Binomial Tabulation: A Short Story

HSIANG-SHANG KO and SHIN-CHENG MU, Academia Sinica, Taiwan
JEREMY GIBBONS, University of Oxford, UK

We reconstruct some of the development in Richard Bird's [2008] paper *Zippy Tabulations of Recursive Functions*, using dependent types and string diagrams rather than mere simple types. This paper serves as an intuitive introduction to and demonstration of these concepts for the curious functional programmer, who ideally already has some exposure to dependent types and category theory, is not put off by basic concepts like indexed types and functors, and wants to see a more practical example.

The paper is presented in the form of a short story, narrated from the perspective of a functional programmer trying to follow the development in Bird's paper. The first section recaps the original simply typed presentation. The second section explores a series of refinements that can be made using dependent types. The third section uses string diagrams to simplify arguments involving functors and naturality. The short story ends there, but the paper concludes with a discussion and reflection in the afterword.

1 FUNCTIONS

'What on earth is this function doing?'

I stare at the late Richard Bird's [2008] paper *Zippy Tabulations of Recursive Functions*, frowning.

```
cd :: B a -> B (L a)
cd (Bin (Tip y) (Tip z)) = Tip [y,z]
cd (Bin (Tip y) u      ) = Tip (y : ys) where Tip ys = cd u
cd (Bin t       (Tip z)) = Bin (cd t) (mapB (: [z]) t)
cd (Bin t       u      ) = Bin (cd t) (zipBWith (:) t (cd u))
```

I know `B` is this Haskell data type of binary trees:

```
data B a = Tip a | Bin (B a) (B a)
```

Presumably `mapB` and `zipBWith` are the usual map and zip functions for these trees, and `L` is the standard list type. But how did Richard come up with such an incomprehensible function definition? He didn't bother to explain it in his paper. It might have helped if he had provided some examples.

1.1 Top-Down and Bottom-Up Algorithms

From the explanations that *are* in the paper, I suppose I can guess roughly what `cd` should do. Richard was studying the relationship between top-down and bottom-up algorithms that solve problems specified recursively on some input data structure. When the input is a nonempty list, a generic top-down algorithm is defined by

```
td :: (a -> s) -> (L s -> s) -> L a -> s
td f g [x] = f x
td f g xs  = g (map (td f g) (dc xs))
```

The input of `td` is a nonempty list of `a`'s, and the output is a solution of type `s` for the input list. Singleton lists form the base case, for which the given function `f` computes a solution directly. If an input list `xs` is not singleton, it is decomposed into shorter lists by some `dc :: L a -> L (L a)`. Then `td` recursively computes a subsolution for each shorter list. Finally, `g` combines the subsolutions

Authors' addresses: Hsiang-Shang Ko, joshko@iis.sinica.edu.tw; Shin-Cheng Mu, scm@iis.sinica.edu.tw, Academia Sinica, Institute of Information Science, 128 Academia Road, Taipei, Taiwan, 115201; Jeremy Gibbons, jeremy.gibbons@cs.ox.ac.uk, University of Oxford, Department of Computer Science, Wolfson Building, Parks Road, Oxford, UK, OX1 3QD.

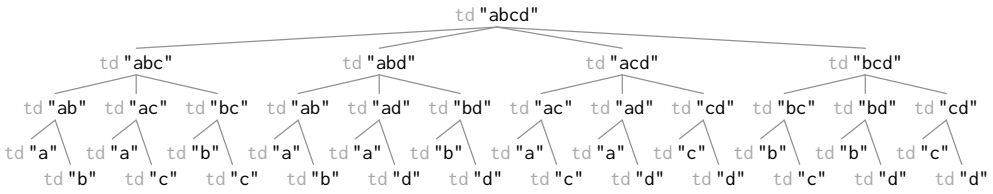


Fig. 1. Computing `td "abcd"` top-down.

into a solution for `xs`. In fact, in order to cover a wider range of algorithms, Richard's definition is more abstract and general. But I'm working with this simplified version as an easier starting point.

In the last example of the paper, `dc` computes all the *immediate sublists* of a given list — all the lists with exactly one element missing:

```
dc :: L a -> L (L a)
dc [x,y]       = [[x],[y]]
dc (xs ++ [x]) = [xs] ++ [ys ++ [x] | ys <- dc xs]
```

(Richard assumed that a list could be matched with a snoc pattern `xs ++ [x]`.) For example, computing a solution for `"abcd"` requires subsolutions for `"abc"`,`"abd"`, `"acd"`, and `"bcd"` (Figure 1). In turn, computing a solution for `"abc"` requires subsolutions for `"ab"`, `"ac"`, and `"bc"`, and so on. When the problem decomposition reaches length-2 sublists — that's a bit of a mouthful, so let me just say '2-sublists' for short — it becomes evident that this `dc` leads to *overlapping subproblems*, and `td` deals with that very inefficiently. For example, a solution for `"ab"` is required for solving the problem for `"abc"` and `"abd"`, and `td` computes that solution twice. And it gets worse further down: a solution for each 1-sublist is computed 6 times!

It's better to proceed bottom-up instead, working upwards through a lattice of sublists (Figure 2), level by level. Level 1 consists of solutions for the 1-sublists. Then solutions for the $(k + 1)$-sublists in level $k + 1$ are computed from subsolutions in level k. Finally, the top level consists of a single solution, for the input list. More specifically, if the levels were represented as lists, level 2 would be

```
[td "ab", td "ac", td "bc", td "ad" ...]
```

One way to construct level 3 from level 2 would be using a function `cd' :: L a -> L (L a)` that copies and rearranges the elements such that the subsolutions for the immediate sublists of the same list are brought together:

```
[[td "ab", td "ac", td "bc"], [td "ab", td "ad", td "bd"] ...]
```

Then applying `map g` to this list of lists would produce level 3:

```
[td "abc", td "abd", td "acd", td "bcd" ...]
```

If such a function `cd'` could be constructed, a bottom-up algorithm computing the same solution as `td` would be given by

```
bu' :: (a -> s) -> (L s -> s) -> L a -> s
bu' f g = head . loop . map f
  where loop [y] = [y]
        loop ys  = loop (map g (cd' ys))
```

Level 1 is obtained by applying `f` to each element of the input list. Then the `loop` function keeps applying `map g . cd'` to get the next level. It stops at a level with a single element, which is a

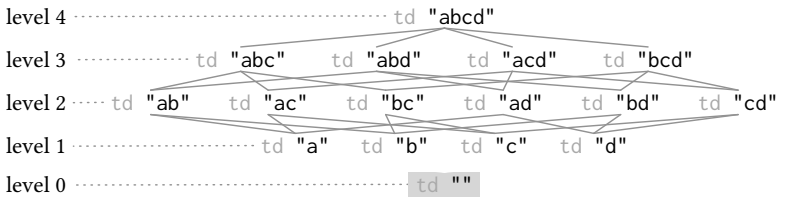


Fig. 2. Computing `td "abcd"` bottom-up.

solution for the entire input list. Like `td`, this `bu'` is a simplified version. To cope with more general problems, Richard had to store something more complex in each level, but I don't think I need that.

1.2 Rearranging Binary Trees

That's what I understand about `cd` so far. But Richard must have realised at some point that it's difficult to define the `cd` rearrangement using lists, and decided to represent each level using the `B` data type. So `cd :: B a -> B (L a)`, and the actual bottom-up algorithm `bu` is defined by

```
bu :: (a -> s) -> (L s -> s) -> L a -> s
bu f g = unTip . loop . cvt . map f
  where loop (Tip x) = Tip x
        loop xs      = loop (mapB g (cd xs))
```

where `unTip` extracts the element of a `Tip` tree, and `cvt` converts a list to a tree:

```
unTip :: B a -> a
unTip (Tip x) = x
```

```
cvt :: L a -> B a
cvt [x]         = Tip x
cvt (xs ++ [x]) = Bin (cvt xs) (Tip x)
```

I wonder if I have to use `B` instead of lists in `cd`. If I'm given level 1 of the lattice (Figure 2) as a 4-list, I know they are solutions for the four 1-sublists, and surely there's no problem rearranging them into level 2…?

Oh wait, I don't actually know. All I get is a 4-list. This 4-list could be level 1, but it could well be level 3. And I don't get any information from the elements — the element type is parametrically quantified. So there isn't enough context for me to decide whether I should produce a 6-list (level 2) or a 1-list (level 4).

Presumably, Richard's trees gave him more context. I try to trace Richard's `cd` to find out how it works. Given input `"abcd"`, the function `cvt . map f` yields a tree slanted to the left as level 1:

```
Bin (Bin (Bin (Tip (td "a")) (Tip (td "b"))) (Tip (td "c"))) (Tip (td "d"))
```

Following Richard's convention, I draw a `Tip x` as `x`, and `Bin t u` as a dot with `t` to its left and `u` below (Figure 3). Applying `mapB g . cd` to this, I get level 2. For a closer look, I apply only `cd` to level 2. Indeed, with its clever mapping and zipping, `cd` manages to bring together precisely the right elements, and produces a 'level $2\frac{1}{2}$'. Then I reach level 3 by applying `mapB g`. There's indeed more context for distinguishing levels 1 and 3: their tree representations have the same size but different shapes.

