JFP **34**, e14, 16 pages, 2024.

doi:10.1017/S0956796824000145

FUNCTIONAL PEARL

Bottom-up computation using trees of sublists

SHIN-CHENG MU

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

Abstract

Some top-down problem specifications, if executed, may compute sub-problems repeatedly. Instead, we may want a bottom-up algorithm that stores solutions of sub-problems in a table to be reused. How the table can be represented and efficiently maintained, however, can be tricky. We study a special case: computing a function h taking lists as inputs such that $h\,xs$ is defined in terms of all immediate sublists of xs. Richard Bird studied this problem in 2008 and presented a concise but cryptic algorithm without much explanation. We give this algorithm a proper derivation and discovered a key property that allows it to work. The algorithm builds trees that have certain shapes—the sizes along the left spine is a prefix of a diagonal in Pascal's triangle. The crucial function we derive transforms one diagonal to the next.

1 Introduction

A list *ys* is said to be an *immediate sublist* of *xs* if *ys* can be obtained by removing exactly one element from *xs*. For example, the four immediate sublists of `"abcd"` are `"abc"`, `"abd"`, `"acd"`, and `"bcd"`. Consider computing a function h that takes a list as input, with the property that the value of $h\,xs$ depends on values of h at all the immediate sublists of *xs*. For example, as seen in Figure 1, h `"abcd"` depends on h `"abc"`, h `"abd"`, h `"acd"`, and h `"bcd"`. In this top-down manner, to compute h `"abc"`, we make calls to h `"ab"`, h `"ac"`, and h `"bc"`; to compute h `"abd"`, we make a call to h `"ab"` as well—many values end up being re-computed. One would like to instead proceed in a bottom-up manner, storing computed values so that they can be reused. For this problem, one might want to build a lattice-like structure, like that in Figure 2, from bottom to top, such that each level reuses values computed in the level below it.

Bird (2008) presented a study of the relationship between top-down and bottom-up algorithms. It was shown that if an algorithm can be written in a specific top-down style, with ingredients that satisfy certain properties, there is an equivalent bottom-up algorithm that stores intermediate results in a table. The "all immediate sublists" instance was the last example of the paper. To tackle the problem, however, Bird had to introduce, out of the

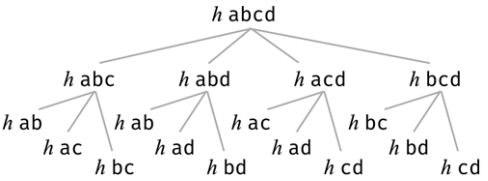


Fig. 1. Computing *h* `"abcd"` top-down. String constants are shown using monospace font but without quotes, to save space.

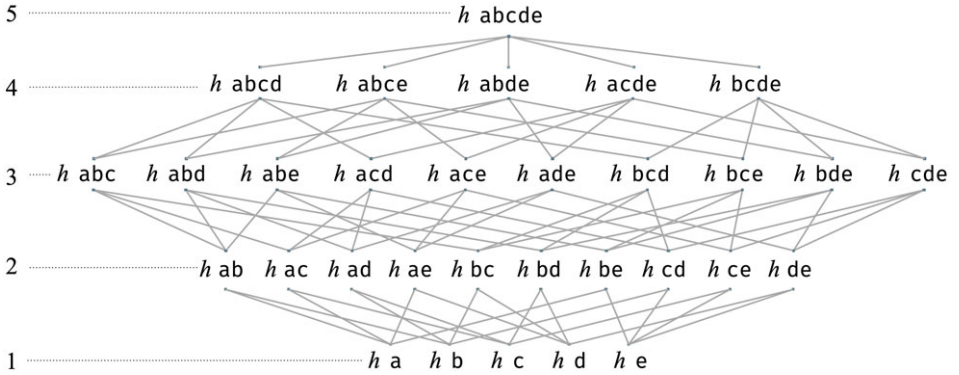


Fig. 2. Computing *h* `"abcde"` bottom-up. The numbers on the left denote the levels.

blue, a binary tree and a concise but cryptic four-line function. The tree appears to obey some shape constraints, but that was not explicitly stated. Regarding the four-line function, which lies at the core of the bottom-up algorithm, we know from its type that it turns a tree into a tree of lists, and that is probably all one can say with confidence. Not only is it hard to see what the function exactly does, it is even not easy to see why the function, involving use of partial operations such as zipping trees of the same shape, always succeeds. Given limited space, Bird did not offer much rationale or explanation, nor did he prove that the function satisfies the said properties that should be met by a bottom-up algorithm.

The author finds this algorithm fascinating and struggled to understand it. As Bird would agree, a good way to understand an algorithm is to calculate it; thus, this pearl came into being. In this pearl, we review this problem, reveal a connection between "n choose k" and "n choose $1 + k$" that was not explicit in Bird (2008), motivate the introduction of the tree, and finally construct a formal specification of the four-line function (which we call *up* in this pearl). Once we have a specification, *up* can be calculated—not without some tricky eureka that made the calculation fun. It then turns out that there is a formula describing the role of *up* in the bottom-up algorithm that is different and simpler than that in Bird (2008).

One might ask: are there actually such problems, whose solution of input *xs* depends on solutions of immediate sublists of *xs*? It turns out that it is well known in the algorithm community that, while problems such as *minimum edit distance* or *longest common subsequence* are defined on two lists, with clever encoding, they can be rephrased as problems defined on one list whose solution depends on immediate sublists. Many problems in additive combinatorics (Tao & Vu, 2012) can also be cast into this form.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

But those are just bonuses. The application of a puzzle is being solved, a good puzzle is one that is fun to solve, and a functional pearl is a story about solving a puzzle. One sees a problem, wonders whether there is an elegant solution, finds the right specification, tries to calculate it, encounters some head-scratching moments and surprising twists alone the way, but eventually overcomes the obstacle and comes up with a concise and beautiful algorithm. One then writes about the adventure, to share the fun.

2 Specification

We use a Haskell-like notation throughout the paper. Like in Haskell, if a function is defined by multiple clauses, the patterns and guards are matched in the order they appear. Differences from Haskell include that we allow $n + k$ pattern, and that we denote the type of list by L, equipped with its *map* function $map :: (a \to b) \to \mathsf{L}\, a \to \mathsf{L}\, b$.

The immediate sublists of a list can be specified in many ways. We use the definition below because it allows the proofs of this pearl to proceed in terms of cons-lists, which is familiar to most readers, while it also generates sublists in an order that is more intuitive:

$$\begin{aligned} &subs :: \mathsf{L}\, a \to \mathsf{L}\, (\mathsf{L}\, a) \\ &subs\, [\,] \quad\;\; = [\,] \\ &subs\, (x : xs) = map\, (x:)\, (subs\, xs) +\!\!+ [xs] \quad . \end{aligned}$$

For example, *subs* `"abcde"` yields [`"abcd"`, `"abce"`, `"abde"`, `"acde"`, `"bcde"`].

