{Init} \cdot \mathit{Ok} \cdot J^\circ \\
= & \quad \{\text{definition of } \mathit{Ok} \text{ and } J\} \\
 & \mathit{Init} \cdot (\mathit{Init}^+)^\circ \cdot \mathit{cat} \cdot \mathit{outr} \cdot (1 \times (1 \times \mathit{Init}^\circ)) \\
= & \quad \{\text{naturality of } \mathit{outr}\} \\
 & \mathit{Init} \cdot (\mathit{Init}^+)^\circ \cdot \mathit{cat} \cdot (1 \times \mathit{Init}^\circ) \cdot \mathit{outr} \\
= & \quad \{\text{fact about } \mathit{cat}\} \\
 & \mathit{Init} \cdot (\mathit{Init}^+)^\circ \cdot \mathit{Init}^\circ \cdot \mathit{cat} \cdot \mathit{outr} \\
= & \quad \{\text{definition of transitive closure, } \mathit{Init}^\circ \cdot \mathit{Init} = \mathit{id}\} \\
 & (\mathit{Init}^+)^\circ \cdot \mathit{cat} \cdot \mathit{outr} \\
= & \quad \{\text{definition of } \mathit{Ok}\} \\
 & \mathit{Ok}.
\end{array}$$

Having established these results, we can proceed with the main task:

$$\begin{array}{ll}
 & \mathit{Init} \cdot \mathit{Decode} \\
\subseteq & \quad \{\text{above}\} \\
 & [0, \mathit{outl}, (\mathit{append} \cap \mathit{Ok}) \cdot J \cup \mathit{outl}] \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \\
= & \quad \{\text{coproduct}\} \\
 & [\mathit{empty}, \mathit{snoc}, \mathit{append} \cap \mathit{Ok}] \cdot (0 + 0 + J) \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \cup \\
 & [0, \mathit{outl}, \mathit{outl}] \cdot \mathsf{F}\, \mathit{Decode} \cdot \alpha^\circ \\
= & \quad \{\text{coproduct}\} \\
 & [\mathit{empty}, \mathit{snoc}, \mathit{append} \cap \mathit{Ok}] \cdot \mathsf{F}\, \mathit{Decode} \cdot (0 + 0 + J) \cdot \alpha^\circ \cup \\
 & \mathit{Decode} \cdot [0, \mathit{outl}, \mathit{outl}] \cdot \alpha^\circ \\
= & \quad \{\text{cata}\} \\
 & \mathit{Decode} \cdot \alpha \cdot (0 + 0 + J) \cdot \alpha^\circ \cup \mathit{Decode} \cdot [0, \mathit{outl}, \mathit{outl}] \cdot \alpha^\circ.
\end{array}$$

The proof is now completed by showing that both $\alpha \cdot (0+0+J) \cdot \alpha^\circ$ and $[0, \mathit{outl}, \mathit{outl}] \cdot \alpha^\circ$ are included in R. But

$$0 + 0 + J \subseteq V\, 1 \subseteq V\, R$$

since R is reflexive, so the first inclusion follows from our earlier result that $\alpha \cdot V\, R \cdot \alpha^\circ \subseteq R$.

Since

$$[0, outl, outl] \cdot \alpha^\circ = outl \cdot (snocs \cup snocp)^\circ = Init$$

the second inclusion follows from the result that $Init \subseteq R$, or equivalently

$$size \cdot Init \quad \subseteq \quad Leq \cdot size.$$

In words, *size* is an ascending function on code sequences. The proof is

$$\begin{array}{ll}
 & Leq \cdot size \cdot (snocs \cup snocp) \\
= & \quad \{\text{cata}\} \\
 & Leq \cdot plus \cdot (size \times c \cup size \times p) \\
= & \quad \{\text{def. } Leq\} \\
 & outl \cdot plus^\circ \cdot plus \cdot (size \times c \cup size \times p) \\
\supseteq & \quad \{plus \text{ entire}\} \\
 & outl \cdot (size \times c \cup size \times p) \\
= & \quad \{\text{naturality of } outl\} \\
 & size \cdot outl.
\end{array}$$

This completes the verification of the conditions of the Hybrid Theorem.

5.4 Implementation in Modula-2

The Hybrid Theorem only yields a recursion equation and the implementation of this recursion equation in a particular programming language has not been addressed. Although we believe that the calculus presented here can be extended to reason about this aspect of program development, we shall take an informal approach.

