A CALCULUS OF FUNCTIONS FOR PROGRAM DERIVATION

by

Richard Bird

Oxford University
Computing Laboratory
Programming Research Group-Library
8-11 Keble Road
Oxford OX1 3QD
Oxford (0865) 54141

Technical Monograph PRG-64

December 1987

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

A Calculus of Functions for Program Derivation

R.S. Bird
Programming Research Group, Oxford University.[1]

1. Introduction.

This paper is about how to calculate programs. We introduce a notation for describing functions, outline a calculus for manipulating function descriptions, and prove two general theorems for implementing certain functions efficiently. Like useful calculi in other branches of mathematics, the calculus of functions consists of a body of knowledge expressed as basic algebraic identities, technical lemmas and more general theorems. We illustrate the calculational approach to program construction by posing and solving a simple problem about coding sequences.

In presenting this work we wish to argue:

(i) that some, perhaps many, algorithms can be derived by a process of systematic calculation from their specifications;

(ii) that an appropriate framework for this activity can be based on notation for describing functions, notation that is wide enough to express both specifications and implementations;

(iii) that an effective calculus of program derivation must be built upon useful general theorems relating certain forms of expression to common patterns of computation, theorems that—in the main—we still lack.

By way of motivation, we start in Section 2 by posing the problem we want to solve. The notational framework is introduced in Section 3. In that section we also state (mostly without proof) a number of simple algebraic laws about functions. In Section 4 we apply these laws to our example problem until we arrive at a point where a more substantial theorem is required. This theorem is stated, together with a second theorem, in Section 5. We believe that these two theorems are of the kind that will eventually prove important in establishing a useful calculus for program derivation.

2. Run-length encoding.

The problem we will use to illustrate our approach is that of run-length encoding. The idea behind run-length encoding is to represent a sequence of values (usually characters) in a compact form by coding each "run" of equal values by a pair consisting of the common value and the length of the

[1]Address: 11 Keble Road, Oxford, OX1 3QD, U.K.

run. For example,

$$code\ \text{"AABCCAAA"} = [(\text{'A'},2),(\text{'B'},1),(\text{'C'},2),(\text{'A'},3)]$$

The algorithm for computing *code* contains no surprises or subtleties. Our objective, however, is to derive the algorithm from its specification, using essentially the same kind of reasoning that a mathematician might employ in solving a problem in, say, formal integration.

To specify *code*, consider the inverse function, *decode*. A sequence of pairs can be decoded by "expanding" each pair to a sequence of values, and concatenating the results. Given *decode*, we can specify $(code\ x)$ as the *shortest* sequence of pairs that decodes to x.

To make this idea precise, suppose we define the *generalised inverse* f^{-1} of a function f by the equation

$$f^{-1}x = \{w \mid f\ w = x\}$$

Using notation from [1] (discussed further in the next section) we can now specify *code* as follows:

$$\begin{array}{rcl} code & :: & [\alpha] \to [(\alpha, N^+)] \\ code & = & \downarrow_{\#} / \cdot decode^{-1} \\ decode & = & +\!\!+/ \cdot exp* \\ exp\,(a,n) & = & K_a * [1\,..\,n] \end{array}$$

The first line of the specification gives the type of *code* as a function from lists of α-values to lists of pairs, each pair consisting of an α-value and a positive integer. The second line says that *code* is the functional composition of a function $(\downarrow_{\#} /)$ that selects the shortest in a set of sequences, and the generalised inverse of *decode*. The third line defines *decode* as the composition of a function $(+\!\!+/)$ that concatenates a list of lists together, and a function $(exp*)$ that applies *exp* to every element of a list. Finally, $exp\,(a,n)$ is obtained for a positive integer n by applying the constant-value function K_a to each element of the list $[1\,..\,n]$, thereby giving a list of n copies of a.

We shall return to this specification in Section 4. First we need to discuss notation in more detail, as well as introduce some of basic algebraic identities employed in the derivation.

3. Notation.

Our notation is basically that of [1] (see also [4], [5] and [6]) with some additions and modifications. In particular, functions are curried, so function

application associates to the left, and simple function arguments are written without brackets.

Lists and sets. We shall use square brackets "[" and "]" to denote lists, and braces "{" and "}" for sets. The symbol $+\!\!+$ denotes list concatenation and $\cup$ denotes set union. The functions $[\cdot]$ and $\{\cdot\}$ return singleton lists and sets, respectively. Thus

$$\begin{aligned} [\cdot]\, a &= [a] \\ \{\cdot\}\, a &= \{a\} \end{aligned}$$

To avoid clumsy subscripts, we shall use a° rather than K_a to describe the function that always returns the value a. In particular, $[\,]^\circ$ denotes the function defined by the equation

$$[\,]^\circ a = [\,]$$

A similar function for sets is used below.

We use $\#x$ to denote the length of the list x (or the size of the set x).

Selection. Suppose f is a numeric-valued function. The binary operator $\downarrow_f$ selects its left or right argument, depending on which has the smaller f-value:

$$\begin{aligned} x \downarrow_f y &= x, \quad \text{if } f\,x < f\,y \\ &= y, \quad \text{if } f\,y < f\,x \end{aligned}$$

Unless f is an injective function on the domain of values of interest, the operator $\downarrow_f$ is under-specified: if $x \neq y$ but $f\,x = f\,y$, then the value of $x \downarrow_f y$ is not specified (beyond the fact that it is one of x or y). For example, the value

$$\text{"bye"} \downarrow_{\#} \text{"all"}$$

is under-specified since both arguments have the same length. In order to reason about $\downarrow_f$ for arbitrary f, it is assumed only that $\downarrow_f$ is associative, idempotent, commutative, selective, and minimising in the sense that

$$f\,(x \downarrow_f y) = f\,x \downarrow f\,y$$

Here, $\downarrow$ is used as an abbreviation for $\downarrow_{id}$, where id is the identity function, and so selects the smaller of its two numeric arguments.