Denote the function we wish to compute by $h :: \mathsf{L}\, \mathsf{X} \to \mathsf{Y}$ for some types X and Y. We assume that it is a partial function defined on non-empty lists and can be computed top-down as below:

$$\begin{aligned} &h :: \mathsf{L}\, \mathsf{X} \to \mathsf{Y} \\ &h\, [x] = f\, x \\ &h\, xs \;\; = (g \circ map\, h \circ subs)\, xs \quad , \end{aligned}$$

where $f :: \mathsf{X} \to \mathsf{Y}$ is used in the base case when the input is a singleton list, and $g :: \mathsf{L}\, \mathsf{Y} \to \mathsf{Y}$ is for the inductive case.

For this pearl, it is convenient to use an equivalent definition. Let *td* (referring to "top-down") be a family of functions indexed by natural numbers (denoted by Nat):

$$\begin{aligned} &td :: \mathsf{Nat} \to \mathsf{L}\, \mathsf{X} \to \mathsf{Y} \\ &td\, 0 \quad\;\; = f \circ ex \\ &td\, (1 + n) = g \circ map\, (td\, n) \circ subs \quad , \end{aligned}$$

Here, the function $ex :: \mathsf{L}\, a \to a$ takes a singleton list and extracts the only component. The intention is that $td\, n$ is a function defined on lists of length exactly $1 + n$. The call $td\, n$ in the body of $td\, (1 + n)$ is defined because *subs*, given an input of length $1 + n$, returns lists of length n. Given input xs, the value we aim to compute is $h\, xs = td\, (length\, xs - 1)\, xs$.

The function $repeat\, k$ composes a function with itself k times:

$$\begin{aligned} &repeat :: \mathsf{Nat} \to (a \to a) \to a \to a \\ &repeat\, 0 \quad\;\; f = id \\ &repeat\, (1 + k)\, f = repeat\, k\, f \circ f \quad . \end{aligned}$$

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

For brevity, we will write *repeat k f* as f^k for the rest of this pearl. We aim to construct a bottom-up algorithm having the following form:

$$bu\, n = post \circ step^n \circ pre \ ,$$

where *pre* preprocesses the input and builds the lowest level in Figure 2, and each *step* builds a level from the one below. For input of length $1 + n$, we repeat n times and, by then, we can extract the singleton value by *post*.

Our aim is to construct *pre*, *step*, and *post* such that $td = bu$.

3 Building a new level

To find out what *step* might be, we need to figure out how to specify a level, and what happens when a level is built from the one below it. We use Figure 2 as our motivating example. As one can see, level 2 in Figure 2 consists of sublists of `"abcde"` that have length 2, and level 3 consists of sublists having length 3, etc. Let *choose k xs* denote choosing k elements from the list *xs*:

$$\begin{aligned}
&choose :: \mathsf{Nat} \to \mathsf{L}\ a \to \mathsf{L}\ (\mathsf{L}\ a)\\
&choose\ 0 \quad\ _ \quad\ = [[\,]]\\
&choose\ k \quad\ xs \quad\ \mid k == length\ xs = [xs]\\
&choose\ (1+k)\ (x : xs) = map\ (x:)\ (choose\ k\ xs) +\!\!+ choose\ (1+k)\ xs \ .
\end{aligned}$$

Its definition follows basic combinatorics: the only way to choose 0 elements from a list is `[]`; if $length\ xs = k$, the only way to choose k elements is *xs*. Otherwise, to choose $1 + k$ elements from $x : xs$, one can either keep x and choose k from *xs*, or choose $1 + k$ elements from *xs*. For example, *choose* 3 `"abcde"` yields `["abc", "abd", "abe", "acd", "ace", "ade", "bcd", "bce", "bde", "cde"]`. Note that *choose k xs* is defined only when $k \leqslant length\ xs$.

If the levels in Figure 2 were to be represented as lists, each level k is given by *map h* (*choose k xs*). For example, level 2 in Figure 2 is (string literals are shown in typewriter font; double quotes are omitted to reduce noise in the presentation):

$$map\ h\ (choose\ 2\ \texttt{abcde}) = [h\ \texttt{ab}, h\ \texttt{ac}, h\ \texttt{ad}, h\ \texttt{ae}, h\ \texttt{bc}, h\ \texttt{bd}, h\ \texttt{be}, h\ \texttt{cd}, h\ \texttt{ce}, h\ \texttt{de}] \ .$$

To build level 3 from level 2, we wish to have a function *upgrade* :: $\mathsf{L}\ \mathsf{Y} \to \mathsf{L}\ (\mathsf{L}\ \mathsf{Y})$ that is able to somehow bring together the relevant entries from level 2:

$$\begin{aligned}
&upgrade\ (map\ h\ (choose\ 2\ \texttt{abcde})) =\\
&\quad [[h\ \texttt{ab}, h\ \texttt{ac}, h\ \texttt{bc}], [h\ \texttt{ab}, h\ \texttt{ad}, h\ \texttt{bd}], [h\ \texttt{ab}, h\ \texttt{ae}, h\ \texttt{be}]...] \ .
\end{aligned}$$

With $[h\ \texttt{ab}, h\ \texttt{ac}, h\ \texttt{bc}]$ one can compute $h\ \texttt{abc}$, and with $[h\ \texttt{ab}, h\ \texttt{ad}, h\ \texttt{bd}]$ one can compute $h\ \texttt{abd}$, etc. That is, if we apply *map g* to the result of *upgrade* above, we get

$$[h\ \texttt{abc}, h\ \texttt{abd}, h\ \texttt{abe}, h\ \texttt{acd}...] \ ,$$

which is level 3, or *map h* (*choose* 3 `abcde`). The function *upgrade* need not inspect the values of each element, but rearrange them by position—it is a natural transformation $\mathsf{L}\ a \to \mathsf{L}\ (\mathsf{L}\ a)$. As far as *upgrade* is concerned, it does not matter whether h is applied or not.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

Letting $h = id$, observe that $upgrade\,(choose\,2\,\texttt{abcde}) = [[\texttt{ab}, \texttt{ac}, \texttt{bc}], [\texttt{ab}, \texttt{ad}, \texttt{bd}]...]$ and $choose\,3\,\texttt{abcde} = [\texttt{abc}, \texttt{abd}, \texttt{abe}, \texttt{acd}...]$ are related by *map subs*. Each *step* we perform in the bottom-up algorithm could be $map\,g \circ upgrade$.