The intuition about having enough context seems useful. I was puzzled by why Richard started from singleton lists instead of the empty list. The intuition helps to explain that too. Level 0 would be a singleton list/tree, and I wouldn't know the number of values level 1 should contain. That

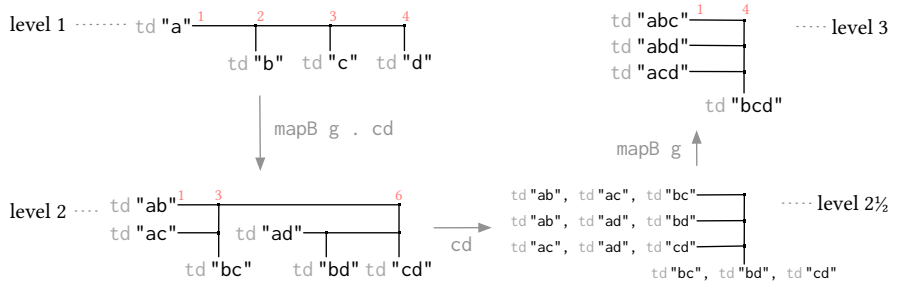


Fig. 3. How `mapB g . cd` constructs a new level.

number is the length of the input list, and `cd` doesn't get that information. So there isn't enough context for going from level 0 to level 1, regardless of how levels are represented.

1.3 Rearranging Binomial Trees

That still doesn't give me much insight into why `cd` works though. Presumably, `cd` does something useful only for the trees built by `cvt . map f` and `cd` itself. What are the constraints on these trees, and how does `cd` exploit them?

Richard did give a hint: the sizes $[1, 2, 3, 4]$, $[1, 3, 6]$, and $[1, 4]$ of subtrees along their left spines (the red numbers in Figure 3) are the diagonals of Pascal's triangle — the trees are related to *binomial coefficients*! The binomial coefficient C_k^n is the number of ways of choosing k elements from an n-list. Indeed, each level k in the lattice (Figure 2) contains values about k-sublists. For example, level 2 has $C_2^4 = 6$ values, and there are 6 ways of choosing 2 elements from a 4-list.

Aha! I can even see a pattern related to the choices in the tree representation of level 2 (Figure 3): the right subtree is about all the 2-sublists that end with `'d'`, and the left subtree about the other 2-sublists not containing `'d'`. To choose 2 elements from `"abcd"`, I can include the rightmost element `'d'` or not. If `'d'` is included, there are C_1^3 ways of choosing 1 element from `"abc"` to go with `'d'`. If `'d'` is not included, there are C_2^3 ways of choosing 2 elements from `"abc"`. And the total number of 2-sublists is $C_2^3 + C_1^3 = C_2^4$. All the `Bin` nodes fit this pattern. I guess the trees are supposed to satisfy a binomial shape constraint (and the name of the `B` data type could refer to 'binomial' as much as to 'binary').

That's about as many clues as I can get from Richard's paper for now. Given these clues, how do I prove that `cd` indeed does the job — bringing values about related immediate sublists together? In fact, how do I even write that down as a formal specification? And how does that help me to prove that `td` equals `bu`?

I'm worried that there will be many complex proofs waiting ahead for me.

2 TYPES

2.1 Shapes in Types

The binomial shape constraint seems to be the easiest next target, because there's a standard solution: capturing the tree shape in its type. Shape-indexed data types are so common now that I'm tempted to say they should count as simple types. They're common even in Haskell. I prefer a proper dependently typed language though, so I open my favourite editor, and switch to Agda.

The classic example of shape-indexing is, of course, length-indexed lists (or 'vectors'):

```
data Vec : ℕ → Set → Set where
  []  :                      Vec zero    a
  _::_ : a → Vec n a → Vec (suc n) a
```

(I enjoy writing Agda types these days because I no longer have to quantify over each and every variable, such as a and n in the type of cons — Agda supports implicit 'generalisation of declared variables' now.) The constructors used in a list of type Vec n a are completely determined by the length index n. The data type definition could even be understood as if it were performing pattern matching on the index: if the index is **zero**, then the list has to be nil; otherwise the index is a **suc**, and the list has to start with a cons. (Chapman et al. [2010] did develop a theory where data types can be defined by pattern matching on indices in this way.)

In the same vein, I write down a shape-indexed version of Richard's B data type (Section 1.3):

```
data B : ℕ → ℕ → Set → Set where
  tip_z : a                          → B n          zero    a
  tip_s : a                          → B (suc k) (suc k) a
  bin   : B n (suc k) a → B n k a → B (suc n) (suc k) a
```

The size of a tree of type B n k a with $k \leq n$ is precisely the binomial coefficient C^n_k. Naturally, there are no inhabitants when $k > n$. Like Vec, the indices n, k determine the constructors. If k is **zero**, then the tree is a $\mathbf{tip_z}$ with one element ($C^n_0 = 1$). If n is **suc** k, then the tree is a $\mathbf{tip_s}$ also with one element ($C^{1+k}_{1+k} = 1$). Otherwise the tree is a **bin**, and the sizes C^n_{1+k} and C^n_k of the two subtrees add up to the expected size C^{1+n}_{1+k} of the whole tree. The trees are now truly *binomial* rather than just binary, formalising Richard's hint about sizes. I'll write B^n_k for B n k, by analogy with C^n_k.

And now I can give a more informative type to cd (Section 1):

```
cd : 1 ≤ k → k < n → B^n_k a → B^n_{1+k} (Vec (1 + k) a)
```

It takes as input the data for level k out of n levels in the sublist lattice (Figure 2), with $1 \leq k < n$; these are the solutions for each of the C^n_k k-sublists of the original n-list. And it returns as output the components for level $1 + k$; there are C^n_{1+k} of these, each a $(1 + k)$-list (to be fed into g when used in bu). There are two (greyed out) 'side conditions' $1 \leq k$ and $k < n$, which I don't want to bother with when I'm thinking at a higher level. I'm going to check these conditions mentally and then ignore everything related to these conditions — I'll call cd as if it were a function without the first two arguments, and skip over cases that are impossible due to the conditions. Agda ensures that I don't forget about all the ignored stuff in the final code though.

I continue to transcribe the definition of cd interactively in Agda.

```
cd : 1 ≤ k → k < n → B^n_k a → B^n_{1+k} (Vec (1 + k) a)
cd (bin     (tip_s y)       (tip_z z)  ) = tip_s (y :: z :: [])
cd (bin     (tip_s y)   u@(bin _ _))     = tip_s (y :: unTip_B (cd u))
cd (bin t@(bin _ _)         (tip_z z)  ) = bin (cd t) (map_B (_:: (z :: [])) t)
cd (bin t@(bin _ _) u@(bin _ _))         = bin (cd t) (zip_BWith _::_ t (cd u))
```

In the first case, **bin** ($\mathbf{tip_s}$ y) ($\mathbf{tip_z}$ z), Agda conveniently figures out for me whether a Tip in the pattern should be a $\mathbf{tip_s}$ or $\mathbf{tip_z}$. I expect Agda to fill in the right constructor for the goal type B^2_2 (Vec 2 a) too, but Agda complains that it cannot decide between $\mathbf{tip_s}$ and **bin**. Indeed, B^2_2 (Vec 2 a) matches the result type of both $\mathbf{tip_s}$ and **bin**. But **bin** is actually impossible because its left subtree would have type B^{1+k}_{2+k} a, and this type is uninhabited. So the indices of B still determine the constructor, though not as directly as in the case of Vec.

I go through the cases **bin** ($\mathbf{tip_s}\ y$) u and **bin** t ($\mathbf{tip_z}\ z$) without difficulties, after supplying the definitions of $\mathsf{unTip_B} : \mathsf{B}^n_n\ a \to a$ and $\mathsf{map_B} : (a \to b) \to \mathsf{B}^n_k\ a \to \mathsf{B}^n_k\ b$. In the final **bin** t u case, the result should be a **bin**, and the left subtree `cd` t is accepted by Agda. Slightly anxiously, I start constructing the right subtree. Agda tells me that $t : \mathsf{B}^{1+n}_{2+k}\ a$ and $u : \mathsf{B}^{1+n}_{1+k}\ a$. When I ask what type `cd` u has, Agda responds with $\mathsf{B}^{1+n}_{2+k}\ (\mathsf{Vec}\ (2 + k)\ a)$. That's the same shape as t, so t and `cd` u can be safely zipped together using a shape-preserving $\mathsf{zip_BWith} : (a \to b \to c) \to \mathsf{B}^n_k\ a \to \mathsf{B}^n_k\ b \to \mathsf{B}^n_k\ c$.

I've gone through the whole definition! So, as I guessed, the binomial shape constraint holds throughout `cd`. What's nice is that I didn't need to do much. The type checker took care of most of the proof, using the indices to keep track of the tree shapes throughout the transcription.

2.2 Properties in Types

So much for the shape. But how do I know that the contents are correct — that `cd` is correctly rearranging the values?

The input to `cd` is a tree of values associated with k-sublists (for some $1 \le k < n$), and these values are rearranged in relation to $(1 + k)$-sublists. These values can have any type `a`, which is *parametrically* quantified in the type of `cd`. The parametric quantification ensures that `cd` cannot do different things to different `a`'s (because there's no way to do a case analysis on `a`). So if I can specify what `cd` should do for a specific choice of `a`, then `cd` will have to do the same thing for all `a`'s. Well, I suppose a natural choice is k-sublists themselves. I'll define Richard's trees (Figure 3) but just put k-sublists in them, and then specify that `cd` should transform level k to level $k\frac{1}{2}$.