The under-definedness of $\downarrow_f$ can be very useful in specifying problems in computation. If necessary, "ties" can be resolved by imposing extra conditions on f. By definition, a *refinement* of f is a function g that respects the

ordering on values given by f, in the sense that

$$f\,x < f\,y \Rightarrow g\,x < g\,y$$

but g may introduce further distinctions. (Here, $\Rightarrow$ denotes logical implication), Such refinements can always be introduced into a specification in order to ease the task of calculation, the only requirement being that the refinement must be recorded and must be consistent with all previous refinements, if any. (In both [1] and [6], refinements were recorded by using a special symbol, either $\rightsquigarrow$ or $\sqsubseteq$. In the present paper, use of such signs is avoided; instead we stay strictly in the framework of equational reasoning, and make refinements explicit.)

Conditionals. We shall use the McCarthy conditional form $(p \rightarrow f, g)$ to describe the function

$$\begin{aligned}(p \rightarrow f, g)\,x &= f\,x, \quad \textbf{if } p\,x \\ &= g\,x, \quad \textbf{otherwise}\end{aligned}$$

For total predicates p we have the following well-known identity:

$$h \cdot (p \rightarrow f, g) = (p \rightarrow h \cdot f, h \cdot g) \tag{1}$$

Equation (1) is referred to as the "dot-cond" law.

Map. The operator $*$ (pronounced "map") takes a function on its left and a list (or set) on its right. Informally, we have

$$f * [a_1, a_2, \ldots, a_n] = [f\,a_1, f\,a_2, \ldots, f\,a_n]$$

with an analogous equation holding for sets. We can specify $*$ over lists by the three equations

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

These equations can also be expressed as identities between functions:

$$f * \cdot [\,]^\circ = [\,]^\circ \tag{2}$$

$$f * \cdot [\cdot] = [\cdot] \cdot f \tag{3}$$

$$f * \cdot +\!\!+/ = +\!\!+/ \cdot (f*) * \tag{4}$$

We will refer to (2) as the "map-empty" law, to (3) as the "map-single" law, and to (4) as the "map-concat" law. Equation (4) makes use of the reduction operator / discussed below. Similar equations hold for the definition of $*$ over sets. In particular, the analogue of (4) (in which $+\!\!+$ is replaced by $\cup$) is called the "map-union" law.

[*Note on syntax.* In [1] laws like the above were written with more brackets. For example, the map-concat law—called "map promotion" in [1]—was written

$$(f*)\cdot(+\!\!+/) \;=\; (+\!\!+/)\cdot((f*)*)$$

In the present paper we avoid these additional brackets by assuming functional composition ($\cdot$) has lowest precedence.]

Another useful identity is provided by the fact that $*$ distributes over functional composition—the "dot-map" law:

$$(f\cdot g)* \;=\; f*\cdot g* \tag{5}$$

Reduce. The reduction operator / takes a binary operator $\oplus$ on its left, and a list (or set) on its right. Informally, we have

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

More formally, we can specify $\oplus/$ on non-empty lists by the equations

$$\begin{array}{lcl} \oplus/[a] & = & a \\ \oplus/(x +\!\!+ y) & = & (\oplus/x)\oplus(\oplus/y) \end{array}$$

For the second equation to be unambiguous we require that $\oplus$ be an associative operator (because $+\!\!+$ is). These equations can also be expressed as functional identities (the "reduce-single" and "reduce-concat" laws):

$$\oplus/\cdot[\cdot] \;=\; id \tag{6}$$

$$\oplus/\cdot+\!\!+/ \;=\; \oplus/\cdot\oplus/* \tag{7}$$

Similarly, we can define $\oplus/$ over non-empty sets by the equations

$$\begin{array}{lcl} \oplus/\{a\} & = & a \\ \oplus/(x\cup y) & = & (\oplus/x)\oplus(\oplus/y) \end{array}$$

In particular, the function level equivalent of the second equation is the "reduce-union" law:

$$\oplus/ \cdot \cup/ \;=\; \oplus/ \cdot \oplus/ * \tag{8}$$

In order for the application of $\oplus/$ to a set to be unambiguous, we require $\oplus$ to be an associative, commutative and idempotent operator (because $\cup$ is). For example,

$$\oplus/x = \oplus/(x \cup x) = (\oplus/x) \oplus (\oplus/x)$$

and so $\oplus$ must be idempotent. Since $\downarrow_f$ has these three properties, we can reduce with it over both sets and lists. In particular, we have the "shortest-map" law:

$$\downarrow_\# / \cdot (f*)* = f* \cdot \downarrow_\# / \tag{9}$$

Identity elements. If $\oplus$ has an identity element e, then we can also define

$$\oplus/[\,] = \oplus/\{\,\} = e$$

Equivalently, the "reduce-empty" laws says

$$\oplus/ \cdot [\,]^\circ = e^\circ \tag{10}$$

It is often useful to invent "fictitious" identity elements for operators that do not possess them. For example, we introduce ω as the fictitious identity element of $\downarrow_\#$: thus

$$\downarrow_\# /\{\,\} = \omega$$

Provided certain care is taken, fictitious identity elements can always be adjoined to a domain (see [1] for a more complete discussion).

Homomorphisms. Functions of the form $\oplus/ \cdot f*$ describe homomorphisms over lists (or sets), and are discussed in [1] (see also [3]). There are three particular homomorphisms that will be needed below: generalised conjunction, filter, and cartesian product.