Formalizing the observations above, we want $upgrade :: \mathsf{L}\,a \to \mathsf{L}\,(\mathsf{L}\,a)$ to satisfy:

$$\begin{aligned}&(\forall xs, k : 2 \leqslant 1 + k \leqslant length\,xs:\\ &\quad upgrade\,(choose\,k\,xs) = map\,subs\,(choose\,(1+k)\,xs))\ .\end{aligned} \tag{3.1}$$

Given (3.1), we may let each step in the bottom-up algorithm be $map\,g \circ upgrade$.

Equation (3.1) is a specification of *upgrade*, constructed by observation and generalization. We want it to serve two purposes: 1. we wish to calculate from it a definition of *upgrade*, and 2. it plays a central role in proving that the bottom-up algorithm, also to be constructed, equals the top-down algorithm. That (3.1) (in fact, a stronger version of it) does meet the purposes above will be formally justified later. For now, we try to gain some intuition by demonstrating below that, with (3.1) satisfied, $map\,g \circ upgrade$ builds level $k+1$ from level k. Let the input be xs. If xs is a singleton list, the bottom-up algorithm has finished, so we consider xs having length at least 2. Recall that level k is $map\,h\,(choose\,k\,xs)$. Applying $map\,g \circ upgrade$ to level k:

$$\begin{aligned}&map\,g\,(upgrade\,(map\,h\,(choose\,k\,xs)))\\ =\quad &\{\ upgrade \text{ natural}\ \}\\ &map\,g\,(map\,(map\,h)\,(upgrade\,(choose\,k\,xs)))\\ =\quad &\{\ \text{by (3.1)}, map\text{-fusion}\ \}\\ &map\,(g \circ map\,h \circ subs)\,(choose\,(1+k)\,xs)\\ =\quad &\{\ \text{definition of } h\ \}\\ &map\,h\,(choose\,(1+k)\,xs)\ .\end{aligned}$$

We get level $1 + k$.

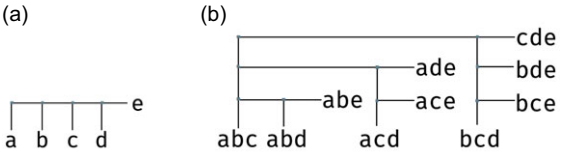
The constraints on k in (3.1) may need some explanation. For $choose\,(1+k)\,xs$ on the RHS to be defined, we need $1 + k \leqslant length\,xs$. Meanwhile, no *upgrade* could satisfy (3.1) when $k = 0$: on the LHS, *upgrade* cannot distinguish between $choose\,0\,\texttt{ab}$ and $choose\,0\,\texttt{abc}$, both evaluating to $[[\,]]$, while on the RHS $choose\,1\,\texttt{ab}$ and $choose\,1\,\texttt{abc}$ have different shapes. Therefore, we only demand (3.1) to hold when $1 \leqslant k$, which is sufficient because we only apply *upgrade* to level 1 and above. Together, the constraint is $2 \leqslant 1 + k \leqslant length\,xs$ — xs should have at least 2 elements.

Can we construct such an *upgrade*?

4 Building levels represented by trees

We may proceed with (3.1) and construct *upgrade*. We will soon meet a small obstacle: in an inductive case *upgrade* will receive a list computed by $choose\,(1+k)\,(x : xs)$, that is, $map\,(x:)\,(choose\,k\,xs)$ and $choose\,(1+k)\,xs$ concatenated by $(+\!\!+)$, and split it back to the two lists. This can be done, but it is rather tedious. This is a hint that some useful information has been lost when we represent levels by lists. To make the job of *upgrade* easier, we switch to a more informative data structure.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press



ch 1 abcde. *ch* 3 abcde.

Fig. 3. Results of *ch*.

4.1 Binomial trees

Instead of lists, we define the following tip-valued binary tree:

$$\textbf{data}\ \mathsf{B}\ a = \mathsf{T}\ a \mid \mathsf{N}\ (\mathsf{B}\ a)\ (\mathsf{B}\ a)\ .$$

We assume that B is equipped with two functions derivable from its definition:

$$\begin{aligned} &\mathit{mapB}\ :: (a \to b) \to \mathsf{B}\ a \to \mathsf{B}\ b\ , \\ &\mathit{zipBW} :: (a \to b \to c) \to \mathsf{B}\ a \to \mathsf{B}\ b \to \mathsf{B}\ c\ . \end{aligned}$$

The function $\mathit{mapB}\ f$ applies f to every tip of the given tree. Given two trees t and u having the same shape, $\mathit{zipBW}\ f\ t\ u$ "zips" the trees together, applying f to values on the tips—the name stands for "zip B-trees with". If t and u have different shapes, $\mathit{zipBW}\ f\ t\ u$ is undefined. Furthermore, for the purpose of specification we assume a function $\mathit{flatten} :: \mathsf{B}\ a \to \mathsf{L}\ a$ that "flattens" a tree to a list by collecting all the values on the tips left-to-right.

Having B allows us to define an alternative to *choose*:

$$\begin{aligned} &\mathit{ch} :: \mathsf{Nat} \to \mathsf{L}\ a \to \mathsf{B}\ (\mathsf{L}\ a) \\ &\mathit{ch}\ 0 \quad\ \_ \quad\ = \mathsf{T}\ [\,] \\ &\mathit{ch}\ k \quad\ \mathit{xs} \quad\ \mid\ k == \mathit{length}\ \mathit{xs} = \mathsf{T}\ \mathit{xs} \\ &\mathit{ch}\ (1+k)\ (x : \mathit{xs}) = \mathsf{N}\ (\mathit{mapB}\ (x:)\ (\mathit{ch}\ k\ \mathit{xs}))\ (\mathit{ch}\ (1+k)\ \mathit{xs})\ . \end{aligned}$$

The function *ch* resembles *choose*. In the first two clauses, T corresponds to a singleton list. In the last clause, *ch* is like *choose* but, instead of appending the results of the two recursive calls, we store the results in the two branches of the binary tree, thus recording how the choices were made: if $\mathit{ch}\ \_\ (x : \mathit{xs}) = \mathsf{N}\ t\ u$, the subtree t contains all the tips with x chosen, while u contains all the tips with x discarded. See Figure 3 for some examples. We have $\mathit{flatten}\ (\mathit{ch}\ k\ \mathit{xs}) = \mathit{choose}\ k\ \mathit{xs}$, that is, *choose* forgets the information retained by *ch*.

