JFP **31**, e29, 9 pages, 2021.
doi:10.1017/S0956796821000198

FUNCTIONAL PEARLS

A greedy algorithm for dropping digits

RICHARD BIRD
Department of Computer Science, University of Oxford, Oxford OX1 3QD, UK
(*e-mail:* richard.bird@cs.ox.ac.uk)

SHIN-CHENG MU
Institute of Information Science, Academia Sinica, Taipei, Taiwan
(*e-mail:* scm@iis.sinica.edu.tw)

Abstract

Consider the following puzzle: given a number, remove k digits such that the resulting number is as large as possible. Various techniques are employed to derive a linear-time solution to the puzzle: we justify the structure of a greedy algorithm by predicate logic, give a constructive proof of the greedy condition using a dependently typed proof assistant and calculate the greedy step as well as the final, linear-time optimisation by equational reasoning.

1 Introduction

Greedy algorithms—algorithms that make locally optimal choices at each step while attempting to find globally optimal solutions (Cormen *et al.*, 2001, Ch. 16)—abound in computing. Well-known examples include Huffman coding, minimum cost spanning trees and the coin-changing problem, but there are many others. This pearl adds yet another problem to this collection. However, as has been said before, greedy algorithms can be tricky things. The trickiness is not in the algorithm itself, which is usually quite short and easy to understand, but in the proof that it does produce a best possible result. The mathematical theory of *matroids*, see Lawler (1976), and its generalisation to *greedoids*, see Korte *et al.* (1991), have been developed to explain why and when many greedy algorithms work, although the theory does not cover all possible cases. In practice, greedy algorithms are usually verified directly rather than by extracting the underlying matroid or greedoid. Curtis (2003) discusses four basic ways in which a greedy algorithm can be proved to work, one of which will be followed with our problem.

The problem is to remove k digits from a natural number containing at least k digits so that the result is as large as possible. For example, removing one digit from the number `"6782334"` gives `"782334"` as the largest possible result, while removing three digits yields `"8334"`. A number containing no digits is interpreted as 0. Given that a number can be seen as a list of digits, the problem can be generalised to removing, from a list whose elements are drawn from a linearly ordered type, k elements so that the result is largest under lexicographic ordering. While the problem was invented out of curiosity rather than for a pressing application, it has apparently been used as an interview question for candidates seeking jobs in computing (LeetCode, 2016). The hope is that we can discover an algorithm that takes linear time in the number of elements.

The first task is to give a formal specification of the problem. Throughout this pearl, we use notations similar to Haskell, with slight variations. For example, the type for lists is denoted by *List*, and we allow $(1 + k)$ as a pattern in function definitions, to match our inductive proofs. Function composition is denoted by $(\circ)$. Laziness is not needed. Consider the function *drops* that removes a single element from a non-empty list in all possible ways:

$$\begin{aligned}
&\mathit{drops} :: \mathit{List}\ a \rightarrow \mathit{List}\ (\mathit{List}\ a)\\
&\mathit{drops}\ [x] \quad\ = [[\,]]\\
&\mathit{drops}\ (x : \mathit{xs}) = \mathit{xs} : \mathit{map}\ (x{:})\ (\mathit{drops}\ \mathit{xs}) \quad .
\end{aligned}$$

For example, *drops* `"abcd"` = [`"bcd"`, `"acd"`, `"abd"`, `"abc"`]. The function *solve* for computing a solution can be defined by a simple exhaustive search:

$$\begin{aligned}
&\mathit{solve} :: \mathit{Ord}\ a \Rightarrow \mathit{Nat} \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a\\
&\mathit{solve}\ k = \mathit{maximum} \circ \mathit{apply}\ k\ \mathit{step} \circ \mathit{wrap} \quad ,\\
&\mathit{step} :: \mathit{List}\ (\mathit{List}\ a) \rightarrow \mathit{List}\ (\mathit{List}\ a)\\
&\mathit{step} = \mathit{concat} \circ \mathit{map}\ \mathit{drops} \quad .
\end{aligned}$$