(i) *Generalised conjunction.* For a boolean-valued function p we define *all* p by the equation

$$\mathit{all}\ p = \wedge/ \cdot p*$$

where $\wedge$ denotes logical conjunction. Thus $(all\ p\ x)$ returns true if every element of the list (or set) x satisfies p, and false otherwise. One simple law ("all-and") is

$$all\,(p \wedge q) \quad = \quad all\ p \wedge all\ q \tag{11}$$

(ii) *Filter*. The operator $\lhd$ (pronounced "filter") is defined for sets by the equation

$$p \lhd = \cup/ \cdot (p \rightarrow \{\cdot\}, \{\,\}^\circ)*$$

Thus, $(p \lhd)$ is a homomorphism on sets. A similar equation holds for lists. In effect, $(p \lhd x)$ returns the subset of elements of x that satisfy p. This subset is obtained by replacing each element a of x by $\{a\}$ if $p\ a$ holds, or $\{\,\}$ if $p\ a$ does not hold, and taking the union of the resulting set of sets.

The "filter-union" law says that

$$p \lhd \cdot \cup/ \quad = \quad \cup/ \cdot (p \lhd)* \tag{12}$$

The following proof of this result shows the way we shall lay out the steps of a calculation:

$$\begin{array}{ll} & p \lhd \cdot \cup/ \\ = & \text{definition of } \lhd \\ & \cup/ \cdot (p \rightarrow \{\cdot\}, \{\,\}^\circ)* \cdot \cup/ \\ = & \text{map-union (4)} \\ & \cup/ \cdot \cup/ \cdot ((p \rightarrow \{\cdot\}, \{\,\}^\circ)*)* \\ = & \text{reduce-union (8)} \\ & \cup/ \cdot \cup/ * \cdot ((p \rightarrow \{\cdot\}, \{\,\}^\circ)*)* \\ = & \text{dot-map (5)} \\ & \cup/ \cdot (\cup/ \cdot ((p \rightarrow \{\cdot\}, \{\,\}^\circ)*)* \\ = & \text{definition of } \lhd \\ & \cup/ \cdot (p \lhd)* \end{array}$$

as required. $\square$

For total predicates p and q, we have the "and-filter" law:

$$(p \wedge q) \lhd \quad = \quad p \lhd \cdot q \lhd \tag{13}$$

Another identity ("reduce-cond") involving $\lhd$ is as follows. Suppose $\oplus$ has identity element e, then

$$\oplus/ \cdot (p \rightarrow f, e^\circ)* \quad = \quad \oplus/ \cdot f* \cdot p \lhd \tag{14}$$

Here is the proof:

$$
\begin{array}{ll}
 & \oplus/ \cdot f* \cdot p \triangleleft \\
= & \text{definition of } \triangleleft \\
 & \oplus/ \cdot f* \cdot +\!\!+/ \cdot (p \rightarrow [\cdot], [\,]^\circ)* \\
= & \text{map-concat (4)} \\
 & \oplus/ \cdot +\!\!+/ \cdot (f*)* \cdot (p \rightarrow [\cdot], [\,]^\circ)* \\
= & \text{reduce-concat (7)} \\
 & \oplus/ \cdot \oplus/* \cdot (f*)* \cdot (p \rightarrow [\cdot], [\,]^\circ)* \\
= & \text{dot-map (5)} \\
 & \oplus/ \cdot (\oplus/ \cdot (f*) \cdot (p \rightarrow [\cdot], [\,]^\circ))* \\
= & \text{dot-cond (1)} \\
 & \oplus/ \cdot (p \rightarrow \oplus/ \cdot f* \cdot [\cdot], \oplus/ \cdot f* \cdot [\,]^\circ)*
\end{array}
$$

Now we argue that

$$
\begin{array}{ll}
 & \oplus/ \cdot f* \cdot [\cdot] \\
= & \text{map-single (3)} \\
 & \oplus/ \cdot [\cdot] \cdot f \\
= & \text{reduce-single (6)} \\
 & f
\end{array}
$$

and also that

$$
\begin{array}{ll}
 & \oplus/ \cdot f* \cdot [\,]^\circ \\
= & \text{map-empty (2)} \\
 & \oplus/ \cdot [\,]^\circ \\
= & \text{reduce-empty (10)} \\
 & e^\circ
\end{array}
$$

completing the calculation. □

(iii) *Cartesian product.* The third homomorphism is a function cp (short for "cartesian product") with type

$$cp :: [\{\alpha\}] \rightarrow \{[\alpha]\}$$

Thus, the cartesian product of a list of sets is a set of lists. Informally we have

$$cp\ [S_1, S_2, \ldots, S_n] = \{[a_1, a_2, \ldots, a_n] \mid a_j \in S_j\}$$

Formally we can define cp as the homomorphism

$$cp = +\!\!+^\circ/ \cdot ([\cdot]*)*$$

$$S +\!\!+^\circ T = \{x +\!\!+ y \mid x \in S;\ y \in T\}$$

Note that $+\!\!+^\circ$ is an associative operator. For example, we can calculate

$$\begin{array}{ll} & cp\,[\{a,b\},\{c\},\{d,e\}] \\ = & +\!\!+^\circ/[\{[a],[b]\},\{[c]\},\{[d],[e]\}] \\ = & \{[a],[b]\} +\!\!+^\circ \{[c]\} +\!\!+^\circ \{[d],[e]\} \\ = & \{[a,c,d],[a,c,e],[b,c,d],[b,c,d]\} \end{array}$$

Two identities involving cp are the "cp-single" and "cp-cond" laws:

$$cp \cdot \{\cdot\}* = \{\cdot\} \tag{15}$$

$$cp \cdot (p \rightarrow f, \{\,\}^\circ)* = (all\ p \rightarrow cp \cdot f*, \{\,\}^\circ) \tag{16}$$

In effect, the first law says that the cartesian product of a list of singleton sets is a singleton set, while the second law says—in part—that if any set in the argument list is empty, then so is the cartesian product.