The counterpart of *upgrade* on trees, which we will call *up*, will be a natural transformation of type $\mathsf{B}\ a \to \mathsf{B}\ (\mathsf{L}\ a)$. Its relationship to *upgrade* is given by $\mathit{flatten}\ (\mathit{up}\ (\mathit{ch}\ k\ \mathit{xs})) = \mathit{upgrade}\ (\mathit{choose}\ k\ \mathit{xs})$. The function *up* should satisfy the following specification:

$$\begin{aligned} &(\forall \mathit{xs}, k : 2 \leqslant 1 + k \leqslant \mathit{length}\ \mathit{xs}\ : \\ &\qquad \mathit{up}\ (\mathit{ch}\ k\ \mathit{xs}) = \mathit{mapB}\ \mathit{subs}\ (\mathit{ch}\ (1+k)\ \mathit{xs}))\ . \end{aligned} \tag{4.1}$$

It is a stronger version of (3.1)–(4.1) reduces to (3.1) if we apply *flatten* to both sides.

Now we are ready to derive *up*.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

4.2 The derivation

The derivation proceeds by trying to construct a proof of (4.1) and, when stuck, pausing to think about how *up* should be defined to allow the proof to go through. That is, the definition of *up* and a proof that it satisfies (4.1) are developed hand-in-hand.

The proof, if constructed, will be an induction on *xs*. The case analysis follows the shape of $ch\,(1+k)\,xs$ (on the RHS of (4.1)). Therefore, there is a base case, a case when *xs* is non-empty and $1+k = length\,xs$, and a case when $1+k < length\,xs$. However, since the constraints demand that *xs* has at least two elements, the base case will be lists of length 2, and in the inductive cases the length of the list will be at least 3.

Case 1. $xs := [y,z]$.[1]
The constraints force *k* to be 1. We simplify the RHS of (4.1):

$$\begin{array}{ll} & mapB\ subs\ (ch\ 2\ [y,z]) \\ = & \quad \{\ \text{def. of } ch\ \} \\ & mapB\ subs\ (\mathsf{T}\ [y,z]) \\ = & \quad \{\ \text{def. of } mapB \text{ and } subs\ \} \\ & \mathsf{T}\ [[y],[z]]\ . \end{array}$$

Now consider the LHS:

$$\begin{array}{ll} & up\ (ch\ 1\ [y,z]) \\ = & \quad \{\ \text{def. of } ch\ \} \\ & up\ (\mathsf{N}\ (\mathsf{T}\ [y])\ (\mathsf{T}\ [z]))\ . \end{array}$$

The two sides can be made equal if we let $up\ (\mathsf{N}\ (\mathsf{T}\ p)\ (\mathsf{T}\ q)) = \mathsf{T}\ [p,q]$.

Case 2. $xs := x : xs$ where $length\,xs \geqslant 2$, and $1+k = length\,(x:xs)$.
We leave details of this case to the readers as an exercise, since we would prefer giving more attention to the next case. For this case, we will construct

$$up\ (\mathsf{N}\ t\ (\mathsf{T}\ q)) = \mathsf{T}\ (unT\ (up\ t) +\!\!+ [q])\ .$$

In this case, $up\ t$ always returns a T. The function $unT\ (\mathsf{T}\ p) = p$ removes the constructor and exposes the list it contains. While the correctness of this case is established by the constructed proof, a complementary explanation why $up\ t$ always returns a singleton tree and thus *unT* always succeeds is given in Section 4.3.

Case 3. $xs := x : xs$, $k := 1+k$, where $length\,xs \geqslant 2$, and $1+(1+k) < length\,(x:xs)$.
The constraints become $2 \leqslant 2+k < length\,(x:xs)$. Again we start with the RHS and try to reach the LHS:

$$\begin{array}{ll} & mapB\ subs\ (ch\ (2+k)\ (x:xs)) \\ = & \quad \{\ \text{def. of } ch, \text{ since } 2+k < length\,(x:xs)\ \} \\ & mapB\ subs\ (\mathsf{N}\ (mapB\ (x:)\ (ch\ (1+k)\ xs))\ (ch\ (2+k)\ xs)) \\ = & \quad \{\ \text{def. of } mapB, mapB\text{-fusion}\ \} \end{array}$$

[1] The (:=) denotes substitution: we mean that the property being proved is (4.1) with $[y,z]$ substituted for *xs*.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

$$\begin{aligned} & \mathsf{N}\,(mapB\,(subs \circ (x:))\,(ch\,(1+k)\,xs))\,(mapB\,subs\,(ch\,(2+k)\,xs)) \\ = \quad & \{\ \text{induction}\ \} \\ & \mathsf{N}\,(mapB\,(subs \circ (x:))\,(ch\,(1+k)\,xs))\,(up\,(ch\,(1+k)\,xs))\ . \end{aligned} \tag{4.2}$$

Note that the induction step is valid because we are performing induction on xs, and thus k in (4.1) is universally quantified. We now look at the LHS:

$$\begin{aligned} & up\,(ch\,(1+k)\,(x:xs)) \\ = \quad & \{\ \text{def. of } ch, \text{ since } 1+k < length\,(x:xs)\ \} \\ & up\,(\mathsf{N}\,(mapB\,(x:)\,(ch\,k\,xs))\,(ch\,(1+k)\,xs))\ . \end{aligned} \tag{4.3}$$

Expressions (4.2) and (4.3) can be unified if we define

$$up\,(\mathsf{N}\,t\,u) = \mathsf{N}\,???\,(up\,u)\ .$$

The missing part ??? shall be an expression that is allowed to use only the two subtrees t and u that up receives. Given $t = mapB\,(x:)\,(ch\,k\,xs)$ and $u = ch\,(1+k)\,xs$ (from (4.3)), this expression shall evaluate to the subexpression in (4.2) (let us call it (4.2.1)):

$$mapB\,(subs \circ (x:))\,(ch\,(1+k)\,xs)\ . \tag{4.2.1}$$

It may appear that, now that up already has $u = ch\,(1+k)\,xs$, the ??? may simply be $mapB\,(subs \circ (x:))\,u$. The problem is that the up does not know what x is—unless $k = 0$.