The function *solve* converts a given input into a singleton list, applies the function *step* exactly k times to produce all possible candidates and computes the lexical maximum of the result. *Nat* is the type of natural numbers. The function *step* is *drops* lifted to a list of candidates. It computes, for each candidate, all the ways to drop 1 element. Functions *wrap* and *apply* are respectively defined by:

$$\begin{aligned}
&\mathit{wrap} :: a \rightarrow \mathit{List}\ a\\
&\mathit{wrap}\ x = [x] \quad ,
\end{aligned}
\qquad
\begin{aligned}
&\mathit{apply} :: \mathit{Nat} \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a\\
&\mathit{apply}\ 0 \qquad\ f = \mathit{id}\\
&\mathit{apply}\ (1 + k)\ f = \mathit{apply}\ k\ f \circ f \quad .
\end{aligned}$$

For brevity, for the rest of the pearl, we will write *apply k f* as f^k. Since a sequence of length n has n drops, and computing the larger of two lists of length $n - k$ takes $O\,(n - k)$ steps, this method for computing the answer takes $O\,(n^{k+1})$ steps. We aim to do better.

2 A greedy algorithm

To obtain a greedy algorithm, one would wish that the best way to remove k digits can be computed by removing 1 digit k times, and each time we greedily remove the digit that makes the current result as large as possible. That is, letting

$$\begin{aligned}
&\mathit{gstep} :: \mathit{Ord}\ a \Rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a\\
&\mathit{gstep} = \mathit{maximum} \circ \mathit{drops} \quad ,
\end{aligned}$$

we wish to have

$$\mathit{maximum} \circ \mathit{step}^k \circ \mathit{wrap} = \mathit{gstep}^k \quad . \tag{2.1}$$

One cannot just claim that this strategy works without proper reasoning, however. It can be shown that (2.1) is true if the following monotonicity condition holds (we denote

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press

lexicographic ordering on lists by $(\trianglelefteq)$, and ordering on individual elements by $(\leqslant)$):

$$xs \trianglelefteq ys \;\Rightarrow\; (\forall xs' \in drops\,xs \bullet (\exists ys' \in drops\,ys \bullet xs' \trianglelefteq ys')) \ , \tag{2.2}$$

where $(\in)$ is overloaded to denote membership for lists. That is, if ys is no worse than xs, whatever candidate we can obtain from xs, we can compute a candidate from ys that is no worse either.

Unfortunately, (2.2) does not hold for our problem. Considering $xs =$ `"1934"` $\lhd$ `"4234"` $= ys$, `"934"` is a possible result of $drops\,xs$, but the best we can do by removing one digit from ys is `"434"`. Note that (2.2) does not hold even if we restrict xs and ys to lists that can be obtained from the same source—certainly both `"1934"` and `"4234"` are both result of removing two digits from, say, `"194234"`.

In the terminology of Curtis (2003), the *Better-Global* principle—which says that if one first step is no worse than another, then there is a global solution using the former that is no worse than one using the latter—does not hold for this problem. What does hold is a weaker property, the *Best-Global* principle: a globally optimal solution can be obtained by starting out with the *best* possible step. Formally, what we do have is that for all xs and k:

$$(\forall xs' \in step^{1+k}\,[xs] \bullet (\exists zs \in (step^{k} \circ wrap \circ gstep)\,xs \bullet xs' \trianglelefteq zs)) \ . \tag{2.3}$$

That is, letting xs' be an arbitrary result of dropping $k+1$ elements from xs, one can always obtain a result zs that is no worse than xs' by greedily dropping the best element (by $gstep$) and then dropping k chosen elements.

Property (2.3), which we will refer to as the "greedy condition", will be proved in Section 4. For now, let us see how (2.3) helps to prove (2.1), that is, $maximum \circ step^{k} \circ wrap = gstep^{k}$. The proof proceeds by induction on k. For $k := 0$, both sides reduce to id. For the inductive case, we need the universal property of $maximum$: for all $s :: a \to b$ and $p :: a \to List\ b$, and for total order $(\trianglelefteq)$ on b:

$$s = maximum \circ p \;\equiv\; (\forall x \bullet s\,x \in p\,x) \wedge (\forall x, y \bullet y \in p\,x \Rightarrow y \trianglelefteq s\,x) \ .$$