In Haskell, I suppose Richard might have defined the tree of `k`-sublists of a list `xs` as `choose k xs`, where the function `choose` is defined by

```
choose :: Int -> L a -> B (L a)
choose    0  _                    = Tip []
choose (k+1) xs | length xs == k+1 = Tip xs
choose (k+1) (xs ++ [x])           = Bin (choose (k+1) xs)
                                         (mapB (++[x]) (choose k xs))
```

The pattern-matching structure of `choose` is the same as how B^n_k analyses its indices (Section 2.1), except that here I'm working with a list rather than just its length. The function generalises Richard's `dc` that computes the immediate sublists (Section 1.1): `dc xs` can be redefined as `flatten (choose (length xs - 1) xs)`, where

```
flatten :: B a -> L a
flatten (Tip x)   = [x]
flatten (Bin t u) = flatten t ++ flatten u
```

Then Richard could have specified `cd` by

```
cd (choose k xs) = mapB (flatten . choose k) (choose (k+1) xs)        (*)
```

Informally: given all the `k`-sublists of an `n`-list `xs`, `cd` should rearrange and duplicate them into the appropriate positions for the `(k+1)`-sublists, where in the position for a particular `(k+1)`-sublist, the result should be the list of all its immediate sublists as computed by `flatten . choose k`.

I could finish what Richard might have done in his paper by deriving the definition of `cd` from the specification (*). Maybe I could switch to B^n_k throughout to make the shapes clear. But there's going to be a large amount of tedious equational reasoning… I'm not thrilled by the prospect.

* * *

My editor is still running Agda and showing the shape-indexed version of cd, with B_k^n in its type (Section 2.1). The whole point of programming with *inductive families* [Dybjer 1994] such as B_k^n is to 'say more and prove less': encode more properties in the indices, so that those properties are automatically taken care of as programs construct and deconstruct indexed data, requiring fewer manual proofs. Instead of just the shapes, maybe it's possible to extend B_k^n and encode the *entire* specification (∗) of cd in its type?

What the specification (∗) says is basically that cd should transform a tree produced by choose into another one also produced by choose and then processed using a mapB. I suppose I could force a tree to be the result of `mapB h (choose k xs)` for some h (which is the common form of the input and output trees of cd) by adding h and xs as indices to B_k^n, and imposing equality constraints on the tree elements:

```
data B′ : (n k : ℕ) (b : Set) → (Vec k a → b) → Vec n a → Set where
  tipz : (y : b) (e : y ≡ h [] )                    → B′ n       zero   b h xs
  tips : (y : b) (e : y ≡ h xs)                     → B′ (suc k) (suc k) b h xs
  bin  : B′ n (suc k) b h xs → B′ n k b (h · (x ::_)) xs → B′ (suc n) (suc k) b h (x :: xs)
```

It's a rather complex extension of B_k^n, but I think I'll be fine if I stick to the same programming methodology: perform 'pattern matching' on the indices, and specify what should be in the tree in each case. In the first case, choose returns `Tip []`, so the tree should be a $\mathbf{tip_z}$, and its element y should be accompanied by a proof e that y equals h [], so that $\mathbf{tip_z}\ y\ e$ 'is' `mapB h (Tip [])`. The second case is similar. In the third case, the first thing I do is switch from Richard's snoc pattern `xs ++ [x]` to a cons index $x :: xs$ — this just 'reverses' the list and shouldn't break anything, as long as the snoc in `mapB (++[x])` is also switched to a cons to match. The first inductive call of choose easily translates into the type of the left subtree. The right subtree should be the result of `map h (map (x:) (choose k xs))`. Luckily, the two maps can be fused into a single `map (h . (x:))`. So the type of the second subtree uses the index $h \cdot (x ::\_)$ instead of h.

Even though I sort of derived B′ from a specification and know it should work, I still feel an urge to see with my own eyes that B′ does work as intended. So I write a tree with 'holes' (missing parts of a program) at the element positions, and let Agda tell me what types the holes should have:

```
testB′ : {b : Set} {h : Vec 2 Char → b} → B′ 4 2 b h "abcd"
testB′ = bin (bin (tips {b}₀ {?₀ ≡ h "cd"}₁)
                  (bin (tips {b}₂ {?₂ ≡ h "bd"}₃) (tipz {b}₄ {?₄ ≡ h "bc"}₅)))
             (bin (bin (tips {b}₆ {?₆ ≡ h "ad"}₇) (tipz {b}₈ {?₈ ≡ h "ac"}₉))
                  (tipz {b}₁₀ {?₁₀ ≡ h "ab"}₁₁))
```

The goal types of the even-numbered holes are all b, and the odd-numbered holes require proofs that the even-numbered holes are equal to $h\ zs$ for all the 2-sublists zs of "abcd". It works!

* * *

B′ doesn't look bad, but I can't help raising an eyebrow. With yet more effort, I suppose I could refine the type of cd to use B′ and encode the full specification (∗). But the refined cd would need to manipulate the equality proofs in those trees, and maybe eventually I'd still be doing essentially the same tedious equational reasoning that I wanted to avoid.

Another problem is that the upgraded cd would only work on trees of sublists, whereas the original cd in Haskell works on trees of *any* type of values. Indeed, the specification (∗) talks about the behaviour of cd on trees of sublists only. By encoding the specification in the type, I'd actually restrict cd to trees of sublists. That doesn't sound too useful.

Still, I can't take my eyes off the definition of B′. The way it absorbs the definition of choose looks right. If only the elements weren't restricted to pairs of the form $y : b$ and $e : y \equiv h\ zs$…

A lightbulb lights up above my head.

$$\begin{array}{l}\textbf{data}\ \mathsf{BT} : (n\ k : \mathbb{N}) \to (\mathsf{Vec}\ k\ a \to \mathsf{Set}) \to \mathsf{Vec}\ n\ a \to \mathsf{Set}\ \textbf{where}\\ \quad \textbf{tip}_\textbf{z} : p\ [] \to \mathsf{BT}\ n\ \ \textbf{zero}\ \ p\ xs\\ \quad \textbf{tip}_\textbf{s} : p\ xs \to \mathsf{BT}\ (\textbf{suc}\ k)\ (\textbf{suc}\ k)\ p\ xs\\ \quad \textbf{bin} : \mathsf{BT}\ n\ (\textbf{suc}\ k)\ p\ xs \to \mathsf{BT}\ n\ k\ (p \cdot (x ::\_))\ xs \to \mathsf{BT}\ (\textbf{suc}\ n)\ (\textbf{suc}\ k)\ p\ (x :: xs)\end{array}$$

Just generalise the element type! More specifically, generalise that to a *type family* $p : \mathsf{Vec}\ k\ a \to \mathsf{Set}$ indexed by k-sublists. Then $\mathsf{B}^n_k\ a$ becomes a special case by specialising p to const a (and supplying any n-list as the last index), and similarly $\mathsf{B}'^n_k\ b\ h\ xs$ by specialising p to $\lambda\ zs \to \Sigma[\ y \in b\ ]\ y \equiv h\ zs$!

I wasn't expecting this generalisation. After taking a moment to calm down, I look more closely at this new, unifying data type. The index p in BT replaces h in B′, and is similarly applied to all the sublists of a particular length. What's different is that p is applied to a sublist to get the whole element type in a tip. So, in general, all the elements in a tree of type $\mathsf{BT}^n_k\ p\ xs$ have *distinct* types, which are $p\ ys$ for all the k-sublists ys of the n-list xs. To see an example:

$$\begin{array}{l}\mathsf{testBT} : \{p : \mathsf{Vec}\ 2\ \mathsf{Char} \to \mathsf{Set}\} \to \mathsf{BT}^4_2\ p\ \texttt{"abcd"}\\ \mathsf{testBT} = \textbf{bin}\ (\textbf{bin}\ (\textbf{tip}_\textbf{s}\ \{p\ \texttt{"cd"}\}_\mathbf{0})\ (\textbf{bin}\ (\textbf{tip}_\textbf{s}\ \{p\ \texttt{"bd"}\}_\mathbf{1})\ (\textbf{tip}_\textbf{z}\ \{p\ \texttt{"bc"}\}_\mathbf{2})))\\ \qquad\qquad (\textbf{bin}\ (\textbf{bin}\ (\textbf{tip}_\textbf{s}\ \{p\ \texttt{"ad"}\}_\mathbf{3})\ (\textbf{tip}_\textbf{z}\ \{p\ \texttt{"ac"}\}_\mathbf{4}))\ (\textbf{tip}_\textbf{z}\ \{p\ \texttt{"ab"}\}_\mathbf{5}))\end{array}$$

It's simpler to think of a tree of type $\mathsf{BT}^n_k\ p\ xs$ as a table with all the k-sublists of xs as the keys. For each key ys, there's an entry of type $p\ ys$. (So the 'T' in BT stands for both 'tree' and 'table'.)