Case 3.1. $k = 0$. We can recover x from $mapB\,(x:)\,(ch\,0\,xs)$ if k happens to be 0 because:

$$\begin{aligned} & mapB\,(x:)\,(ch\,0\,xs) \\ = \; & mapB\,(x:)\,(\mathsf{T}\,[\,]) \\ = \; & \mathsf{T}\,[x]\ . \end{aligned}$$

That is, the left subtree up receives must have the form $\mathsf{T}\,[x]$, from which we can retrieve x and apply $mapB\,(subs \circ (x:))$ to the other subtree. We can furthermore simplify $mapB\,(subs \circ (x:))\,(ch\,(1+0)\,xs)$ a bit:

$$\begin{aligned} & mapB\,(subs \circ (x:))\,(ch\,(1+0)\,xs) \\ = \; & mapB\,(\lambda q \to [[x], q])\,(ch\,1\,xs)\ . \end{aligned}$$

The equality above holds because every tip in $ch\,1\,xs$ contains singleton lists and, for a singleton list $[z]$, we have $subs\,(x:[z]) = [[x],[z]]$. In summary, we have established

$$up\,(\mathsf{N}\,(\mathsf{T}\,p)\,u) = \mathsf{N}\,(mapB\,(\lambda q \to [p,q])\,u)\,(up\,u)\ .$$

Case 3.2. $0 < k$ (and $k < length\,xs - 1$). In this case, we have to construct (4.2.1), that is, $mapB\,(subs \circ (x:))\,(ch\,(1+k)\,xs)$, out of the two subtrees, $mapB\,(x:)\,(ch\,k\,xs)$ and $ch\,(1+k)\,xs$, without knowing what x is.

What follows is perhaps the most tricky part of the derivation. Starting from $mapB\,(subs \circ (x:))\,(ch\,(1+k)\,xs)$, we expect to use induction somewhere; therefore, a possible strategy is to move $mapB\,subs$ rightward, next to ch, in order to apply (4.1). Let us consider $mapB\,(subs \circ (x:))\,u$ for a general u, and try to move $mapB\,subs$ next to u.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

- By definition, $subs\ (x : xs) = map\ (x:)\ (subs\ xs) +\!\!+ [xs]$. That is, $subs \circ (x:) = \lambda xs \to snoc\ (map\ (x:)\ (subs\ xs))\ xs$—it duplicates the argument xs and applies $map\ (x:) \circ subs$ to one of them, before calling *snoc*.
- $mapB\ (subs \circ (x:))$ does the above to *each* list in the tree u.
- Equivalently, we may also duplicate each values in u to pairs, before applying $\lambda(xs, xs') \to snoc\ (map\ (x:)\ (subs\ xs))\ xs'$ to each pair.
- Values in u can be duplicated by zipping u to itself, that is, $zipBW\ pair\ u\ u$ where $pair\ xs\ xs' = (xs, xs')$.

With the idea above in mind, we calculate:

$$
\begin{array}{ll}
 & mapB\ (subs \circ (x:))\ u \\
= & \quad \{\ \text{definition of } subs\ \} \\
 & mapB\ (\lambda xs \to snoc\ (map\ (x:)\ (subs\ xs))\ xs)\ u \\
= & \quad \{\ \text{discussion above}\ \} \\
 & mapB\ (\lambda(xs, xs') \to snoc\ (map\ (x:)\ (subs\ xs))\ xs')\ (zipBW\ (\lambda xs\ xs' \to (xs, xs'))\ u\ u) \\
= & \quad \{\ zipBW \text{ natural}\ \} \\
 & zipBW\ snoc\ (mapB\ (map\ (x:) \circ subs)\ u)\ u \quad .
\end{array}
$$

We have shown that

$$mapB\ (subs \circ (x:))\ u = zipBW\ snoc\ (mapB\ (map\ (x:) \circ subs)\ u)\ u \quad , \tag{4.4}$$

which brings $mapB\ subs$ next to u. The naturality of $zipBW$ in the last step is the property that, provided that $h\ (f\ x\ y) = k\ (g\ x)\ (r\ y)$, we have:

$$mapB\ h\ (zipBW\ f\ t\ u) = zipBW\ k\ (mapB\ g\ t)\ (mapB\ r\ u) \quad .$$

Back to (4.2.1), we may then calculate:

$$
\begin{array}{ll}
 & mapB\ (subs \circ (x:))\ (ch\ (1 + k)\ xs) \\
= & \quad \{\ \text{by } (4.4)\ \} \\
 & zipBW\ snoc\ (mapB\ (map\ (x:) \circ subs)\ (ch\ (1 + k)\ xs))\ (ch\ (1 + k)\ xs) \\
= & \quad \{\ \text{induction}\ \} \\
 & zipBW\ snoc\ (mapB\ (map\ (x:))\ (up\ (ch\ k\ xs)))\ (ch\ (1 + k)\ xs) \\
= & \quad \{\ up \text{ natural}\ \} \\
 & zipBW\ snoc\ (up\ (mapB\ (x:)\ (ch\ k\ xs)))\ (ch\ (1 + k)\ xs) \quad .
\end{array}
$$

Recall that our aim is to find a suitable definition of up such that (4.2) equals (4.3). The calculation shows that we may let

$$up\ (\mathsf{N}\ t\ u) = \mathsf{N}\ (zipBW\ snoc\ (up\ t)\ u)\ (up\ u) \quad .$$

In summary, we have constructed:

$$
\begin{array}{l}
up :: \mathsf{B}\ a \to \mathsf{B}\ (\mathsf{L}\ a) \\
up\ (\mathsf{N}\ (\mathsf{T}\ p)\ (\mathsf{T}\ q)) = \mathsf{T}\ [p, q] \\
up\ (\mathsf{N}\ t\ \ \ \ (\mathsf{T}\ q)) = \mathsf{T}\ (snoc\ (unT\ (up\ t))\ q) \\
up\ (\mathsf{N}\ (\mathsf{T}\ p)\ u\ \ \ \ ) = \mathsf{N}\ (mapB\ (\lambda q \to [p, q])\ u)\ (up\ u) \\
up\ (\mathsf{N}\ t\ \ \ \ u\ \ \ \ ) = \mathsf{N}\ (zipBW\ snoc\ (up\ t)\ u)\ (up\ u) \quad ,
\end{array}
$$

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

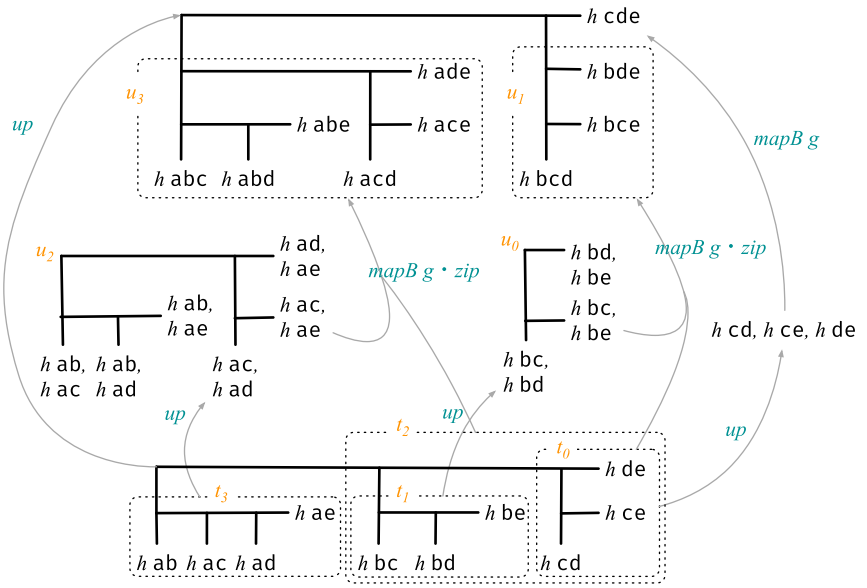


Fig. 4. Applying *mapB g* ∘ *up* to *mapB h* (*ch* 2 `abcde`). We abbreviate *zipBW snoc* to *zip*.