To prove $maximum \circ step^{1+k} \circ wrap = gstep^{1+k}$, we need to show that 1. for all xs, $gstep^{1+k}\,xs$ is a member of $step^{1+k}\,[xs]$, which is a routine proof, and 2. for all xs and for all $ys \in step^{1+k}\,[xs]$, we have $ys \trianglelefteq gstep^{1+k}\,xs$. We reason below:

$$
\begin{array}{ll}
 & ys \trianglelefteq gstep^{1+k}\,xs \\
\equiv & \quad \{\ \text{since } f^{1+k} = f^{k} \circ f\ \} \\
 & ys \trianglelefteq gstep^{k}\,(gstep\,xs) \\
\equiv & \quad \{\ \text{induction hypothesis}\ \} \\
 & ys \trianglelefteq maximum\,(step^{k}\,[gstep\,xs]) \\
\equiv & \quad \{\ \text{since } ys \trianglelefteq maximum\,xss \;\equiv\; (\exists zs \in xss \bullet ys \trianglelefteq zs)\ \} \\
 & (\exists zs \in step^{k}\,[gstep\,xs] \bullet ys \trianglelefteq zs) \\
\Leftarrow & \quad \{\ \text{by (2.3)}\ \} \\
 & ys \in step^{1+k}\,[xs] \ ,
\end{array}
$$

which is our assumption. We have thus proved (2.1).

Remark: the proof above was carried out using predicate logic. There is a relational counterpart, in the style of Bird & de Moor (1997), that is slightly more concise and more

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press

general, but requires the use of additional technical machinery. In (2.3), we restrict our discussion to total orders to ensure that *maximum* returns one unique result. More general scenarios are discussed in Bird & de Moor (1997).

3 Refining the greedy step

We will prove the greedy condition (2.3) in Section 4. It will turn out that the proof makes use of properties of *gstep* that will be evident from its inductive definition. Therefore, we calculate an inductive definition of *gstep* in this section.

It is easy to derive $gstep\,[x] = [\,]$. For the inductive case, we reason

$$
\begin{array}{ll}
& gstep\,(x:y:xs) \\
= & \quad \{\text{ definition of } gstep \} \\
& maximum\,(drops\,(x:y:xs)) \\
= & \quad \{\text{ definition of } drops \} \\
& maximum\,((y:xs):map\,(x:)\,(drops\,(y:xs))) \\
= & \quad \{\text{ definition of } maximum \} \\
& max\,(y:xs)\,(maximum\,(map\,(x:)\,(drops\,(y:xs)))) \\
= & \quad \{\text{ since } maximum\,(map\,(x:)\,xss) = x : maximum\,xss, \text{ provided } xss \text{ is nonempty } \} \\
& max\,(y:xs)\,(x:maximum\,(drops\,(y:xs))) \\
= & \quad \{\text{ definition of } gstep \} \\
& max\,(y:xs)\,(x:gstep\,(y:xs)) \\
= & \quad \{\text{ definition of } max \text{ and lexicographic ordering } \} \\
& \textbf{if}\ x < y\ \textbf{then}\ y:xs \\
& \quad \textbf{else if}\ x == y\ \textbf{then}\ x : max\,xs\,(gstep\,(y:xs)) \\
& \qquad \textbf{else}\ x : gstep\,(y:xs) \\
= & \quad \{\text{ since } xs \trianglelefteq gstep\,(y:xs) \} \\
& \textbf{if}\ x < y\ \textbf{then}\ y:xs\ \textbf{else}\ x:gstep\,(y:xs) \quad .
\end{array}
$$

Hence, we have

$$
\begin{array}{ll}
gstep\,[x] & = [\,] \\
gstep\,(x:y:xs) & = \textbf{if}\ x < y\ \textbf{then}\ y:xs\ \textbf{else}\ x:gstep\,(y:xs) \quad .
\end{array}
$$

It turns out that *gstep xs* deletes *the last element of the longest descending prefix* of *xs*. For easy reference, we will refer to this element as the *hill foot* of the list. For example, *gstep* `"8755578"` = `"875578"`, where the hill foot, the element deleted, is the third `'5'`.