This BT definition is really intriguing…Most likely, there is a way to derive BT from choose, and that'll work for a whole class of functions. The type of $\mathsf{BT}^n_k : (p : \mathsf{Vec}\ k\ a \to \mathsf{Set}) \to \mathsf{Vec}\ n\ a \to \mathsf{Set}$ looks just like a continuation-passing version of some $\mathsf{Vec}\ n\ a \rightsquigarrow \mathsf{Vec}\ k\ a$, which would be the type of a version of choose that nondeterministically returns a k-sublist of an n-list. And the index p works like a continuation too. Take the (expanded) type of **bin** for example:

$$\begin{array}{l}\textbf{bin} : \mathsf{BT}\ n\ (\textbf{suc}\ k)\ (\lambda\ ys \to p\ ys)\ \ \ xs\\ \quad \to \mathsf{BT}\ n\ \ \ k\ \ (\lambda\ zs \to p\ (x :: zs))\ xs \to \mathsf{BT}\ (\textbf{suc}\ n)\ (\textbf{suc}\ k)\ p\ (x :: xs)\end{array}$$

I can read it as 'to compute a $(1 + k)$-sublist of $x :: xs$ and pass it to continuation p, either compute a $(1 + k)$-sublist ys of xs and pass ys to p directly, or compute a k-sublist zs of xs and pass $x :: zs$ to p'. All the results from p are then collected in a tree as the indices of the element types. A simpler and familiar example (which can be found in the Agda standard library) is

$$\begin{array}{l}\textbf{data}\ \mathsf{All} : (a \to \mathsf{Set}) \to \mathsf{Vec}\ n\ a \to \mathsf{Set}\ \textbf{where}\\ \quad [] \ \ : \mathsf{All}\ p\ []\\ \quad \_::\_ : p\ x \to \mathsf{All}\ p\ xs \to \mathsf{All}\ p\ (x :: xs)\end{array}$$

This should be derivable from the function that nondeterministically returns an element of a list. I'm onto something general — maybe it's interesting enough for a research paper!

2.3 Specifications as Types

That paper will have to wait though. I've still got a problem to solve: how do I use BT to specify cd?

What's special about BT is that the element types are indexed by sublists, so I know from the type of an element which sublist it is associated with. That is, I can now directly say 'values associated with sublists' and rearrange these values, rather than indirectly specify the rearrangement in terms of sublists and then extend to other types of values through parametricity. $\mathsf{BT}^n_k\ p\ xs$ is the type of a tree of p-typed values associated with the k-sublists of xs, and that's precisely the intended

meaning of cd's input. What about the output? It should tabulate the $(1 + k)$-sublists of xs, so the type should be $\mathsf{BT}^n_{1+k}\ q\ xs$ for some $q : \mathsf{Vec}\ (1 + k)\ a \to \mathsf{Set}$. For each sublist $ys : \mathsf{Vec}\ (1 + k)\ a$, I want a list of p-typed values associated with the immediate sublists of ys, which are k-sublists. Or, instead of a list, I can just use a tree of type $\mathsf{BT}^{1+k}_k\ p\ ys$. Therefore the whole type is

$$\mathsf{retabulate} : k < n \to \mathsf{BT}^n_k\ p\ xs \to \mathsf{BT}^n_{1+k}\ (\mathsf{BT}^{1+k}_k\ p)\ xs$$

which is parametric in p (so, like the Haskell version of cd, the elements can have any types). I think it's time to rename cd to something more meaningful, and decide to use 'retabulate' because I'm moving values in a table into appropriate positions in a new (nested) table with a new tabulation scheme. And a side condition $k < n$ is needed to guarantee that the output shape BT^n_{1+k} is valid.

The type of retabulate looks like a sensible refinement of the type of cd, except that I'm letting retabulate return a tree of trees, rather than a tree of lists. Could that change be too drastic? Hm… actually, no — the shape of BT^{1+k}_k is always a (nonempty) list! If k is **zero**, a BT^1_0-tree has to be a $\mathbf{tip_z}$. Otherwise, a BT^{2+k}_{1+k}-tree has to take the form $\mathbf{bin}\ (\mathbf{tip_s}\ y)\ t$. This expression is in fact a cons-like operation:

$$\begin{aligned}&\_::_{\mathsf{BT}}\_ : p\ xs \to \mathsf{BT}^{1+k}_k\ (p \cdot (x ::\_))\ xs \to \mathsf{BT}^{2+k}_{1+k}\ p\ (x :: xs)\\ &y ::_{\mathsf{BT}} t = \mathbf{bin}\ (\mathbf{tip_s}\ y)\ t\end{aligned}$$

To construct a table indexed by all the immediate sublists of $x :: xs$, I need an entry for xs, the immediate sublist without x, and a table of entries for all the other immediate sublists, which are $x :: ws$ for all the immediate sublists ws of xs.

I go on to define retabulate. Its type is much more informative than that of cd. Rather than transcribing cd, can I use the type of retabulate to guide me through the implementation? I type the left-hand side into the editor, leave the right-hand side as a hole, and perform some case splitting on the input tree:

$$\begin{aligned}&\mathsf{retabulate}\ (\mathbf{tip_z}\ \_) = \{\ \}_0\\ &\mathsf{retabulate}\ (\mathbf{tip_s}\ \_) = \{\ \}_1\\ &\mathsf{retabulate}\ (\mathbf{bin}\ (\mathbf{tip_s}\ \_)\ \_) = \{\ \}_2\\ &\mathsf{retabulate}\ (\mathbf{bin}\ (\mathbf{bin}\ \_\ \_)\ (\mathbf{tip_z}\ \_)) = \{\ \}_3\\ &\mathsf{retabulate}\ (\mathbf{bin}\ (\mathbf{bin}\ \_\ \_)\ (\mathbf{tip_s}\ \_)) = \{\ \}_4\\ &\mathsf{retabulate}\ (\mathbf{bin}\ t@(\mathbf{bin}\ \_\ \_)\ u@(\mathbf{bin}\ \_\ \_)) = \{\mathsf{BT}^{2+n}_{3+k}\ (\mathsf{BT}^{3+k}_{2+k}\ p\ (x :: x_1 :: xs))\}_5\end{aligned}$$

The cases for $\mathbf{tip_s}\ \_$ and $\mathbf{bin}\ \_\ (\mathbf{tip_s}\ \_)$ can be eliminated immediately since the side condition $k < n$ is violated. I go straight to the last and the most difficult case, $\mathbf{bin}\ t\ u$, where t and u are both constructed by **bin**. Their types are

$$\begin{aligned}&t : \mathsf{BT}^{1+n}_{2+k}\ p\ (x_1 :: xs)\\ &u : \mathsf{BT}^{1+n}_{1+k}\ (p \cdot (x ::\_))\ (x_1 :: xs)\end{aligned}$$

Neither of them have the right shape to be used immediately, so in $\{\ \}_5$ I try starting with a **bin**:

$$\begin{aligned}\mathsf{retabulate}\ (\mathbf{bin}\ t@(\mathbf{bin}\ \_\ \_)\ u@(\mathbf{bin}\ \_\ \_)) = \mathbf{bin}\ &\{\mathsf{BT}^{1+n}_{3+k}\ (\mathsf{BT}^{3+k}_{2+k}\ p)\ (x_1 :: xs)\}_6\\ &\{\mathsf{BT}^{1+n}_{2+k}\ (\mathsf{BT}^{3+k}_{2+k}\ p \cdot (x ::\_))\ (x_1 :: xs)\}_7\end{aligned}$$

$\{\ \}_6$ is directly fulfilled by the induction hypothesis $\mathsf{retabulate}\ t$! That's a good sign.

What can I put in $\{\ \}_7$? Prompted by the success with $\{\ \}_6$, I try the induction hypothesis for u:

$$\mathsf{retabulate}\ u : \mathsf{BT}^{1+n}_{2+k}\ (\mathsf{BT}^{2+k}_{1+k}\ (p \cdot (x ::\_)))\ (x_1 :: xs)$$

Hm. Its type has the same outer shape $\mathsf{BT}^{1+n}_{2+k}\ \_\ (x_1 :: xs)$ that I want in the goal type, but the element types don't match…

To fill out $\{\,\}_7$ I really have to pay more attention to what the types say. I stare at the types, and, slowly, they start to make sense.

- The outer shape $\mathsf{BT}^{1+n}_{2+k}\ \_\ (x_1 :: xs)$ tells me that I'm dealing with tables indexed by the $(2 + k)$-sublists zs of $x_1 :: xs$.
- For each zs, I need to construct a table of p-typed entries indexed by all the immediate sublists of $x :: zs$, because that's what the element types $\mathsf{BT}^{3+k}_{2+k}\ p \cdot (x ::\_)$ in the goal type mean.
- What do I have in retabulate u? I have a table of entries indexed by $x :: ws$ for all the immediate sublists ws of zs in retabulate u — that's what the element types $\mathsf{BT}^{2+k}_{1+k}\ (p \cdot (x ::\_))$ mean.
- What's the relationship between these sublists $x :: ws$ and the table I need to construct, which is indexed by the immediate sublists of $x :: zs$? Oh, right, the sublists $x :: ws$ are the immediate sublists of $x :: zs$ with the first element x! So I've already got most of the entries I need.
- I'm still missing an entry for the immediate sublist of $x :: zs$ without x, which is zs. Do I have that? I search through the context. The type of t draws my attention: it has the familiar outer shape $\mathsf{BT}^{1+n}_{2+k}\ \_\ (x_1 :: xs)$. What entries are in t? Its type tells me that there's an entry for each $(2 + k)$-sublist zs of $x_1 :: xs$. That's precisely what I'm missing!