Generalised inverse. The generalised inverse of a function f is defined by

$$f^{-1}a = \{b \mid f\ b = a\}$$

The "dot-inverse" and "map-inverse" laws are

$$(f \cdot g)^{-1} = \cup/ \cdot g^{-1} * \cdot f^{-1} \tag{17}$$

$$(f*)^{-1} = cp \cdot f^{-1} * \tag{18}$$

We give a proof of (17):

$$\begin{array}{ll} & (f \cdot g)^{-1} a \\ = & \text{definition of inverse} \\ & \{b \mid f\ (g\ b) = a\} \\ = & \text{set theory} \\ & \cup/\{\{b \mid g\ b = c\} \mid f\ c = a\} \\ = & \text{definition of inverse} \\ & \cup/\{g^{-1}c \mid f\ c = a\} \\ = & \text{definition of } * \\ & \cup/g^{-1} * \{c \mid f\ c = a\} \\ = & \text{definition of inverse} \\ & \cup/g^{-1} * f^{-1}a \end{array}$$

as required. □

Partitions. Finally, we introduce a special case of generalised inverse. By definition, a *partition* of a list x is a decomposition of x into contiguous segments. A *proper* partition is a decomposition into non-empty segments. The (infinite) set of partitions of x is just $(+\!\!+/)^{-1}x$. The function *parts*, where

$$parts = all\ (\neq [\,]) \cdot (+\!\!+/)^{-1}$$

returns the (finite) set of proper partitions of a sequence. Two theorems about *parts* are given in Section 5.

4. Calculation of *code*.

Using the identities given in the previous section, we can now begin to calculate an algorithm for *code*. The derivation is almost entirely mechanical: at each step—with one or two minor exceptions—there is only one law that can be applied.

$$\begin{array}{ll} & code \\ = & \text{definition of code} \\ & \downarrow_{\#}/ \cdot\ decode^{-1} \\ = & \text{definition of decode} \\ & \downarrow_{\#}/ \cdot (+\!\!+/ \cdot exp*)^{-1} \\ = & \text{dot-inverse (17)} \\ & \downarrow_{\#}/ \cdot \cup/ \cdot (exp*)^{-1} * \cdot (+\!\!+/)^{-1} \\ = & \text{reduce-union (8)} \\ & \downarrow_{\#}/ \cdot \downarrow_{\#}/ * \cdot (exp*)^{-1} * \cdot (+\!\!+/)^{-1} \\ = & \text{dot-map (5)} \\ & \downarrow_{\#}/ \cdot (\downarrow_{\#}/ \cdot (exp*)^{-1}) * \cdot (+\!\!+/)^{-1} \\ = & \text{introduction of } f \\ & \downarrow_{\#}/ \cdot f * \cdot (+\!\!+/)^{-1} \end{array}$$

where

$$f = \downarrow_{\#}/ \cdot (exp*)^{-1}$$

The purpose of this last step is just to name the subexpression that will be the focus of future manipulation.

Before calculating f, we first consider the function exp^{-1} that will arise during the calculation. This function has values given by

$$exp^{-1}x = \{(a, n) \mid exp\,(a, n) = x\}$$

For the set on the right to be non-empty, we require that x is a non-empty list of duplicated values (Recall that n must be a positive integer). Suppose

we define

$$\begin{array}{lcl} nedup\ x & = & (x \neq [\,]) \wedge dup\ x \\ dup\ x & = & all\ (=head\ x)\ x \\ rep\ x & = & (head\ x, \#x) \end{array}$$

where $head\,x$ returns the first element of the non-empty list x. It then follows that

$$exp^{-1}x = (nedup\ x \rightarrow \{rep\ x\}, \{\,\})$$

or, expressed at the function level, that

$$exp^{-1} = (nedup \rightarrow \{\cdot\} \cdot rep, \{\,\}^\circ)$$

This equation is needed below.

Now we return to calculating f:

$$\begin{array}{ll} & f \\ = & \text{definition of } f \\ & \downarrow_\# / \cdot (exp*)^{-1} \\ = & \text{map-inverse (18)} \\ & \downarrow_\# / \cdot\ cp \cdot exp^{-1}* \\ = & \text{definition of } exp^{-1} \\ & \downarrow_\# / \cdot\ cp \cdot (nedup \rightarrow \{\cdot\} \cdot rep, \{\,\}^\circ)* \\ = & \text{cp-cond (16)} \\ & \downarrow_\# / \cdot (all\ nedup \rightarrow cp \cdot (\{\cdot\} \cdot rep)*, \{\,\}^\circ) \\ = & \text{dot-cond (1)} \\ & (all\ nedup \rightarrow \downarrow_\# / \cdot\ cp \cdot (\{\cdot\} \cdot rep)*, \downarrow_\# / \cdot\ \{\,\}^\circ) \\ = & \text{dot-map (5), and introducing } \omega \text{ as the identity of } \downarrow_\# \\ & (all\ nedup \rightarrow \downarrow_\# / \cdot\ cp \cdot \{\cdot\} * \cdot rep*, \omega^\circ) \\ = & \text{cp-single (15)} \\ & (all\ nedup \rightarrow \downarrow_\# / \cdot \{\cdot\} \cdot rep*, \omega^\circ) \\ = & \text{reduce-single (6)} \\ & (all\ nedup \rightarrow rep*, \omega^\circ) \end{array}$$