In the preceding section, it was shown that $Compress \supseteq H$, where H is the least solution of

$$T \quad = \quad min\, R \cdot \mathsf{P}\,(\alpha \cdot \mathsf{F}\, T) \cdot \Lambda\,(mnl\, S) \cdot \Lambda\, P^\circ.$$

To find an implementation of H, we first have to find a concrete expression for $mnl\, S \cdot \Lambda\, P^\circ$. We shall not go into the details of this calculation, as it is of a highly mechanical nature. We write *inl*, *inm* and *inr* for the respective coproduct injections. The result of elaborating $mnl\, S \cdot \Lambda\, P^\circ$ is

$$\begin{array}{rcl}
mnl\, S \cdot \Lambda\, P^\circ & = & inl \cdot empty^\circ \cup \\
 & & inm \cdot snoc^\circ \cup \\
 & & inr \cdot Mrt.
\end{array}$$

Here $Mrt = mnl\ W \cdot \Lambda\,(append \cap Ok^{\circ})$ is a partial function that returns the *maximum repeated tail*. In words, the maximum repeated tail of a string x is the longest suffix of x that also occurs as a contiguous subword of $Init\, x$. More accurately speaking, Mrt returns a pointer to the first occurrence of this suffix.

It now follows that $F \subseteq H$, where F is the total function defined by:

$$\begin{array}{rcl}
F\,[\,] & = & [\,] \\
F\,(x \mathbin{+\!\!\!<} a) & = & \left\{ \begin{array}{ll} (F\,x \mathbin{+\!\!\!<}_s a) & \text{if } (y,p)\text{undefined} \\ (F\,x \mathbin{+\!\!\!<}_s a) \sqcap (F\,y \mathbin{+\!\!\!<}_p (y,p)) & \text{otherwise} \end{array} \right. \\
 & & \text{where } (y,p) = Mrt\,(x \mathbin{+\!\!\!<} a).
\end{array}$$

The binary operator $\sqcap$ returns whichever of its arguments has the lesser *size*. If the arguments have equal *size*, it returns the left-hand argument. The recursion scheme for F is easily implemented in an imperative programming language by means of *tabulation*. The result of doing this is displayed in figure 1, which contains a program in Modula-2 [24] for computing the total function F. This program maintains a separate data structure for the computation of Mrt. The procedure *Empty* initialises this data structure, and *Append* inserts a new symbol. We shall not go into the details of the algorithm for computing Mrt. An elegant solution may be found in Crochemore's paper on repetitions [11].

6 Discussion

This paper presented an experiment in the algebra of optimisation. We have applied our earlier work to a significant example, carrying out the proofs in minute detail. Although the derivation is quite satisfactory, this case study has revealed several issues that need to be addressed in future research.

Many of the proofs require little or no invention. This is an indication of the success of our research effort: we aimed from the outset to make matters as simple as possible. It is precisely this feature, however, that presents a problem when communicating the results of our work. What is needed is a more efficient way of communicating formal proofs, namely by means of proof *tactics* instead of proof *steps*. There is a great deal of work available on proof tactics in connection with automatic theorem proving [22], and we foresee no difficulties in taking advantage of that work. Note, however, that our motivation comes from a concern with effective communication, and not from a desire to automate the derivation process.

A related concern is the implementation of our abstract results in existing programming languages. Here the calculations are of such an uninspiring nature that they could be left to a compiler. We propose, therefore, to design and implement a relational programming language based on the concepts of our calculus. Again, there exists a substantial body of existing work that can serve as a starting point,

```
MODULE Compact;

FROM InOut IMPORT Done, Read;

FROM Symbols IMPORT Symbol;

FROM MaxRepTail IMPORT Empty, Append, Mrt;

FROM Code IMPORT C, P, MaxLen, Code, SnocS, SnocP, EmitCodes;

VAR
      Pos, Len, I : CARDINAL;
      A           : Symbol;
      Sizes       : ARRAY[0..MaxLen] OF CARDINAL;
      Codes       : ARRAY[1..MaxLen] OF Code;


BEGIN
   Empty;
   Sizes[0] := 0;
   I := 1;
   Read(A);
   WHILE Done AND (I 6 MaxLen) DO
       Append(A);
       Mrt(Pos, Len);
       IF (Len = 0) OR ((Sizes[I − 1] + C) 6 (Sizes[I − Len] + P))
       THEN Sizes[I] := Sizes[I − 1] + C;
              SnocS(Codes[I],A)
       ELSE Sizes[I] := Sizes[I − Len] + P;
             SnocP(Codes[I],Pos,Len)
       END;
       I := I + 1;
       Read(A);
   END;
   EmitCodes(Codes,I)
END Compact.
```

Figure 1: Modula 2 program for data compression.

namely the research in categorical programming languages (*e.g.* [10]). So far, however, that work has been confined to a strictly functional setting. Also, it does not incorporate pragmatic concepts like *memoisation* [18], which will be essential in our applications.