So, for each zs, I can construct a table indexed by all the immediate sublists of $x :: zs$ by combining an entry for zs (the immediate sublist without x) from t with a table of entries for the other immediate sublists with x from retabulate u. An entry and a table of entries — aren't they exactly the arguments of $\_::_{\mathsf{BT}}\_$? Indeed, I can fulfil $\{\,\}_7$ by combining all the corresponding entries in t and retabulate u (that share the same index zs) using $\_::_{\mathsf{BT}}\_$, that is, $\mathsf{zip}_{\mathsf{BT}}\mathsf{With}\ \_::_{\mathsf{BT}}\_\ t\ (\mathsf{retabulate}\ u)$!

The rest of the cases are much easier, and I come up with a definition of retabulate,

$$
\begin{aligned}
&\mathsf{retabulate} : k < n \to \mathsf{BT}^n_k\ p\ xs \to \mathsf{BT}^n_{1+k}\ (\mathsf{BT}^{1+k}_k\ p)\ xs \\
&\mathsf{retabulate}\ \{xs = \_ :: []\ \}\ (\mathbf{tip_z}\ y) &&= \mathbf{tip_s}\ (\mathbf{tip_z}\ y) \\
&\mathsf{retabulate}\ \{xs = \_ :: \_ :: \_\}\ (\mathbf{tip_z}\ y) &&= \mathbf{bin}\ (\mathsf{retabulate}\ (\mathbf{tip_z}\ y))\ (\mathbf{tip_z}\ (\mathbf{tip_z}\ y)) \\
&\mathsf{retabulate}\ (\mathbf{bin}\ (\mathbf{tip_s}\ y)\ u\ ) &&= \mathbf{tip_s}\ (y ::_{\mathsf{BT}} u) \\
&\mathsf{retabulate}\ (\mathbf{bin}\ t@(\mathbf{bin}\ \_\ \_)\ (\mathbf{tip_z}\ z)\ ) &&= \mathbf{bin}\ (\mathsf{retabulate}\ t)\ (\mathsf{map}_{\mathsf{BT}}\ (\_::_{\mathsf{BT}}\ (\mathbf{tip_z}\ z))\ t) \\
&\mathsf{retabulate}\ (\mathbf{bin}\ t@(\mathbf{bin}\ \_\ \_)\ u@(\mathbf{bin}\ \_\ \_)) &&= \mathbf{bin}\ (\mathsf{retabulate}\ t) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{zip}_{\mathsf{BT}}\mathsf{With}\ \_::_{\mathsf{BT}}\_\ t\ (\mathsf{retabulate}\ u))
\end{aligned}
$$

where the map and zip functions are the expected ones:

$$
\begin{aligned}
&\mathsf{map}_{\mathsf{BT}} : (\forall\ \{ys\} \to p\ ys \to q\ ys) \to \forall\ \{xs\} \to \mathsf{BT}^n_k\ p\ xs \to \mathsf{BT}^n_k\ q\ xs \\
&\mathsf{zip}_{\mathsf{BT}}\mathsf{With} : (\forall\ \{ys\} \to p\ ys \to q\ ys \to r\ ys) \\
&\qquad\qquad \to\ \forall\ \{xs\} \to \mathsf{BT}^n_k\ p\ xs \to \mathsf{BT}^n_k\ q\ xs \to \mathsf{BT}^n_k\ r\ xs
\end{aligned}
$$

Amazingly, the more informative type of retabulate did help me to develop its definition. As I filled in the holes, I didn't feel I had much of a choice — in a good way, because that reflected the precision of the type. Furthermore, I discovered a new presentation that varies slightly from Richard's cd! The first two cases of cd are subsumed by the third case, $\mathbf{bin}\ (\mathbf{tip_s}\ y)\ u$, of retabulate. The second case of cd recursively traverses the given tree to convert it to a list, which is not needed in retabulate, because it yields a tree of trees. Therefore the first two cases of cd can be unified into one. I forgot to include the side condition $1 \le k$ in retabulate, but that leads to two new $\mathbf{tip_z}$ cases instead of preventing me from completing the definition. Whereas cd has to start from level 1 of the sublist lattice (Figure 2), this pair of cases of retabulate is capable of producing a level-1 table (with as many elements as xs) from a level-0 table, which is a $\mathbf{tip_z}$. This is due to xs now being available as an index, providing the missing context for retabulate.

In fact, looking at the type more closely, I suspect that the extensional behaviour of retabulate is completely determined by the type (so the type works as a nice and tight specification): the shape

of the output table is completely determined by the indices; moreover, all the input elements have distinct types in general, so each element in the output table has to be the only input element with the required type — there is no choice to make. Formally, the proof will most likely be based on parametricity (and might be similar to Voigtländer's [2009]). That'll probably be a fun exercise… but I'll leave that for another day.

2.4 Precision of Types

Right now I'm more eager to understand why the bottom-up algorithm `bu` equals the top-down `td` (Sections 1.1 and 1.2). Will dependent types continue to be helpful? I should try and find out by transcribing the two algorithms into Agda too.

The first thing to do is make the type of the two algorithms as precise as the type of retabulate. I start from the combining function `g`. Its type `L s -> s` has been making me shudder involuntarily whenever I see it: it says so little about what `g` should do. The intention — that `g` should compute the solution for a list from those of its immediate sublists — is nowhere to be seen.

But now I have the right vocabulary to state the intention precisely in Agda. I can use BT to say things about all sublists of a particular length. And to say 'a solution for a list' (instead of just 'a solution') I should switch from a type to a type family

$$s \;:\; \forall\,\{k\} \to \mathsf{Vec}\ k\ a \to \mathsf{Set}$$

such that $s\ ys$ is the type of solutions for ys. So

$$g \;:\; \forall\,\{k\} \to \{ys \;:\; \mathsf{Vec}\ (2 + k)\ a\} \to \mathsf{BT}^{2+k}_{1+k}\ s\ ys \to s\ ys$$ ✘

I look at the type with a satisfied smile — now that's what I call a nice and informative type! It says concisely and precisely what g should do: compute a solution for any $ys : \mathsf{Vec}\ (2 + k)\ a$ from a table of solutions for all the length-$(1 + k)$ (that is, immediate) sublists of ys.

The smile quickly turns into a frown though. I still don't feel comfortable with BT^{2+k}_{1+k}. The indices are apparently not the most general ones — why not BT^{1+k}_{k}?

I delete (✘) the type of g and start pondering. I wrote BT^{2+k}_{1+k} in that type because that was what Richard wanted to say. Richard used singleton lists as the base cases instead of the empty list, so `g` was applied to solutions for sublists of length at least 1, hence the subscript $1 + k$. But the most general type

$$g \;:\; \forall\,\{k\} \to \{ys \;:\; \mathsf{Vec}\ (1 + k)\ a\} \to \mathsf{BT}^{1+k}_{k}\ s\ ys \to s\ ys$$

looks just fine. In particular, when k is 0, the type says that g should compute a solution for a singleton list from a solution for the empty list (the only immediate sublist of a singleton list), which seems reasonable…

…And indeed it is reasonable!

When I discovered the extra $\mathbf{tip}_\mathbf{z}$ cases of retabulate (Section 2.3), I saw that it may be possible to start from level 0 of the sublist lattice (Figure 2) after all. Now it's confirmed. Instead of starting with solutions for singleton lists, I can start with a given solution for the empty list

$$e \;:\; s\ []$$ ✘

at level 0, and work upwards using g. There's no problem going from level 0 to level 1, because there's now additional context in the indices so that g knows for which singleton list a solution should be computed. Making types precise has helped me to find a more natural and general form of recursive computation over immediate sublists!

And it's not just any recursive computation — it's now an alternative *induction principle* for lists. The base case e is still about the empty list, and the inductive case g assumes that the induction hypothesis holds for all the immediate sublists. (I guess I've been instinctively drawn towards induction principles, in common with most dependently typed programmers.)


```
ImmediateSublistInduction : Set₁
ImmediateSublistInduction =
  {a : Set} (s  : ∀ {k} → Vec k a → Set)
            (e  : s [])
            (g  : ∀ {k} → {ys : Vec (1 + k) a} → BT^{1+k}_k s ys → s ys)
  {n : ℕ}   (xs : Vec n a) → s xs
```

* * *

ImmediateSublistInduction looks very nice, but I'm still worried: will it be possible to transcribe `td` and `bu` for this type easily, that is, without ugly stuff like type conversions (subst, **rewrite**, etc)?