4 Proving the greedy condition

In this section, we aim to prove (2.3), recited here:

$$(\forall xs' \in step^{1+k}\,[xs] \bullet (\exists zs \in (step^k \circ wrap \circ gstep)\,xs \bullet xs' \trianglelefteq zs)) \quad .$$

Proving a proposition containing universal and existential quantification can be thought of as playing a game. The opponent challenges us by providing *xs* and a way of removing $1 + k$ elements to obtain xs'. We win by presenting a way of removing k elements from

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press

gstep xs, such that the result *zs* satisfies $xs' \trianglelefteq zs$. Equivalently, we win by presenting a way of removing $1+k$ elements from *xs*, while making sure that the hill foot of *xs* is among the $1+k$ elements removed. To prove (2.3) is to come up with a strategy to always win.

We could just invent the strategy and argue for its correctness. However, we experimented with another approach: could a proof assistant offer some help? Can we conjecture the existence of a function that computes *zs* given the opponent's input, and try to develop the function and the proof that $xs' \trianglelefteq zs$ at the same time, letting their developments mutually guide each other? It would be a modern realisation of Dijkstra's belief that a program and its proof should be developed hand in hand (Dijkstra, 1974).

We modelled the problem in the dependently typed language / proof assistant Agda (Norell, 2008). To be consistent with earlier parts of this pearl, we use Haskell-like notations for the Agda code. Typing relation is denoted by (::) and list cons by (:). The two constructors of *Nat* are 0 and (1+). Universally quantified implicit arguments in constructor and function declarations are omitted.

The datatypes. For the setting-up, we need to define a number of datatypes. Firstly, given a type *a* with a total ordering ($\leqslant$) (which derives a strict ordering ($<$)), lexicographic ordering for *List a* can be defined by:

$$
\begin{array}{l}
\textbf{data}\ \_\trianglelefteq\_ :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{Set}\ \textbf{where} \\
\quad []_{\trianglelefteq}\ \ ::\ [\,] \trianglelefteq xs \\
\quad <_{\trianglelefteq}\ ::\ x < y \quad \to (x:xs) \trianglelefteq (y:ys) \\
\quad \equiv_{\trianglelefteq}\ ::\ xs \trianglelefteq ys \to (x:xs) \trianglelefteq (x:ys)\ \ ,
\end{array}
$$

that is, $[\,]$ is no larger than any list, $(x:xs) \trianglelefteq (y:ys)$ if $x<y$, and two lists having the same head is compared by their tails.

Secondly, rather than actually deleting elements of a list, in proofs it helps to remember which elements are deleted. The following datatype *Dels k xs* can be seen as instructions on how *k* elements are deleted from *xs*:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Dels} :: \mathit{Nat} \to \mathit{List}\ a \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{end}\ \ :: \mathit{Dels}\ 0\ [\,] \\
\quad \mathit{keep} :: \mathit{Dels}\ k\ xs \to \mathit{Dels}\ k\ (x:xs) \\
\quad \mathit{del}\ \ \ :: \mathit{Dels}\ k\ xs \to \mathit{Dels}\ (1+k)\ (x:xs)\ \ .
\end{array}
$$

For example, letting $xs = $ `"abcde"`, $ds = \mathit{keep}\ (\mathit{del}\ (\mathit{keep}\ (\mathit{del}\ (\mathit{keep}\ \mathit{end})))) :: \mathit{Dels}\ 2\ xs$ says that the 1st and 3rd elements of *xs* (counting from 0) are to be deleted. The function *dels* actually carries out the instructions:

$$
\begin{array}{l}
\mathit{dels} :: (xs :: \mathit{List}\ a) \to \mathit{Dels}\ k\ xs \to \mathit{List}\ a \\
\mathit{dels}\ [\,] \qquad \mathit{end} \qquad\ \ = [\,] \\
\mathit{dels}\ (x:xs)\ (\mathit{keep}\ ds) = x : \mathit{dels}\ xs\ ds \\
\mathit{dels}\ (x:xs)\ (\mathit{del}\ ds)\ \ \ = \mathit{dels}\ xs\ ds\ \ .
\end{array}
$$

For example, $\mathit{dels}\ xs\ ds = $ `"ace"`.