which is the mysterious four-line function in Bird (2008)! There is only one slight difference: where we use *snoc*, Bird used (:), which has an advantage of being more efficient. Had we specified *subs* and *choose* differently, we would have derived the (:) version, but either the proofs so far would have to proceed in terms of snoc lists, or the sublists in our examples would come in a less intuitive order. For clarity, we chose to present this version. For curious readers, code of the (:) version of *up* is given in Appendix A.

An Example. To demonstrate how *up* works, shown at the bottom of Figure 4 is the tree built by *mapB h* (*ch* 2 `abcde`). If we apply *up* to this tree, the fourth clause of *up* is matched, and we traverse along its right spine until reaching t_0, which matches the second clause of *up*, and a singleton tree containing [*h* `cd`, *h* `ce`, *h* `de`] is generated.

Traversing backward, *up* t_1 generates u_0, which shall have the same shape as t_0 and can be zipped together to form u_1. Similarly, *up* t_3 generates u_2, which shall have the same shape as t_2. Zipping them together, we get u_3. They constitute *mapB h* (*ch* 3 `abcde`), shown at the top of Figure 4.

4.3 Interlude: Shape constraints with dependent types

While the derivation guarantees that the function *up*, as defined above, satisfies (4.1), the partiality of *up* still makes one uneasy. Why is it that *up t* in the second clause always returns a T? What guarantees that *up t* and *u* in the last clause always have the same shape and can be zipped together? In this section, we try to gain more understanding of the tree construction with the help of dependent types.

Certainly, *ch* does not generate all trees of type B, but only those trees having certain shapes. We can talk about the shapes formally by annotating B with indices, as in the

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

following Agda datatype:

$$\begin{aligned}&\textbf{data}\ \mathsf{B}\ (a : \mathsf{Set}) : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{T}_0 : a \to \mathsf{B}\ a\ 0\ n\\ &\quad \mathsf{T}_n : a \to \mathsf{B}\ a\ (1+n)\ (1+n)\\ &\quad \mathsf{N}\ \ : \mathsf{B}\ a\ k\ n \to \mathsf{B}\ a\ (1+k)\ n \to \mathsf{B}\ a\ (1+k)\ (1+n)\ \ .\end{aligned}$$

The intention is that $\mathsf{B}\ a\ k\ n$ is the tree representing choosing k elements from a list of length n. Notice that the changes of indices in B follow the definition of *ch*. We now have two base cases, T_0 and T_n, corresponding to choosing 0 elements and all elements from a list. A tree $\mathsf{N}\ t\ u : \mathsf{B}\ a\ (1+k)\ (1+n)$ represents choosing $1+k$ elements from a list of length $1+n$, and the two ways to do so are $t : \mathsf{B}\ a\ k\ n$ (choosing k from n) and $u : \mathsf{B}\ a\ (1+k)\ n$ (choosing $1+k$ from n). With the definition, *ch* may have type

$$ch : (k : \mathbb{N}) \to \{n : \mathbb{N}\} \to k \leqslant n \to \mathsf{Vec}\ a\ n \to \mathsf{B}\ (\mathsf{Vec}\ a\ k)\ k\ n\ \ ,$$

where $\mathsf{Vec}\ a\ n$ denotes a list (vector) of length n.

One can see that a pair of (k, n) uniquely determines the shape of the tree. Furthermore, one can also prove that $\mathsf{B}\ a\ k\ n \to k \leqslant n$, that is, if a tree $\mathsf{B}\ a\ k\ n$ can be built at all, it must be the case that $k \leqslant n$.

The Agda implementation of *up* has the following type:

$$up : 0 < k \to k < n \to \mathsf{B}\ a\ k\ n \to \mathsf{B}\ (\mathsf{Vec}\ a\ (1+k))\ (1+k)\ n\ \ .$$

The type states that it is defined only for $0 < k < n$; the shape of its input tree is determined by (k, n); the output tree has shape determined by $(1+k, n)$, and the values in the tree are lists of length $1+k$. One can also see from the types of its components that, for example, the two trees given to *zipBW* always have the same shape. More details are given in Appendix B.

In **Case 2** of Section 4.2, we saw a function *unT*. Its dependently typed version has type $\mathsf{B}\ a\ (1+n)\ (1+n) \to a$. It always succeeds because a tree having type $\mathsf{B}\ a\ (1+n)\ (1+n)$ must be constructed by T_n—for $\mathsf{N}\ t\ u$ to have type $\mathsf{B}\ a\ (1+n)\ (1+n)$, u would have type $\mathsf{B}\ a\ (1+n)\ n$, an empty type.

Dependent types help us rest assured that the "partial" functions we use are actually safe. The current notations, however, are designed for interactive theorem proving, not program derivation. The author derives program on paper by equational reasoning in a more concise notation and double-checks the details by theorem prover afterward. All the proofs in this pearl have been translated to Agda. For the rest of the pearl, we switch back to non-dependently typed equational reasoning.