References

[1] C. Aarts. Galois connections presented calculationally. Eindhoven University of Technology, 1992.

[2] R.J.R. Back. A calculus of refinements for program derivation. *Acta Informatica*, 25:593–624, 1988.

[3] R.C. Backhouse, P. Hoogendijk, E. Voermans, and J.C.S.P. van der Woude. A relational theory of datatypes. Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands., June 1992.

[4] R.S. Bird and O. de Moor. From dynamic programming to greedy algorithms. To appear in Procs. State-of-the-Art Seminar on Formal Program Development, Rio de Janeiro, 1992.

[5] R.S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. To appear in Procs. Mathematics of Program Construction, Oxford, 1992.

[6] F. Borceux. User's guide for the *diagram* macros. UCL, Louvain-la-Neuve, Belgium, 1990.

[7] T. Brook. *Order and Recursion in Topoi*, volume 9 of *Notes on Pure Mathematics*. Australian National University, Canberra, 1977.

[8] A. Carboni and S. Kasangian. Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, 33:259–267, 1984.

[9] A. Carboni, G.M. Kelly, and R.J. Wood. A 2–categorical approach to geometric morphisms, I. *Cahiers de Topologie et Geometrie Differentielle Categoriques*, 32(1):47–95, 1991.

[10] R. Cockett and T. Fukushima. Draft: About charity. Dept. of Computer Science, University of Calgary, Calgary, Alberta, Canada. Available via anonymous ftp from cpsc.ucalgary.ca., 1991.

[11] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[12] J.G. Ecker and M. Kupferschmid. *Introduction to Operations Research*. John Wiley, 1988.

[13] P.J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North–Holland, 1990.

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

[15] C.B. Jay. Local adjunctions. *Journal of Pure and Applied Algebra*, 53:227–238, 1988.

[16] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[17] L. Meertens. Paramorphisms. To appear, Formal Aspects of Computing, 1990.

[18] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

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

[20] O. de Moor. Inductive data types for predicate transformers. To appear, Information Processing Letters, 1992.

[21] C.C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[22] L. Paulson. *Logic and Computation*. Cambridge University Press, 1988.

[23] J.A. Storer and T.G. Szymanski. Data compression via textual substitution. *Journal of the Association for Computing Machinery*, 29(4):928–951, 1982.

[24] N. Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer–Verlag, 1983.

Appendix: performance of algorithm

The table below presents some figures about the performance of the algorithm derived in this paper. All files are in ASCII; our implementation does not deal with binary files. The program simply tags codes with an extra bit to indicate whether they are symbols or pointers. Symbols are written in 7 bits; pointers have a fixed length of 31 bits (23 for the position and 8 for the length). It follows that our program can only cope with files of length $< 2^{23}$ bytes. To make sure that the length of a pointer can be written in 8 bits, the maximum repeated tail is restricted to a maximum of 255 symbols. It follows that our implementation has $C = 1$ and $P = 4$.

The first column in the table below gives a short description of the file, and the second column gives its size in bytes. The third column gives the size of the output of our program as a percentage of the original size. The fourth column of the table shows the result of applying Huffman coding to the output of our program; again the size is given as a percentage of the original file size. For comparison, the fifth column gives the result of the Unix utility *compress*.

description	original	our program	+ Huffman	compress
chapter 4 of De Moor's thesis	85933	36.75	30.92	38.38
part 1 of De Moor's thesis	94165	44.68	37.04	42.54
part 2 of De Moor's thesis	261669	31.86	27.75	34.81
paper on list partitions	45969	47.63	39.06	45.49
parser in Modula 2	66149	40.26	33.80	41.04
term rewriting in Orwell	18762	42.86	35.97	39.74
directory of Orwell programs	38595	27.42	23.01	32.46
directory of OBJ3 programs	15504	19.75	19.75	33.19
Borceux's diagram macros	76282	19.49	17.09	28.22
Lamport's LaTeX package	267736	37.05	31.31	35.21
BibTeX database	43426	34.46	29.01	37.86
part of database PRG library	150000	33.98	29.13	38.85
email directory	40384	50.98	41.40	42.91
average	92660	35.94	31.44	37.89

From these figures, we may conclude that the algorithm does very well for highly repetitive data. The improvement is, however, only worthwhile in an environment where the additional complexity of the compression process does not matter. An example of such an environment is an *ftp* site, where the same compressed file is transmitted many times over.