Thirdly, the predicate *HFoot i xs* holds if the *i*-th element in *xs* is the hill foot, that is, the element that would be removed by *gstep xs*:

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press

$$\begin{aligned}
&\textbf{data}\ \mathit{HFoot} :: \mathit{Nat} \to \mathit{List}\ a \to \mathit{Set}\ \textbf{where}\\
&\quad \mathit{last}\ :: \mathit{HFoot}\ 0\ (x : [\,])\\
&\quad \mathit{this}\ :: x < y \to \mathit{HFoot}\ 0\ (x : y : xs)\\
&\quad \mathit{next} :: x \geqslant y \to \mathit{HFoot}\ i\ (y : xs) \to \mathit{HFoot}\ (1+i)\ (x : y : xs)\ \ .
\end{aligned}$$

For example, $\mathit{next}\ (\mathit{next}\ (\mathit{next}\ (\mathit{next}\ \mathit{this})))$ may have type $\mathit{HFoot}\ 4$ `"8755578"`, since the 4th element is the last in the descending prefix `"87555"`.

Finally, we define a datatype $\mathit{IsDel} :: \mathit{Nat} \to \mathit{Dels}\ k\ xs \to \mathit{Set}$ such that, for all $ds : \mathit{Dels}\ k\ xs$, the relation $\mathit{IsDel}\ i\ ds$ holds if ds instructs that the i-th element of xs is to be deleted. Its definition is repetitive and thus omitted here.

The function and the proofs. The aim is to construct the following function *alter*:

$$\mathit{alter} :: \mathit{Dels}\ (1+k)\ xs \to \mathit{HFoot}\ i\ xs \to \mathit{Dels}\ (1+k)\ xs\ \ .$$

It takes an instruction, given by the opponent, that deletes $1+k$ elements from xs, and an evidence that the i-th element of xs is its hill foot, and produces a possibly altered instruction that also deletes $1+k$ elements. Recalling the discussion in the beginning of this section, *alter* should satisfy two properties:

$$\begin{aligned}
&\mathit{mono}\ \ :: (ds :: \mathit{Dels}\ (1+k)\ xs) \to (\mathit{ft} :: \mathit{HFoot}\ i\ xs) \to \mathit{dels}\ xs\ ds \trianglelefteq \mathit{dels}\ xs\ (\mathit{alter}\ ds\ \mathit{ft})\ \ ,\\
&\mathit{unfoot} :: (ds :: \mathit{Dels}\ (1+k)\ xs) \to (\mathit{ft} :: \mathit{HFoot}\ i\ xs) \to \mathit{IsDel}\ i\ (\mathit{alter}\ ds\ \mathit{ft})\ \ .
\end{aligned}$$

Given ds and ft, the property *mono* says that $\mathit{alter}\ ds\ \mathit{ft}$ always produces a list that is not worse than that produced by ds, while *unfoot* says that $\mathit{alter}\ ds\ \mathit{ft}$ does delete the hill foot.

The goal now is to develop *alter*, *mono* and *unfoot* together. The reader is invited to give it a try—it is more fun trying it yourself![1] In most of the steps, the type and proof constraints leave us with only one reasonable choice, while in one case we are led to discover a lemma. In the following, we go through each case. A visualisation of each case is given in Figure 1. The cases to consider are[2]

1. $\mathit{alter}\ \{xs = x : y : ys\}\ (\mathit{keep}\ ds)\ (\mathit{this}\ x{<}y)$—the opponent keeps x, which is the hill foot because $x < y$. Due to *unfoot*, we have to delete x; a simple way to satisfy *mono* is to keep y. Thus, we return $\mathit{del}\ (\mathit{keep}\ ds')$, where ds' can be any instruction that deletes k elements in xs—it doesn't matter how ds' does it!
2. $\mathit{alter}\ \{xs = x : y : ys\}\ (\mathit{keep}\ ds)\ (\mathit{next}\ x{\geqslant}y\ \mathit{ft})$—the opponent keeps x, and we have not reached the hill foot yet. In this case, it is safe to imitate the opponent and keep x too, before recursively calling *alter* to generate the rest of the instruction.
3. $\mathit{alter}\ \{xs = [x]\}\ (\mathit{del}\ \mathit{end})\ \mathit{last}$—the opponent deletes the sole element x. In this case, we delete x too, returning $\mathit{del}\ \mathit{end}$.
4. $\mathit{alter}\ \{xs = x : y : ys\}\ (\mathit{del}\ ds)\ (\mathit{this}\ x{<}y)$—the element x is the hill foot and is deleted by the opponent. In this case, since *unfoot* is satisfied, we can do exactly the same. We end up returning the same instruction as the opponent's but it is fine, since both *mono* and *unfoot* are satisfied.