Pascal's Triangle. With so much discussion about choosing, it is perhaps not surprising that the sizes of subtrees along the right spine of a B tree correspond to prefixes of diagonals in Pascal's Triangle. After all, the k-th diagonal (counting from zero) in Pascal's Triangle denotes binomial coefficients—the numbers of ways to choose k elements from k, $k+1$, $k+2$... elements. This is probably why Bird (2008) calls the data structure a *binomial tree*, hence the name B.[2] See Figure 5 for example. The sizes along the right spine of *ch* 2 `abcde`, that is, 10, 6, 3, 1, is the second diagonal (in orange), while the right spine of *ch* 3 `abcde` is

[2] It is not directly related to the tree, having the same name, used in *binomial heaps*.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

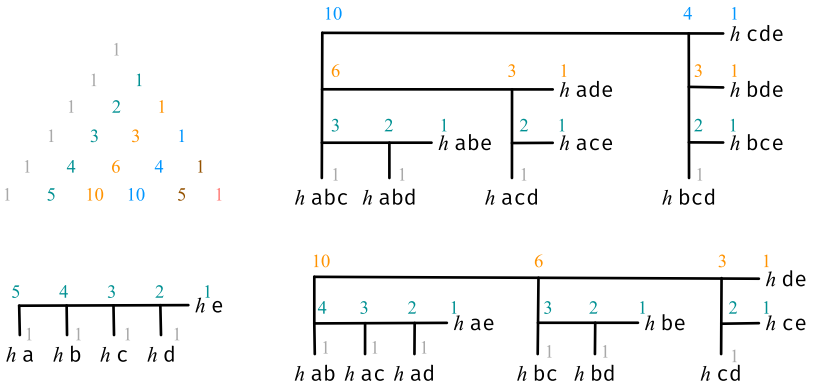


Fig. 5. Sizes of B alone the right spine correspond to prefixes of diagonals in Pascal's Triangle.

the fourth diagonal (in blue). Applying *up* to a tree moves it rightward and downward. In a sense, a B tree represents a prefix of a diagonal in Pascal's Triangle *with a proof* of how it is constructed.

5 The bottom-up algorithm

Now that we have constructed an *up* that satisfies (4.1), it is time to derive the main algorithm. Recall that we have defined, in Section 2, $h\,xs = td\,(length\,xs - 1)\,xs$, where

$$\begin{aligned} &td :: \mathbb{N} \to \mathsf{L}\,\mathsf{X} \to \mathsf{Y} \\ &td\,0 \quad\;\; = f \circ ex \\ &td\,(1+n) = g \circ map\,(td\,n) \circ subs \quad . \end{aligned}$$

The intention is that $td\,n$ is a function defined for inputs of length exactly $1+n$. We also define a variation:

$$\begin{aligned} &td' :: \mathbb{N} \to \mathsf{L}\,\mathsf{Y} \to \mathsf{Y} \\ &td'\,0 \quad\;\; = ex \\ &td'\,(1+n) = g \circ map\,(td'\,n) \circ subs \quad . \end{aligned}$$

The difference is that td' calls only *ex* in the base case. It takes only a routine induction to show that $td\,n = td'\,n \circ map\,f$. All the calls to f are thus factored to the beginning of the algorithm. We may then focus on transforming td'.

Note that for $ch\,n\,xs$ where $n = length\,xs$ always results in $\mathsf{T}\,xs$. That is, we have

$$unT\,(ch\,n\,xs) = xs, \text{ where } n = length\,xs. \tag{5.1}$$

Our main theorem is that

Theorem 1. *For all* $n :: \mathbb{N}$ *we have* $td\,n = bu\,n$, *where*

$$bu\,n = unT \circ (mapB\,g \circ up)^n \circ mapB\,ex \circ ch\,1 \circ map\,f \quad .$$

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

That is, the top-down algorithm $td\ n$ is equivalent to a bottom-up algorithm $bu\ n$, where the input is preprocessed by $mapB\ ex \circ ch\ 1 \circ map\ f$, followed by n steps of $mapB\ g \circ up$. By then we will have a singleton tree, whose content can be extracted by unT.

Proof Let $length\ xs = 1 + n$. We reason:

$$
\begin{array}{ll}
 & td\ n\ xs \\
= & \quad \{ \text{ since } td\ n = td'\ n \circ map\ f \ \} \\
 & (td'\ n \circ map\ f)\ xs \\
= & \quad \{ \text{ by } (5.1) \ \} \\
 & (td'\ n \circ unT \circ ch\ (1+n) \circ map\ f)\ xs \\
= & \quad \{ \text{ naturality of } unT \ \} \\
 & (unT \circ mapB\ (td'\ n) \circ ch\ (1+n) \circ map\ f)\ xs \\
= & \quad \{ \text{ Lemma 1 } \} \\
 & (unT \circ (mapB\ g \circ up)^n \circ mapB\ ex \circ ch\ 1 \circ map\ f)\ xs \\
= & \quad \{ \text{ definition of } bu \ \} \\
 & bu\ n\ xs \ .
\end{array}
$$

■

The real work is done in Lemma 1 below. It shows that $mapB\ (td'\ n) \circ ch\ (1+n)$ can be performed by processing the input by $mapB\ ex \circ ch\ 1$, before n steps of $mapB\ g \circ up$. This is the key lemma that relates (4.1) to the main algorithm.

Lemma 1. *For inputs of length* $1 + n$ $(n > 1)$*, we have*

$$mapB\ (td'\ n) \circ ch\ (1+n) = (mapB\ g \circ up)^n \circ mapB\ ex \circ ch\ 1 \ .$$

Proof For $n := 0$ both sides simplify to $mapB\ ex \circ ch\ 1$. For $n := 1 + n$, we start from the LHS, assuming an input of length $2 + n$:

$$
\begin{array}{ll}
 & mapB\ (td'\ (1+n)) \circ ch\ (2+n) \\
= & \quad \{ \text{ def. of } td' \ \} \\
 & mapB\ (g \circ map\ (td'\ n) \circ subs) \circ ch\ (2+n) \\
= & \quad \{ \text{ by } (4.1) \ \} \\
 & mapB\ (g \circ map\ (td'\ n)) \circ up \circ ch\ (1+n) \\
= & \quad \{ up \text{ natural } \} \\
 & mapB\ g \circ up \circ mapB\ (td'\ n) \circ ch\ (1+n) \\
= & \quad \{ \text{ induction } \} \\
 & mapB\ g \circ up \circ (mapB\ g \circ up)^n \circ mapB\ ex \circ ch\ 1 \\
= & \quad \{ (\circ) \text{ associative, def. of } f^n \ \} \\
 & (mapB\ g \circ up)^{1+n} \circ mapB\ ex \circ ch\ 1 \ .
\end{array}
$$

■

6 Conclusion and discussions

We have derived the mysterious four-line function of Bird (2008) and built upon it a bottom-up algorithm that solves the sublists problem. The specifications (3.1) and (4.1)

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

may look trivial in retrospect, but it did took the author a lot of efforts to discover them. In typical program calculation, one starts with a specification of the form $up\ t = rhs$, where t, the argument to the function to be derived, is a variable. One then tries to pattern match on t, simplifies rhs, and finds an inductive definition of up. The author has tried to find such a specification with no avail, before settling down on (3.1) and (4.1), where up is applied to a function call. Once (4.1) is found, the rest of the development is tricky at times, but possible. The real difficulty is that when we get stuck in the calculation, we may hardly know which is the major source of the failure—the insufficiency of the specification, or the lack of a clever trick to bridge the gap. Techniques for performing such calculations to find a solution for up is one of the lessons the author learned from this experience.