Having calculated f, we continue with the calculation of *code*:

$$
\begin{array}{ll}
 & \mathit{code} \\
= & \text{calculation so far} \\
 & \downarrow_{\#} / \cdot f * \cdot (+\!\!+/)^{-1} \\
= & \text{calculation of } f \\
 & \downarrow_{\#} / \cdot (\mathit{all}\ \mathit{nedup} \rightarrow \mathit{rep}*, \omega^{\circ}) * \cdot (+\!\!+/)^{-1} \\
= & \text{reduce-cond (14)} \\
 & \downarrow_{\#} / \cdot (\mathit{rep}*) * \cdot \mathit{all}\ \mathit{nedup} \triangleleft \cdot (+\!\!+/)^{-1} \\
= & \text{shortest-map (9)} \\
 & \mathit{rep} * \cdot \downarrow_{\#} / \cdot \mathit{all}\ \mathit{nedup} \triangleleft \cdot (+\!\!+/)^{-1} \\
= & \text{definition of } \mathit{nedup}\text{; all-and (11); and-filter (13)} \\
 & \mathit{rep} * \cdot \downarrow_{\#} / \cdot \mathit{all}\ \mathit{dup} \triangleleft \cdot \mathit{all}\ (\neq [\,]) \triangleleft \cdot (+\!\!+/)^{-1} \\
= & \text{definition of } \mathit{parts} \\
 & \mathit{rep} * \cdot \downarrow_{\#} / \cdot \mathit{all}\ \mathit{dup} \cdot \mathit{parts}
\end{array}
$$

We have shown that

$$\mathit{code}\ x = \mathit{rep} * \downarrow_{\#} / \mathit{all}\ \mathit{dup} \triangleleft \mathit{parts}\ x$$

In words, *code* x can be obtained by taking the shortest partition of x into non-empty segments of duplicated values, and representing each segment by its common value and its length. Expressed this way, the derived equation seems entirely reasonable, and might even have served as the specification of the problem. Unlike the original specification, the new definition of *code* is executable. The set *parts* x contains a finite number of elements (in fact, 2^{n-1} elements, if $n > 0$ is the length of x), and these can be enumerated, filtered, and the shortest taken. What we now need is a faster method for computing expressions of the above form, and for this we need to consider various algorithms for computing partititons.

5. Algorithms for partititons.

Expressions of the form

$$\downarrow_{f} / \mathit{all}\ p \triangleleft \mathit{parts}\ x$$

arise in a number of applications. For example, in sorting by natural-merging, the input is first divided into runs of non-decreasing values. The function *runs* which does this division can be expressed in the form

$$\mathit{runs}\ x = \downarrow_{\#} / \mathit{all}\ \mathit{nondec} \triangleleft \mathit{parts}\ x$$

This reads: the shortest partititon of x, all of whose components are non-decreasing sequences.

Similarly, the problem of filling a paragraph (see [2]) can be expressed as a problem about partitions:

$$\mathit{fill}\ m\ ws = \downarrow_W / \mathit{all}\ (\mathit{fits}\ m) \triangleleft \mathit{parts}\ ws$$

This reads: the least wasteful (according to the waste function W) partition of a sequence of words ws, all of whose component segments (or "lines") will fit on a line of given width m.

There are numerous other examples (including, of course, *code*). This leads to the question: can we find ways of computing solutions to problems of the form

$$\downarrow_f / \mathit{all}\ p \triangleleft \mathit{parts}\ x$$

efficiently?

The Greedy algorithm. Here is a "greedy" algorithm for computing a partititon:

$$\begin{array}{lcll} \mathit{greedy}\ p\ x & = & [\,], & \textbf{if } x = [\,] \\ & = & [y] +\!\!+ \mathit{greedy}\ p\ (x \rightharpoonup y), & \textbf{otherwise} \\ & & \textbf{where } y = \uparrow_{\#} / p \triangleleft \mathit{inits}^{+} x & \end{array}$$

In this algorithm, the expression $\mathit{inits}^{+} x$ denotes the list of non-empty initial segments of x, in increasing order of length, and $(x \rightharpoonup y)$ is the sequence that remains when the initial segment y of x is deleted from x. The operator $\rightharpoonup$ is specified by the equation

$$(u +\!\!+ v) \rightharpoonup u = v$$

for all sequences u and v.

It is easy to see that, at each step, the greedy algorithm chooses the longest non-empty initial segment y of the remaining input x that satisfies the predicate p. In order for *greedy* to make progress, it is necessary to suppose that p holds for $[\,]$ and every singleton sequence at least. In this way a non-empty portion of the input is "consumed" at each step.

There is a useful condition on p which enables the next segment y to be computed efficiently. Say p is *prefix-closed* if

$$p\ (x +\!\!+ y) \Rightarrow p\ x$$

In words, if p holds for a sequence, then it holds for every initial segment of the sequence (including $[\,]$). If p is prefix-closed, then there exists a *derivative* function δ such that

$$p\ (x +\!\!+ [a]) = p\ x \wedge \delta\ a\ x$$

For example, with $p = dup$ we have

$$\delta\ a\ x = (x = [\,]) \vee (a = head\ x)$$

and with $p = nondec$ we have

$$\delta\ a\ x = (x = [\,]) \vee (last\ x \leq a)$$

where $last\ x$ returns the last element of a non-empty sequence x.

If p is prefix-closed, we can formulate the following version of the greedy algorithm:

$$\begin{array}{lcll} greedy\ p\ x & = & shunt\ [\,]\ x & \\ shunt\ y\ [\,] & = & [\,] & \\ shunt\ y\ ([a] +\!\!+ x) & = & shunt\ (y +\!\!+ [a])\ x, & \textbf{if } \delta\ a\ y \\ & = & [y] +\!\!+ shunt\ [a]\ x, & \textbf{otherwise} \end{array}$$

This version is linear in the number of δ-calculations.