Sadly, there may never be any theory that tells me quickly whether or not a type admits pretty programs. The only way to find out is to try writing some. I start transcribing `td` (Section 1.1). The only component for which I still don't have a dependently typed version is `dc`, which I've generalised to `choose` (Section 2.3). Now I can give it a precise type:

```
choose : (n k : ℕ) → k ≤↑ n → (xs : Vec n a) → BT^n_k Exactly xs
```

The key ingredient is the BT^n_k in the result type, which tabulates all the k-sublists as the indices of the element types. I just need to plug in this data type

```
data Exactly : a → Set where
  exactly : (x : a) → Exactly x
```

to say that the elements in the resulting table should be exactly the tabulated indices. In contrast to the Haskell version, I need to make the Agda version of choose total by saying explicitly that the length n of the list xs is at least k, so that there are enough elements to choose from. Somewhat notoriously, there are many versions of natural number inequality. To align with the inductive structure of `choose`, I pick the data type $m \leq_\uparrow n$ where m remains fixed throughout the definition and serves as an 'origin', and an inhabitant of $m \leq_\uparrow n$ is the distance (which is essentially a natural number) that n is away from the origin m:

```
data _≤↑_ : ℕ → ℕ → Set where
  zero :          m ≤↑ m
  suc  : m ≤↑ n → m ≤↑ suc n
```

Given the precise type, transcribing `choose` is pretty much straightforward:

```
choose : (n k : ℕ) → k ≤↑ n → (xs : Vec n a) → BT^n_k Exactly xs
choose _       zero    _       _        = tip_z (exactly [])
choose (suc k) (suc k) zero    xs       = tip_s (exactly xs)
choose (suc n) (suc k) (suc d) (x :: xs) =
  bin (choose n (suc k) d xs) (map_BT (map_Exactly (x ::_)) (choose n k (incr d) xs))
    -- map_Exactly : (f : a → b) → {x : a} → Exactly x → Exactly (f x)
```

The function performs an induction on k, like the Haskell version. The second and third cases are an inner induction on the distance that n is away from **suc** k. In the second case the distance is **zero**, meaning that n is (at the origin) **suc** k, so xs has the right length and can be directly returned.

Otherwise the distance is **suc** d in the third case, where the first inductive call is on d, and the second inductive call is on k while the distance is unchanged, but the type **suc** $k \leq_\uparrow$ **suc** n needs to be adjusted to $k \leq_\uparrow n$ (by invoking incr : **suc** $m \leq_\uparrow n \to m \leq_\uparrow n$ on d).

* * *

I step back and take another look at choose. There's one thing that bothers me: Exactly x is a type that has a unique inhabitant, so I could've used the unit type $\top$ as the element types instead, and I'd still give the same amount of information, which is none! That doesn't make a lot of sense — I thought I was computing all the k-sublists and returning them in a table, but somehow those sublists didn't really matter, and I could just return a blank table of type BT^n_k (const $\top$) xs…?

Hold on, it's actually making sense…

It's because all the information is already in the table structure of BT. Indeed, I can write

$$\mathsf{map}_{\mathsf{BT}}\ (\lambda\ \{\{ys\}\ \mathbf{tt} \to \mathbf{exactly}\ ys\})\ :\ \mathsf{BT}^n_k\ (\mathsf{const}\ \top)\ xs \to \mathsf{BT}^n_k\ \mathsf{Exactly}\ xs$$

to recover a table of Exactlys from just a blank table by replacing every **tt** (the unique inhabitant of $\top$) with the index ys there. What choose does is not really compute the sublists — BT has already 'computed' them. Instead, choose merely affirms that there is a table indexed by all the k-sublists of an n-list whenever $k \leq_\uparrow n$. The elements in the table don't matter, and might as well be **tt**.

So, instead of choose, I can use

```
blank : (n k : ℕ) → k ≤↑ n → {xs : Vec n a} → BT^n_k (const ⊤) xs
blank _       zero    _                 = tip_z tt
blank (suc k) (suc k) zero              = tip_s tt
blank (suc n) (suc k) (suc d) {_ :: _}  = bin (blank n (suc k) d) (blank n k (incr d))
```


to construct a blank table indexed by all the immediate sublists in the inductive case of td, where I'll then compute and fill in solutions for all the immediate sublists by invoking td inductively, and finally invoke g to combine all those solutions. And the base case simply returns e.

```
td : ImmediateSublistInduction
td s e g {zero } []  = e
td s e g {suc n} xs  = g (map_BT (λ {{ys} tt → td s e g {n} ys}) (blank^{1+n}_n (suc zero)))
```


I look aghast at the monster I've created. Sure, the definition type-checks, but oh my… it's terribly ugly. The problems are cosmetic though, and should be easy to fix. (1) The induction is on n, which shouldn't have been an implicit argument. (2) In the base case, I have to match xs with [] to make it type-correct to return e, but that could've been avoided if I set e : $\{xs$: Vec 0 $a\} \to s\ xs$. (3) In the inductive case, xs doesn't need to be explicit because it's passed around only implicitly in the indices on the right-hand side. (4) I can get rid of the λ around the inductive call to td if I make ys implicit and add a dummy $\top$ argument. In fact, if I add a dummy $\top$ argument to e and blank as well, I can make the definition point-free like in Richard's paper — a temptation I cannot resist. So I revise the induction principle,

```
ImmediateSublistInduction : Set₁
ImmediateSublistInduction =
  {a : Set} (s : ∀ {k} → Vec k a → Set)
            (e : {ys : Vec 0 a} → ⊤ → s ys)
            (g : ∀ {k} → {ys : Vec (1 + k) a} → BT^{1+k}_k s ys → s ys)
            (n : ℕ) {xs : Vec n a} → ⊤ → s xs
```

add the dummy $\top$ argument to blank (also ignoring the inequality argument from now on),

$$\mathsf{blank} \;:\; (n\ k \;:\; \mathbb{N}) \to k \leq_{\uparrow} n \to \{xs \;:\; \mathsf{Vec}\ n\ a\} \to \top \to \mathsf{BT}^n_k\ (\mathsf{const}\ \top)\ xs$$

and get my point-free td:

$$\begin{aligned}
&\mathsf{td} \;:\; \mathsf{ImmediateSublistInduction}\\
&\mathsf{td}\ s\ e\ g\ \ \mathbf{zero}\ \ \ \ = e\\
&\mathsf{td}\ s\ e\ g\ (\mathbf{suc}\ n) \;=\; g \cdot \mathsf{map}_{\mathsf{BT}}\ (\mathsf{td}\ s\ e\ g\ n) \cdot \mathsf{blank}^{1+n}_n
\end{aligned}$$

The revised ImmediateSublistInduction may not be too user-friendly, but that can be amended later (when there's actually a user).

* * *

And it'd be wonderful if the revised ImmediateSublistInduction worked for bu too! I proceed to transcribe bu (Section 1.2):

$$\begin{aligned}
&\mathsf{bu} \;:\; \mathsf{ImmediateSublistInduction}\\
&\mathsf{bu}\ s\ e\ g\ n \;=\; \mathsf{unTip} \cdot \mathsf{loop}\ 0\ \{0 \downarrow\leq n\}_{\mathbf{0}}\ \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}^n_0\\
&\quad \mathbf{where}\ \mathsf{loop} \;:\; (k \;:\; \mathbb{N}) \to k \downarrow\leq n \to \mathsf{BT}^n_k\ s\ xs \to \mathsf{BT}^n_n\ s\ xs\\
&\qquad\quad \mathsf{loop}\ n\ \ \mathbf{zero}\ \ \ \ = \mathsf{id}\\
&\qquad\quad \mathsf{loop}\ k\ (\mathbf{suc}\ d) \;=\; \mathsf{loop}\ (1 + k)\ d \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate}
\end{aligned}$$

I construct the initial table by reusing blank to create a blank level-0 table and using $\mathsf{map}_{\mathsf{BT}}\ e$ to fill in the initial solution for the empty list. Then the loop function increases the level to n by repeatedly retabulating a level-k table as a level-$(1 + k)$ table and filling in solutions for all the length-$(1 + k)$ sublists using $\mathsf{map}_{\mathsf{BT}}\ g$. Finally, a solution for the whole input list is extracted from the level-n table using

$$\begin{aligned}
&\mathsf{unTip} \;:\; \mathsf{BT}^n_n\ p\ xs \to p\ xs\\
&\mathsf{unTip}\ \ \ \ \ \ \ \ \ \ (\mathbf{tip_s}\ p) \;=\; p\\
&\mathsf{unTip}\ \{xs = []\}\ (\mathbf{tip_z}\ p) \;=\; p
\end{aligned}$$

The **bin** case is impossible and ignored. (Hm, retabulate and unTip smell comonadic, and maybe the indices constitute a grading [Gaboardi et al. 2016]…but I can't get distracted now.)

The argument/counter k of loop should satisfy the invariant $k \downarrow\leq n$. Again, this version of natural number inequality is chosen to align with the inductive structure of loop. The data type $m \downarrow\leq n$ is dual to $\_\leq_{\uparrow}\_$ in the sense that this time it is n that is fixed throughout the definition, and m moves away from n:

$$\begin{aligned}
&\mathbf{data}\ _\downarrow\leq_ \;:\; \mathbb{N} \to \mathbb{N} \to \mathsf{Set}\ \mathbf{where}\\
&\quad \mathbf{zero} \;:\; \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ n \downarrow\leq n\\
&\quad \mathbf{suc}\ \ \ \;:\; \mathbf{suc}\ m \downarrow\leq n \to m \downarrow\leq n
\end{aligned}$$

Then loop simply performs induction on the distance $k \downarrow\leq n$; the counter k goes up as the distance decreases in inductive calls, and eventually reaches n when the distance becomes **zero**. The remaining $\{0 \downarrow\leq n\}_{\mathbf{0}}$ is actually nontrivial, but the Agda standard library covers that.

2.5 Equality from Types

Okay, I've made the type of both td and bu precise. How does this help me prove td equals bu? The definitions still look rather different except for their type…

…And the type is an induction principle.

Is it possible to have extensionally different implementations of an induction principle?