[1] The Agda code can be downloaded from https://scm.iis.sinica.edu.tw/home/2020/dropping-digits/. Also available as web-accessible accompanying material of JFP.

[2] Curly brackets are used in Agda to mention implicit arguments. In each case here, we pattern-match $\{xs = \ldots\}$ such that the readers know what input list we are dealing with.

```
alter :: Dels (1 + k) xs → HFoot i xs → Dels (1 + k) xs
alter (keep ds)   (this x<y)    = del (keep (deleteAny ds))
alter (keep ds)   (next x⩾y ft) = keep (alter ds ft)
alter (del end)   last          = del end
alter (del ds)    (this x<y)    = del ds
alter {k = 0}     (del ds) (next x⩾y ft) = keep (delfoot ft)
alter {k = 1 + k} (del ds) (next x⩾y ft) = del (alter ds ft)
```

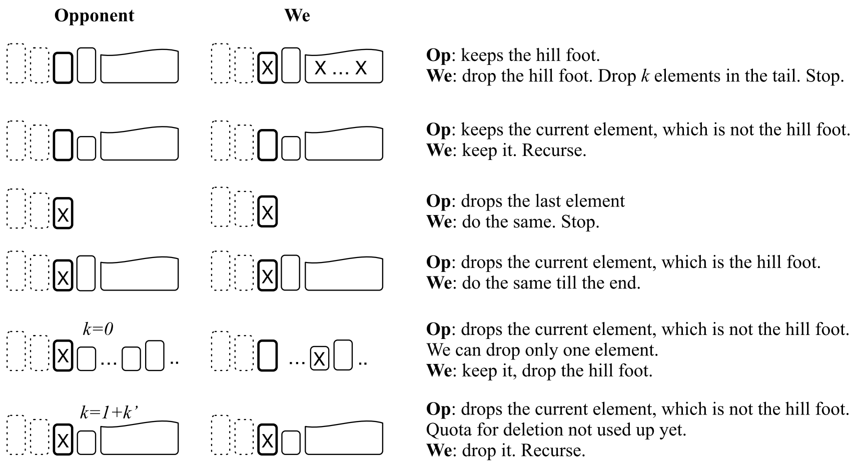

Fig. 1. The function *alter*, where *deleteAny ds* generates a *Dels k xs*, and a graphical summary. Elements with dotted outlines are those that are considered already; the one with thick outline is the current element. It is the hill foot if it is smaller than the element to the right, or if itself is the last element of the input. Deleted elements are marked with a cross.

5. $alter\ \{xs = x : y : ys\}\ (del\ ds)\ (next\ x{\geqslant}y\ ft)$—the opponent deletes x, which is in the descending prefix but is not the hill foot. This turns out to be the most complex case. One may try to imitate and delete x as well, returning $del\ ds'$ for some ds'. However, ds', having type $Dels\ k\ (y : ys)$, cannot be produced by a recursive call to *alter*, whose return type is $Dels\ (1 + k)\ \_$. It could be the case that k is 0 and, since we have not deleted the hill foot yet, returning $Dels\ 0\ (y : ys)$ would violate *unfoot*. The lesson learnt from the type is that we can only delete $1 + k$ elements, and we have to save at least one *del* for the hill foot, which is yet to come. We thus have to further distinguish between two cases:

   a. $k = 1 + k'$ for some k'. In this case, we still have room to delete more elements, thus we can safely imitate the opponent, delete x, and recursively call *alter*.
   b. $k = 0$. In this case, we keep x, returning $keep\ (delfoot\ ft) :: Dels\ 1\ (y : ys)$, where $delfoot :: HFoot\ i\ zs \to Dels\ 1\ zs$ computes an instruction that deletes exactly one element, the hill foot. What is left to prove to establish *mono* for this case can be extracted to be a lemma:

$$monoAux :: x \geqslant y \to (ft :: HFoot\ i\ (y : ys)) \to$$
$$(y : ys) \trianglelefteq (x : dels\ (y : ys)\ (delfoot\ ft)) \quad ,$$