Some final notes on the previous works. The sublists problem was one of the examples of Bird & Hinze (2003), a study of memoization of functions, with a twist: the memo table is structured according to the call graph of the function, using trees of shared nodes (which they called *nexuses*). To solve the sublists problem, Bird & Hinze (2003) introduced a data structure, also called a "binomial tree". Whereas the binomial tree in Bird (2008) and in this pearl models the structure of the function *choose*, that in Bird & Hinze (2003) can be said to model the function computing *all* sublists:

$$\begin{aligned} &sublists\ [\,] &&= [[\,]] \\ &sublists\ (x : xs) &&= map\ (x:)\ (sublists\ xs) +\!\!+ sublists\ xs \quad . \end{aligned}$$

Such trees were then extended with up links (and became *nexuses*). Trees were built in a top-down manner, creating carefully maintained links going up and down.

Bird then went on to study the relationship between top-down and bottom-up algorithms, and the sublists problem was one of the examples in Bird (2008) to be solved bottom-up. In Bird's formulation, the function used in the top-down algorithm that decomposes problems into sub-problems (like our *subs*) is called *dc*, with type $\mathsf{L}\ a \to \mathsf{F}\ (\mathsf{L}\ a)$ for some functor F. The bottom-up algorithm uses a function *cd* (like our *upgrade*), with type $\mathsf{L}\ a \to \mathsf{L}\ (\mathsf{F}\ a)$. One of the properties they should satisfy is $dc \circ cd = mapF\ cd \circ dc$, where *mapF* is the functorial mapping for F.

For the sublists problem, $dc = subs$, and $\mathsf{F} = \mathsf{L}$. We know parts of the rest of the story: Bird had to introduce a new datatype B, and a new *cd* with type $\mathsf{B}\ a \to \mathsf{B}\ (\mathsf{L}\ a)$, the four-line function that inspired this pearl. That was not all, however. Bird also quickly introduced, on the last page of the paper, a new $dc' :: \mathsf{B}\ a \to \mathsf{L}\ (\mathsf{B}\ a)$, which was as cryptic as *cd*, and claimed that $dc' \circ cd = mapF\ cd \circ dc'$. The relationship between dc' and dc (and the original problem) was not clear. In this pearl, we took a shorter route, proving directly that the bottom-up algorithm works, and the step function is $mapB\ g \circ up$.

Acknowledgments

The author would like to thank Hsiang-Shang Ko and Jeremy Gibbons for many in-depth discussions throughout the development of this work, Conor McBride for discussion at a WG 2.1 meeting, and Yu-Hsuan Wu and Chung-Yu Cheng for proof-reading drafts of this pearl. The examples of how the immediate sublists problem may be put to work was suggested by Meng-Tsung Tsai. The first version of the Agda proofs was constructed by You-Zheng Yu.

Conflicts of interest

None.

Supplementary material

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796824000145.

References

Bird, R. S. (2008) Zippy tabulations of recursive functions. In *Mathematics of Program Construction*. Springer-Verlag. pp. 92–109.

Bird, R. S. & Hinze, R. (2003) Functional pearl: Trouble shared is trouble halved. In Haskell Workshop. Academic Press, pp. 1–6.

Tao, T. & Vu, V. H. (2012) *Additive Combinatorics*. Cambridge University Press.

A Variation of *up* That Uses (:)

Show below is the variation of *up* that uses (:). The function is called *cd* in Bird (2008).

$$
\begin{aligned}
&up :: \mathsf{B}\ a \to \mathsf{B}\ (\mathsf{L}\ a)\\
&up\ (\mathsf{N}\ (\mathsf{T}\ p)\ (\mathsf{T}\ q)) = \mathsf{T}\ [p, q]\\
&up\ (\mathsf{N}\ t\ \ \ \ (\mathsf{T}\ q)) = \mathsf{N}\ (up\ t)\ (mapB\ (:[q])\ t)\\
&up\ (\mathsf{N}\ (\mathsf{T}\ p)\ u\ \ \ \ ) = \mathsf{T}\ (p : unT\ (up\ u))\\
&up\ (\mathsf{N}\ t\ \ \ \ u\ \ \ \ ) = \mathsf{N}\ (up\ t)\ (zipBW\ (:)\ t\ (up\ u))
\end{aligned}
$$

B Agda Implementation of *up*

The following is an Agda implementation of *up*. The type states that it is defined only for $0 < k < n$; the shape of its input tree is determined by (k, n); the output tree has shape determined by $(1 + k, n)$, and the values in the tree are lists of length $1 + k$.

```
up : (0 < k) → (k < n) → B a k n → B (Vec a (1 + k)) (1 + k) n
up 0<0 _       (T₀ x)          = ⊥-elim (<-irefl refl 0<0)
up _ 1+n<1+n   (Tₙ x)          = ⊥-elim (<-irefl refl 1+n<1+n)
up _ 2+n<2+n   (N (Tₙ _) _)    = ⊥-elim (<-irefl refl 2+n<2+n)
up _ _         (N (T₀ p)  (Tₙ q)       ) = Tₙ (p :: q :: [ ])
up _ _         (N t@(N _ _) (Tₙ q)       ) = Tₙ (snoc (unTₙ (up (s≤s z≤n) (s≤s ≤-refl) t)) q)
up _ _         (N (T₀ p)  u@(N _ u′)) = N (mapB (λq → p :: q :: [ ]) u)
                                          (up ≤-refl (s≤s (bounded u′)) u)
up _ 2+k<2+n   (N t@(N _ _) u@(N _ u′)) = N (zipBW snoc (up (s≤s z≤n) (s≤s⁻¹ 2+k<2+n) t) u)
                                          (up (s≤s z≤n) (s≤s (bounded u′)) u)
```

The first three clauses of *up* eliminate impossible cases. The remaining four clauses are essentially the same as in the non-dependently typed version, modulo the additional

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press

arguments and proof terms, shown in light brown, that are needed to prove that k and n are within bounds. In the clause that uses unT, the input tree has the form $\mathsf{N}\, t\, (\mathsf{T}_n\, q)$. The right subtree being a T_n forces the other subtree t to have type $\mathsf{B}\, a\, (1+k)\, (2+k)$—the two indices must differ by 1. Therefore, $up\, t$ has type $\mathsf{B}\, a\, (2+k)\, (2+k)$ and must be built by T_n. The last clause receives inputs having type $\mathsf{B}\, a\, (2+k)\, (2+n)$. Both u and $up\, t$ have types $\mathsf{B}\, \ldots\, (2+k)\, (1+n)$ and, therefore, have the same shape.

https://doi.org/10.1017/S0956796824000145 Published online by Cambridge University Press