Let me think about the induction principle of natural numbers.

```
ℕ-Induction : Set₁
ℕ-Induction = (p : ℕ → Set) (pz : p zero) (ps : ∀ {m} → p m → p (suc m))
              (n : ℕ) → p n
ind : ℕ-Induction
ind p pz ps zero    = pz
ind p pz ps (suc n) = ps (ind p pz ps n)
```

The motive p is parametrically quantified, so a proof of $p\ n$ has to be n applications of ps to pz. There are intensionally different ways to construct that (ind versus a tail-recursive implementation, for example), but extensionally they're all the same.

Of course, parametricity is needed to prove that formally. I look up Bernardy et al.'s [2012] translation, which (after a bit of simplification) gives the following statement:

```
ℕ-Induction-unary-parametricity : ℕ-Induction → Set₁
ℕ-Induction-unary-parametricity f =
  {p  : ℕ → Set}                    (q  : ∀ {m} → p m → Set)
  {pz : p zero}                     (qz : q pz)
  {ps : ∀ {m} → p m → p (suc m)}    (qs : ∀ {m} {x : p m} → q x → q (ps x))
  {n  : ℕ} → q (f p pz ps n)
```

Unary parametricity can be thought of as adding an invariant (q) to a parametrically quantified type or type family; this invariant is assumed to hold for any first-order input (qz) and be preserved by any higher-order input (qs), and is guaranteed to hold for the output. Now choose the invariant that any proof of $p\ m$ is equal to the one produced by ind $p\ pz\ ps\ m$, and that's it:

```
ℕ-Induction-uniqueness-from-parametricity :
    (f : ℕ-Induction) → ℕ-Induction-unary-parametricity f
  → (p : ℕ → Set) (pz : p zero) (ps : ∀ {m} → p m → p (suc m)) (n : ℕ)
  → f p pz ps n ≡ ind p pz ps n
ℕ-Induction-uniqueness-from-parametricity f param p pz ps n =
  param (λ {m} x → x ≡ ind p pz ps m) refl (cong ps)
    -- refl : {x : a} → x ≡ x;  cong : (f : a → b) → {x y : a} → x ≡ y → f x ≡ f y
```

The same argument works for ImmediateSublistInduction — any function of the type satisfying unary parametricity is pointwise equal to td. I finish the Agda proofs for both induction principles in a dreamlike state.

Yeah, I have a proof that td equals bu.

Well, strictly speaking I don't have one yet. (Vanilla) Agda doesn't have internal parametricity [Van Muylder et al. 2024], so I'd need to prove the parametricity of bu, painfully. But there shouldn't be any surprise.

Somehow I feel empty though. I was expecting a more traditional proof based on equational reasoning. This kind of proof may require more work, but allows me to compare what td and bu do *intensionally*. That's an aspect overlooked from the parametricity perspective. Despite having a proof now, I think I'm going to have to delve into the definitions of td and bu anyway, to get a clearer picture of their relationship.

3 DIAGRAMS

Another hint Richard left was 'naturality', a category-theoretic notion which he used a lot in his paper. In functional programming, naturality usually stems from parametric polymorphism: all parametric functions, such as `cd` and `unTip`, satisfy naturality. I've got some parametric functions too, such as retabulate and `unTip`. Their dependent function types with all the indices are more advanced than Richard's types though, and the simply typed form of naturality Richard used no longer makes sense. But one nice thing about category theory is its adaptability — all I need to figure out is which category I'm in, and then I'll be able to work out what naturality means for my functions systematically within the world of category theory.

And, if the key is naturality, now I have an additional tool that Richard didn't: string diagrams. I've seen how dramatically string diagrams simplify proofs using naturality, so it's probably worthwhile to take a look at the two algorithms from a string-diagrammatic perspective.

But before I get to string diagrams, I need to work through some basic category theory…

3.1 From Categories to String Diagrams

Functional programmers are familiar with types and functions, and know when functions can be composed sequentially — when adjacent functions meet at the same type. And it's possible to compose an empty sequence of functions, in which case the result is the identity function. *Categories* are settings in which the same intuition about sequential composition works. Instead of types and functions, the categorical programmer can switch to work with some *objects* and *morphisms* specified by a category, where each morphism is labelled with a source object and a target object (like the source and target types of a function), and morphisms can be composed sequentially when adjacent morphisms meet at the same object. And, like identity functions, there are identity morphisms too. Working in a new setting that's identified as a category is a blessing for the functional programmer: it means that the programmer can still rely on some of their intuitions about types and functions to navigate in the new setting. More importantly, some notions that prove to be useful in functional programming (such as naturality) can be defined generically on categories and systematically transported to other settings.

A clue about the kind of category I'm in is that I'm tempted to say 'retabulate transforms a tree of p's to a tree of trees of p's'. When a simply typed functional programmer says 'a tree of something', that 'something' is a type, that is, an object in the familiar category of types and functions. But here p is not a type. It's a type family. So I've landed in a different kind of category where the objects are type families.

There are quite a few versions of 'categories of families'. I go through the types of the components used in the algorithms (Section 2.4) to find a common form, and it seems that the simplest version suffices: given an index type $a : \mathsf{Set}$, a category of families **Fam** a has objects of type

$$\begin{aligned}&\mathsf{Fam} : \mathsf{Set} \to \mathsf{Set}_1\\&\mathsf{Fam}\ a = a \to \mathsf{Set}\end{aligned}$$

and morphisms of type

$$\begin{aligned}&\_\rightrightarrows\_ : \mathsf{Fam}\ a \to \mathsf{Fam}\ a \to \mathsf{Set}\\&p \rightrightarrows q = \forall\ \{x\} \to p\ x \to q\ x\end{aligned}$$

That is, a morphism from p to q is a family of functions between corresponding types (with the same index) in p and q. Everything in the definition is parametrised by the index type a, so actually I'm working in not just one but many related categories of families, with different index types. These categories are still inherently types and functions, so it's no surprise that their sequential composition works in the way familiar to the functional programmer.

With the definition of **Fam**, now I can rewrite the parametric function types of retabulate and unTip to look more like the ones in Haskell:

$$\begin{aligned}&\mathsf{retabulate} \;:\; \mathsf{BT}^n_k\; p \rightrightarrows \mathsf{BT}^n_{1+k}\; (\mathsf{BT}^{1+k}_k\; p)\\ &\mathsf{unTip} \quad\;\; :\; \mathsf{BT}^n_n\; p \rightrightarrows p\end{aligned}$$

I can fit blank^n_k into **Fam** (Vec n a) by lifting its $\top$ argument to a type family const $\top$ (that is, an object of the category):

$$\mathsf{blank}^n_k \;:\; \mathsf{const}\;\top \rightrightarrows \mathsf{BT}^n_k\;(\mathsf{const}\;\top)$$

The base and inductive cases of ImmediateSublistInduction fit into these **Fam** categories too: given a : Set and s : $\forall\,\{k\} \to$ Fam (Vec k a), I can write

$$\begin{aligned}&g \;:\; \mathsf{BT}^{1+k}_k\; s \rightrightarrows s\\ &e \;:\; \mathsf{const}\;\top \rightrightarrows s\,\{0\}\end{aligned}$$

(It's important to say explicitly that the target of e is $s\,\{0\}$: Fam (Vec 0 a) to make it clear that e gives a solution for the empty list.)

* * *

I haven't done much really. It's just a bit of abstraction that hides part of the indices, and might even be described as cosmetic. What's important is that, by fitting my programs into the **Fam** categories, I can start talking about them in categorical language. In particular, I want to talk about naturality. That means I should look for *functors* and *natural transformations* in my programs.

A parametric data type such as BT^n_k is categorically the object part of a *functor*, which maps objects in a category to objects in a possibly different category. In the case of BT^n_k, the functor goes from **Fam** (Vec k a) to **Fam** (Vec n a) — indeed I can rewrite the type of BT^n_k as

$$\mathsf{BT}^n_k \;:\; \mathsf{Fam}\;(\mathsf{Vec}\;k\;a) \to \mathsf{Fam}\;(\mathsf{Vec}\;n\;a)$$

The BT-typed trees are made up of the constructors of BT and hold elements of types p. A categorical insight is that the constructors constitute an *independent* layer of data added by the functor outside the elements. The independence of this functor layer is described formally by the definitions of functors and natural transformations.

One aspect of this independence is that the functor layer can stay the same and impervious to whatever is happening at the inner layer. Categorically, 'whatever is happening' means an arbitrary morphism. In the case of BT, the inner layer (the elements) may be changed by some arbitrary morphism of type $p \rightrightarrows q$, and that can always be lifted to a morphism of type $\mathsf{BT}^n_k\; p \rightrightarrows \mathsf{BT}^n_k\; q$ that doesn't change the functor layer (the tree constructors). This lifting is the morphism part of a functor, and is the 'map' function that comes with any (normal) parametric data type. I've already had a map function for BT, and indeed its type can be rewritten as

$$\mathsf{map}_{\mathsf{BT}} \;:\; (p \rightrightarrows q) \to (\mathsf{BT}^n_k\; p \rightrightarrows \mathsf{BT}^n_k\; q)$$

In a sense, a lifted morphism such as $\mathsf{map}_{\mathsf{BT}}\; f$ is essentially just f since $\mathsf{map}_{\mathsf{BT}}\; f$ does nothing to the functor layer. So when f is a composition, that composition shows up at the level of lifted morphisms too. Formally, this is stated as a *functoriality* equation:

$$\mathsf{map}_{\mathsf{BT}}\;(f' \cdot f) \;=\; \mathsf{map}_{\mathsf{BT}}\; f' \cdot \mathsf{map}_{\mathsf{BT}}\; f$$

(Also $\mathsf{map}_{\mathsf{BT}}$ id = id in the degenerate case of composing no morphisms.)