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press

> whose proof is an induction on ys, keeping $x \geqslant y$ as an invariant. If $x < y$ or y is the hill foot, we are done. Otherwise, $x = y$ and we inductively inspect the tail ys. Without Agda, it would not be easy to discover this lemma.

In summary, the function *alter* we have constructed is shown in Figure 1.

Remark: We may also tuple *alter* and the properties together and try to construct:

$$\mathit{alter}' :: (\mathit{ds} :: \mathit{Dels}\,(1+k)\,\mathit{xs}) \rightarrow (\mathit{ft} :: \mathit{HFoot}\,i\,\mathit{xs}) \rightarrow$$
$$\exists\,(\lambda(\mathit{ds}' :: \mathit{Dels}\,(1+k)\,\mathit{xs}) \rightarrow (\mathit{dels}\,\mathit{xs}\,\mathit{ds} \trianglelefteq \mathit{dels}\,\mathit{xs}\,\mathit{ds}') \times \mathit{IsDel}\,i\,\mathit{ds}')$$

An advantage is that the code of each case of *alter* are next to its proof. A disadvantage is that we have to pattern-match the result of *alter′*. It is up to personal preference which style one prefers.

5 Improving efficiency

We have proved that $\mathit{solve}\,k = \mathit{gstep}^k$, where *gstep* is given by:

$$\begin{array}{ll} \mathit{gstep}\,[x] & = [\,] \\ \mathit{gstep}\,(x:y:\mathit{xs}) & = \mathbf{if}\ x < y\ \mathbf{then}\ y:\mathit{xs}\ \mathbf{else}\ x:\mathit{gstep}\,(y:\mathit{xs}) \quad . \end{array}$$

Each time *gstep* is called, it takes $O(n)$ steps to go through the descending prefix and find the hill foot, before the next invocation of *gstep* starts from the beginning of the list again. Therefore, *solve k* takes $O(kn)$ steps over all. This is certainly not necessary—to find the next hill foot, the next *gstep* could start from where the previous one left off.

The way to implement this idea is to bring in an accumulating parameter. Suppose we generalise *solve* to a function *gsolve*, defined by:

$$\mathit{gsolve}\,k\,\mathit{xs}\,\mathit{ys} = \mathit{solve}\,k\,(\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) \quad ,$$

with the proviso that the argument *xs* is constrained to be a descending sequence. In particular, $\mathit{solve}\,k\,\mathit{xs} = \mathit{gsolve}\,k\,[\,]\,\mathit{xs}$. We aim to develop a recursive definition of *gsolve*. Clearly, $\mathit{gsolve}\,0\,\mathit{xs}\,\mathit{ys} = \mathit{xs} \mathbin{+\!\!+} \mathit{ys}$. Recalling that *gstep* drops the last element of a descending list, we know that k repetitions of *gstep* on a decreasing list will drop the last k elements. Hence,

$$\mathit{gsolve}\,k\,\mathit{xs}\,[\,] = \mathit{dropLast}\,k\,\mathit{xs} \quad ,$$

where *dropLast k* drops the last k elements of a list. We will not give a formal definition of *dropLast* as it will be replaced by another function in a moment. That deals with the two base cases. For the recursive case, it is easy to prove the following property of *gstep*:

$$\begin{array}{lll} \mathit{gstep}\,(\mathit{xs} \mathbin{+\!\!+} y:\mathit{ys}) & \mid \mathit{null}\,\mathit{xs} \vee \mathit{last}\,\mathit{xs} \geqslant y & = \mathit{gstep}\,((\mathit{xs} \mathbin{+\!\!+} [y]) \mathbin{+\!\!+} \mathit{ys}) \\ & \mid \mathit{otherwise} & = \mathit{init}\,\mathit{xs} \mathbin{+\!\!+} y:\mathit{ys} \quad , \end{array}$$