A related condition on p is that of being *suffix-closed*, i.e.

$$p\ (x +\!\!+ y) \Rightarrow p\ y$$

If p is both prefix- and suffix-closed, then p is said to be *segment-closed*. For example, both *dup* and *nondup* are segment-closed. The suffix-closed condition arises in one of the theorems discussed below.

The Leery algorithm. Here is another algorithm for computing partitions:

$$\begin{array}{l} leery\ f\ p\ x \\ \quad = [\,], \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } x = [\,] \\ \quad = \downarrow_f / \{[y] +\!\!+ leery\ f\ p\ (x - y) \mid y \in p \lhd inits^+ x\} \quad \textbf{otherwise} \end{array}$$

This algorithm is a little more "careful" than the greedy algorithm (whence the name "leery"). At each stage, some initial segment y of x is chosen so that (i) y satisfies p; and (ii) $f\ ([y] +\!\!+ ys)$ is as small as possible, where ys is the result of adopting the same strategy on the rest of the input. Unlike the

greedy algorithm, the leery algorithm will not necessarily choose the longest initial segment of x satisfying p at each stage.

The direct recursive implementation of *leery* is inefficient, since values of *leery* on tail segments of x will be recomputed many times. Instead, *leery* is better implemented by a dynamic-programming scheme that computes and stores the results of applying *leery* to all tail segments of the input. In making a decision about the next component of the partition, these subsidiary results are then available without recomputation. We shall not, however, formulate the efficient version of *leery* here.

We now state two theorems about the greedy and leery algorithms. For the first, we need the following definition.

Definition 1 *A function f, from sequences to numbers, is said to be* prefix-stable *if*

$$f\ x \leq f\ y \Leftrightarrow f\ ([a] +\!\!+ x) \leq f\ ([a] +\!\!+ y)$$

for all a, x, and y. Equivalently, f is prefix-stable if

$$\downarrow_f / \cdot ([a]+\!\!+)* = ([a]+\!\!+) \cdot \downarrow_f /$$

for all a.

Theorem 1 (The Leery **Theorem)** *If f is prefix-stable, then*

$$\downarrow_f / all\ p \lhd parts\ x = leery\ f\ p\ x,$$

provided p holds for $[\,]$ and all singleton sequences.

Proof. We shall need the following recursive definition of *parts*:

$$\begin{array}{lcll} parts\ [\,] & = & \{[\,]\} & \\ parts\ x & = & \cup / subparts * inits^+\ x & \textbf{if } x \neq [\,] \\ & & \textbf{where } subparts\ y = ([y]+\!\!+) * parts\ (x \rightarrowtail y) & \end{array}$$

We omit the proof that this definition of *parts* satisfies the specification

$$parts = all\ (\neq [\,]) \cdot (+\!\!+/)^{-1}$$

Let θ be defined by

$$\theta x = \downarrow_f / all\ p \lhd parts\ x$$

The theorem is proved by showing that θ satisfies the recursive definition of *leery*. There are two cases to consider:

Case $x = [\,]$. The derivation of

$$\theta[\,] = [\,]$$

is straightforward and is omitted.

Case $x \neq [\,]$. In the case $x \neq [\,]$, we argue

$$\begin{array}{ll} & \downarrow_f / \mathit{all}\ p \triangleleft \mathit{parts}\ x \\ = & \text{definition of } \mathit{parts} \\ & \downarrow_f / \mathit{all}\ p \triangleleft \cup / \mathit{subparts} * \mathit{inits}^+ x \\ = & \text{filter-union (12); reduce-union (8)} \\ & \downarrow_f / \downarrow_f / * (\mathit{all}\ p \triangleleft) * \mathit{subparts} * \mathit{inits}^+ x \\ = & \text{dot-map (5)} \\ & \downarrow_f / (\downarrow_f / \cdot\ \mathit{all}\ p \triangleleft \cdot \mathit{subparts}) * \mathit{inits}^+ x \\ = & \text{introduction of } h \\ & \downarrow_f / h * \mathit{inits}^+ x \end{array}$$

where

$$h\ y = \downarrow_f / \mathit{all}\ p \triangleleft \mathit{subparts}\ y$$

We continue by calculating h (note that h depends on x):

$$\begin{array}{ll} & \downarrow_f / \mathit{all}\ p \triangleleft \mathit{subparts}\ y \\ = & \text{definition of } \mathit{subparts} \\ & \downarrow_f / \mathit{all}\ p \triangleleft ([y] +\!\!+) * \mathit{parts}\ (x \rightarrowtail y) \\ = & \text{property of } \mathit{all} \\ & \downarrow_f / (p\ y \rightarrow ([y] +\!\!+) * \mathit{all}\ p \triangleleft \mathit{parts}\ (x \rightarrowtail y), \{\,\}) \\ = & \text{dot-cond (1)} \\ & (p\ y \rightarrow \downarrow_f / ([y] +\!\!+) * \mathit{all}\ p \triangleleft \mathit{parts}\ (x \rightarrowtail y), \downarrow_f / \{\,\}) \\ = & \text{prefix-stability} \\ & (p\ y \rightarrow [y] +\!\!+ \downarrow_f / \mathit{all}\ p \triangleleft \mathit{parts}\ (x \rightarrowtail y), \downarrow_f / \{\,\}) \\ = & \text{definition of } \theta \\ & (p\ y \rightarrow [y] +\!\!+ \theta(x \rightarrowtail y), \downarrow_f / \{\,\}) \end{array}$$

Finally, if we substitute the derived definition of h into the derived equation for θ and apply the reduce-cond law, we obtain

$$\begin{array}{lll} \theta x & = & \downarrow_f / \phi * p \triangleleft \mathit{inits}^+ x \\ & & \textbf{where}\ \ \phi y = [y] +\!\!+ \theta(x \rightarrowtail y) \end{array}$$

In other words, θ satisfies the recursive definition of leery, completing the proof of the theorem. □

Perhaps the simplest useful example of a prefix-stable function is the length function $\#$. Another example is the function $+/$ that sums a list of numbers. However, prefix-stability is too strong a condition to be satsfied by commonly useful optimisation functions. Fortunately there is a weaker condition that can be artificially strengthened to give prefix-stability.

Definition 2 *A function f, from sequences to numbers, is said to be* weakly prefix-stable *if*

$$f\ x \leq f\ y \Rightarrow f\ ([a] +\!\!+ x) \leq f\ ([a] +\!\!+ y)$$

for all x, y and a.

For example, the function $(\uparrow /)$ (where $\uparrow$ selects the greater of its two numeric arguments) is weakly prefix-stable, but not prefix-stable.

The proof of the following lemma, due to A.W. Roscoe, is given in an Appendix.

Lemma 1 *Suppose $f :: [\alpha] \rightarrow Q$ is weakly prefix-stable, where α contains a countable number of elements, and Q denotes the rationals. Then there exists a prefix-stable function $g :: [\alpha] \rightarrow Q$ that refines f.*

By using Lemma 1, we can therefore apply the Leery theorem in the case of a weakly prefix-stable function f.

For the second theorem we need the following definition.

Definition 3 *A function f, from sequences (of seqences) to numbers, is* greedy *if both the following conditions hold for all x, y, z and s:*

$$f\ ([x +\!\!+ y] +\!\!+ s) < f\ ([x] +\!\!+ [y] +\!\!+ s)$$
$$f\ ([x +\!\!+ y] +\!\!+ [z] +\!\!+ s) < f\ ([x] +\!\!+ [y +\!\!+ z] +\!\!+ s)$$

The first greedy condition says that shorter partitions have a smaller f-value than longer ones; the second condition says that, for partitions of equal length, the longer the first component, tbe smaller is its f-value.

Theorem 2 (The Greedy Theorem) *If f is prefix-stable and greedy, and p is suffix-closed, then*

$$\downarrow_f / all\ p \lhd parts\ x = greedy\ p\ x$$

provided p holds for all singleton sequences.

Proof. We show *leery f p* = *greedy p*. Abbreviating *leery f p* by θ, it suffices to prove that

$$f\,([x +\!\!+ y] +\!\!+ \theta z) \leq f\,([x] +\!\!+ \theta(y +\!\!+ z))$$

for all x, y and z, with equality only in the case $y = [\,]$. In words, the longer the next component of the partition, the smaller is its f-value.

The proof is by induction on the length of y. The base case, $\#y = 0$, is immediate. For the induction step, consider the value of $\theta(y +\!\!+ z)$. There are two possibilities: either

$$\theta(y +\!\!+ z) \;=\; [y +\!\!+ z_1] +\!\!+ \theta(z_2) \qquad \text{(case A)}$$

for some $z_1 \neq [\,]$ and z_2, such that $z = z_1 +\!\!+ z_2$ and $p\,(y +\!\!+ z_1)$ holds; or

$$\theta(y +\!\!+ z) \;=\; [y_1] +\!\!+ \theta(y_2 +\!\!+ z) \qquad \text{(case B)}$$

for some $y_1 \neq [\,]$ and y_2, such that $y = y_1 +\!\!+ y_2$ and $p\ y_1$ holds; (This case includes the possibility that $y_2 = [\,]$.)

In case A, we reason that $p\ z_1$ holds, since p is suffix-closed, and so by definition of θ,

$$f\,(\theta z) \leq f\,([z_1] +\!\!+ \theta\, z_2)$$

Hence

$$\begin{array}{ll} & f\,([x +\!\!+ y] +\!\!+ \theta z) \\ \leq & \text{prefix-stability} \\ & f\,([x +\!\!+ y] +\!\!+ [z_1] +\!\!+ \theta\, z_2) \\ < & \text{second greedy condition} \\ & f\,([x] +\!\!+ [y +\!\!+ z_1] +\!\!+ \theta\, z_2) \\ = & \text{case A} \\ & f\,([x] +\!\!+ \theta(y +\!\!+ z)) \end{array}$$

as required.

In case B, we reason

$$\begin{array}{ll} & f\,([x +\!\!+ y] +\!\!+ \theta z) \\ = & \text{case B:} \\ & f\,([x +\!\!+ y_1 +\!\!+ y_2] +\!\!+ \theta z) \\ \leq & \text{induction hypothesis, as } \#y_2 < \#y \\ & f\,([x +\!\!+ y_1] +\!\!+ \theta(y_2 +\!\!+ z)) \\ < & \text{first greedy condition} \\ & f\,([x] +\!\!+ [y_1] +\!\!+ \theta(y_2 +\!\!+ z)) \\ = & \text{case B} \\ & f\,([x] +\!\!+ \theta(y +\!\!+ z)) \end{array}$$

The length function $\#$ satisfies the first greedy condition, but not the second. As with prefix-stability, there is a weaker version of the greedy condition that is sufficient to enable the Greedy theorem to be used.

Definition 4 *Say f is* weakly greedy *if the greedy conditions hold when $<$ is replaced by $\leq$.*

Lemma 2 *A weakly greedy function has a greedy refinement.*

Proof. Let f be weakly greedy. Define g by the rule

$$g\ xs < g\ ys \qquad \text{if } (f\ xs < f\ ys) \vee (f\ xs = f\ ys \wedge \#head\ xs > \#head\ ys)$$

It is clear that g respects the ordering of f and is greedy. □

6. Application.

We give one application of the Greedy theorem. Since *dup* is suffix-closed (in fact, segment-closed), and $\#$ is weakly greedy, we have that

$$code = rep * \cdot greedy\ dup$$

is a solution to the problem of run-length encoding.