which can be used to construct the following case of *gsolve*:

$$\begin{array}{lll} \mathit{gsolve}\,(1+k)\,\mathit{xs}\,(y:\mathit{ys}) & \mid \mathit{null}\,\mathit{xs} \vee \mathit{last}\,\mathit{xs} \geqslant y & = \mathit{gsolve}\,(1+k)\,(\mathit{xs} \mathbin{+\!\!+} [y])\,\mathit{ys} \\ & \mid \mathit{otherwise} & = \mathit{gsolve}\,k\,(\mathit{init}\,\mathit{xs})\,(y:\mathit{ys}) \quad . \end{array}$$

The second optimisation is simply to replace the list *xs* in the definition of *gsolve* by *reverse xs* to avoid adding elements at the end of a list. That leads to our final algorithm:

$$\mathit{solve}\ k\ \mathit{xs} = \mathit{gsolve}\ k\ [\,]\ \mathit{xs}\ \ ,$$

$$\begin{array}{lll}
\mathit{gsolve}\ 0\ \mathit{xs}\ \mathit{ys} & & = \mathit{reverse}\ \mathit{xs} +\!\!+\ \mathit{ys} \\
\mathit{gsolve}\ k\ \mathit{xs}\ [\,] & & = \mathit{reverse}\ (\mathit{drop}\ k\ \mathit{xs}) \\
\mathit{gsolve}\ k\ \mathit{xs}\ (y:\mathit{ys}) & \mid \mathit{null}\ \mathit{xs} \vee \mathit{head}\ \mathit{xs} \geqslant y & = \mathit{gsolve}\ k\ (y:\mathit{xs})\ \mathit{ys} \\
 & \mid \mathit{otherwise} & = \mathit{gsolve}\ (k-1)\ (\mathit{tail}\ \mathit{xs})\ (y:\mathit{ys})\ \ ,
\end{array}$$

where *drop k* is a standard Haskell function that drops the first k elements from a list. For an operational explanation, *gsolve* traverses through the list, keeping looking for the next hill foot to delete—the *otherwise* case is when a hill foot is found. The list *xs* is the traversed part—(*xs*, *ys*) forms a zipper. The head of *xs* is a possible candidate of the hill foot. While the algorithm looks simple once understood, without calculation it is not easy to get the details right. The authors have come up with several versions that are wrong, before sitting down to calculate it!

To time the program, note that at each step either k is reduced to $k-1$ or $y:ys$ is reduced to *ys*. Hence, *solve k xs* takes $O\,(k+n) = O\,(n)$ steps, where $n = \mathit{length}\ \mathit{xs}$.

6 Conclusion

To construct a linear-time algorithm for solving the puzzle, various techniques were employed. The structure of the greedy algorithm was proved using predicate logic, and the proof was simplified from relational program calculus. Agda was used to give a constructive proof of the greedy condition, and equational reasoning was used to derive the greedy step as well as the final, linear-time optimisation.

Conflicts of Interest

None.

References

Bird, R. S. & de Moor, O. (1997) *Algebra of Programming*. International Series in Computer Science. Prentice Hall.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*. MIT.

Curtis, S. (2003) The classification of greedy algorithms. *Sci. Comput. Program.* **49**(1–3), 125–157.

Dijkstra, E. W. (1974) Programming as a discipline of mathematical nature. *Am. Math. Monthly* **81**(6), 608–612. EWD 361.

Korte, B., Lovász, L. & Schrader, R. (1991) *Greedoids*. Springer-Verlag.

Lawler, E. L. (1976) *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston.

LeetCode. (2016) *402. Remove k digits*. https://leetcode.com/problems/remove-k-digits/.

Norell, U. (2008) Dependently typed programming in Agda. In *Advanced Functional Programming*, Koopman, P., Plasmeijer, R. & Swierstra, S. D. (eds). Springer, pp. 230–266.

https://doi.org/10.1017/S0956796821000198 Published online by Cambridge University Press