Acknowledgements Much of the calculus presented above was developed in collaboration with Lambert Meertens of the CWI, Amsterdam, to whom I am grateful for both support and friendship. I would like to thank Bill Roscoe, of the PRG, Oxford, who supplied the proof of Lemma 1 at a moment's notice. I am also grateful to Carroll Morgan and Tony Hoare for many enjoyable discussions on the proper role of refinement in a purely functional calculus.

References

1. Bird, R.S. An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design*, (edited by M. Broy), Springer-Verlag, (1987) 3-42.

2. Bird, R.S. Transformational programming and the paragraph problem. *Science of Computer Programming* 6 (1986) 159-189.

3. Bird, R.S. and Hughes, R.J.M. The alpha-beta algorithm: an exercise in program transformation. *Inf. Proc. Letters* 24 (1987) 53-57.

4. Bird, R.S. and Meertens L.G.L.T Two exercises found in a book on algorithmics. *Program Specification and Transformation* (edited by L.G.L.T Meertens), North-Holland, (1987), 451-458.

5. Bird, R.S. and Wadler, P. *An Introduction to Functional Programming* Prentice-Hall (to be published Spring-1988).

6. Meertens, L.G.L.T Algorithmics - towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science*, CWI Monographs, North-Holland, 1 (1986) 289-334.

Appendix: Proof of Lemma 1

The proof of Lemma 1 is by constructing an injective weakly prefix-stable function g. Such a function must also be prefix-stable. We argue this by contradiction. Suppose the implication

$$g\ ([a] +\!\!+\ x) \leq g\ ([a] +\!\!+\ y) \Rightarrow g\ x \leq g\ y$$

fails to hold. In such a case

$$g\ ([a] +\!\!+\ x) \leq g\ ([a] +\!\!+\ y) \wedge g\ x > g\ y$$

for some a, x and y. By assumption, g is weakly prefix-stable, so the above assertion implies

$$g\ ([a] +\!\!+\ x) \leq g\ ([a] +\!\!+\ y) \wedge g\ ([a] +\!\!+\ y) \leq g\ ([a] +\!\!+\ x)$$

However, since g is injective, it follows that $x = y$ which contradicts the hypothesis.

To construct g, we suppose the finite sequences of $[\alpha]$ are enumerated in such a way that x precedes $[a] +\!\!+\ x$ for all a and x. Let this enumeration be denoted by $\{x_0, x_1, \ldots\}$. Furthermore, to avoid a subsequent appea to the Axiom of Choice, let $\{q_0, q_1, \ldots\}$ be some fixed enumeration of the (positive and negative) rationals Q.

For the base case in the inductive construction of g, arbitrarily define $g\ x_0 = 0$. Note that $x_0 = [\,]$. Assume, by way of induction, that

$$g\ x_0, g\ x_1, \ldots, g\ x_{i-1}$$

have so far been defined and that these distinct values satisfy all that is required of g. To determine $g\ x_i$, we partition $\{x_j \mid j < i\}$ into three sets, any of which may be empty:

$$\begin{aligned} A &= \{x_j \mid j < i \wedge f\ x_j < f\ x_i\} \\ B &= \{x_j \mid j < i \wedge f\ x_j = f\ x_i\} \\ C &= \{x_j \mid j < i \wedge f\ x_j > f\ x_i\} \end{aligned}$$

Because g preserves the ordering of f, the rational values assigned to g for arguments in A must precede those assigned to g for arguments in B, which must in turn precede those for arguments in C.

Suppose $x_i = [a] +\!\!+ x_k$. By the given enumeration of sequences, we have $k < i$. Define the following two subsets of B:

$$\begin{aligned} B_1 &= \{[a] +\!\!+ x_l \mid g\ x_l < g\ x_k\} \\ B_2 &= \{[a] +\!\!+ x_l \mid g\ x_l > g\ x_k\} \end{aligned}$$

For $x \in B_1$ and $y \in B_2$ we have $g\ x < g\ y$, since—by induction—g is injective and weakly prefix-stable on the values already defined. We now choose $g\ x_i$ to be the rational q, of least index in the given enumeration of Q, such that q is greater than every element of $g * (A \cup B_1)$ and less than every element of $g * (C \cup B_2)$. By construction we have, for $j < i$, that

$$\begin{aligned} f\ x_j < f\ x_i &\Rightarrow g\ x_j < g\ x_i \\ f\ x_j > f\ x_i &\Rightarrow g\ x_j > g\ x_i \end{aligned}$$

so the extended definition of g respects the ordering of f. We must also show that the extension of g maintains weak prefix-stablility. Supposing $x_i = [a] +\!\!+ x_k$ and $x_j = [a] +\!\!+ x_l$, where $j < i$, we need to show

$$\begin{aligned} g\ x_l < g\ x_k &\Rightarrow g\ x_j < g\ x_i \\ g\ x_l > g\ x_k &\Rightarrow g\ x_j > g\ x_i \end{aligned}$$

Since f is weakly prefix-stable, and g respects f, we have

$$f\ x_i < f\ x_j \Rightarrow f\ x_k < f\ x_l \Rightarrow g\ x_k < g\ x_l$$

Thus,

$$\begin{aligned} g\ x_l < g\ x_k &\Rightarrow g\ x_l < g\ x_k \wedge f\ x_i \geq f\ x_j \\ &\Rightarrow g\ x_j < g\ x_i, \end{aligned}$$

by definition of $g\ x_i$. The proof of the second part is similar. This completes the induction, and the proof of the lemma. □