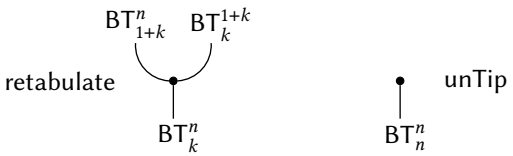
The functor layer may also be changed by *natural transformations* independently of whatever is happening at the inner layer. In **Fam** categories, a natural transformation has type $\forall\,\{p\} \to F\; p \rightrightarrows G\; p$ for some functors F and G, and transforms an F-layer to a G-layer without changing the inner

layer p, whatever p is. For example, retabulate transforms the functor layer BT^n_k to a *composition* of functors $\mathsf{BT}^n_{1+k} \cdot \mathsf{BT}^{1+k}_k$ (which can be regarded as two functor layers) without changing p. Indeed, retabulate transforms only the tree constructors and doesn't change the elements to something else. Moreover, this transformation of the functor layer does not interfere with whatever is happening at the inner layer. Again 'whatever is happening' amounts to a quantification over all morphisms: for any $f : p \rightrightarrows q$ happening at the inner layer, if retabulate is happening at the functor layer too, it doesn't make a difference whether f or retabulate happens first, because they happen at independent layers. Formally, this is stated as a *naturality* equation (where f needs to be lifted appropriately):

$$\mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\, f \;=\; \mathsf{map}_{\mathsf{BT}}\, (\mathsf{map}_{\mathsf{BT}}\, f) \cdot \mathsf{retabulate}$$

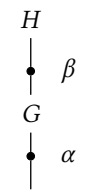
* * *

With functor composition, in general there can be many functor layers in an object (like the target of retabulate), and all these layers can be transformed independently by natural transformations. The best way of managing this structure is to use *string diagrams*. In string diagrams, functors are drawn as wires, and natural transformations are drawn as dots with input functors/wires attached below and output functors/wires above. (I learned string diagrams mainly from Coecke and Kissinger [2017], so my string diagrams have inputs below and outputs on top.) The natural transformations I've got are retabulate and unTip, and I can draw their types as

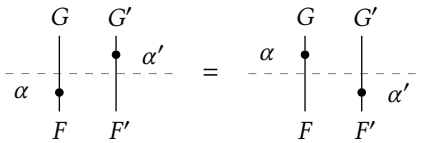


String diagrams focus on the functor layers and represent them explicitly as a bunch of wires — functor composition is represented as juxtaposition of wires, and the identity functor is omitted (it is drawn as no wires). As a string diagram, retabulate has one input wire labelled BT^n_k and two output wires BT^n_{1+k} and BT^{1+k}_k, since it transforms a BT^n_k layer to two layers $\mathsf{BT}^n_{1+k} \cdot \mathsf{BT}^{1+k}_k$. The diagram of unTip goes from one wire to none, since unTip transforms BT^n_n to the identity functor. Indeed, what unTip does is get rid of the $\mathbf{tip_z}$ or $\mathbf{tip_s}$ constructor.

Whereas functor composition is arranged horizontally, sequential composition of natural transformations goes vertically. Given transformations $\alpha : \forall\, \{p\} \to F\ p \rightrightarrows G\ p$ and $\beta : \forall\, \{p\} \to G\ p \rightrightarrows H\ p$, their sequential composition $\beta \cdot \alpha : \forall\, \{p\} \to F\ p \rightrightarrows H\ p$ is drawn in a string diagram as α and β juxtaposed vertically and sharing the middle wire with label G (obscuring a section of the wire).



The power of string diagrams becomes evident when things happen in both the horizontal and vertical dimensions. For example, suppose there are two layers F and F', where the outer layer F should be transformed by α and the inner layer F' by $\alpha' : \forall\, \{p\} \to F'\ p \rightrightarrows G'\ p$. There are two ways of doing this: either $\mathsf{map}_G\ \alpha' \cdot \alpha$, where the outer layer F is transformed to G first, or $\alpha \cdot \mathsf{map}_F\ \alpha'$, where the inner layer F' is transformed to G' first. The two ways are equal by naturality of α, but the equality can be seen more directly with string diagrams:



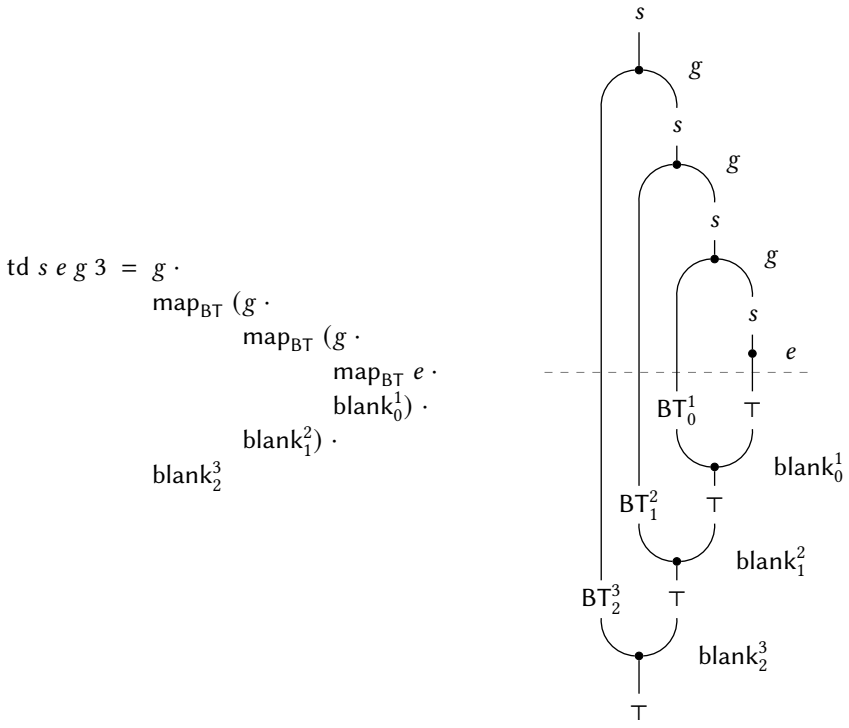
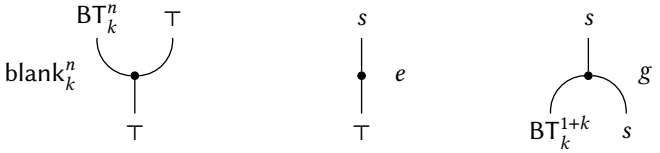


Fig. 4. A special case of the top-down algorithm as a string diagram.

The map means skipping over the outer/left functor and transforming the inner/right functor; so in the diagrams, α' is applied to the inner/right wire. (I've added dashed lines to emphasise that both diagrams are constructed as the sequential composition of two transformations.) By placing layers of functors in a separate dimension, it's much easier to see which layers are being transformed, and determine whether two sequentially composed transformations are in fact applied independently, so that their order of application can be swapped. This is abstracted as a diagrammatic reasoning principle: dots in a diagram can be moved upwards or downwards, possibly changing their vertical positions relative to other dots (while stretching or shrinking the wires, which can be thought of as elastic strings), and the (extensional) meaning of the diagram will remain the same.

* * *

I want to draw td and bu as string diagrams. However, some of their components, namely blank, e, and g, are not natural transformations. Technically, only natural transformations can go into string diagrams. But I'm still tempted to draw those components intuitively as



After a bit of thought, I come up with some technical justification. Any morphism $f : p \rightrightarrows q$ can be lifted to have the type $\forall\ \{r\} \to (\mathsf{const}\ p)\ r \rightrightarrows (\mathsf{const}\ q)\ r$, and become a natural transformation

$$
\begin{aligned}
& \mathsf{td}\ s\ e\ g\ 3 \\
= \quad & \{\text{- definition -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}_0^1) \cdot \mathsf{blank}_1^2) \cdot \mathsf{blank}_2^3 \\
= \quad & \{\text{- functoriality -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ e) \cdot \mathsf{map}_{\mathsf{BT}}\ \mathsf{blank}_0^1 \cdot \mathsf{blank}_1^2) \cdot \mathsf{blank}_2^3 \\
= \quad & \{\text{- functoriality -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ e)) \cdot \\
& \mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ \mathsf{blank}_0^1 \cdot \mathsf{blank}_1^2) \cdot \mathsf{blank}_2^3
\end{aligned}
$$

Fig. 5. Rewriting td $s\ e\ g\ 3$ into two phases using functoriality.

from const p to const q. It's fine to leave the lifting implicit and just write p and q for wire labels, since it's usually clear that p and q are not functors and need to be lifted. For example, s is a type family, which is an object in a **Fam** category, and needs to be lifted to const s to be a functor. For $\top$ there's one more step: $\top$ abbreviates the type family const $\top$, which is an object in a **Fam** category, and needs to be further lifted to const (const $\top$) to be a functor. It's kind of technical, but in the end these diagrams are okay.

3.2 Diagrammatic Reasoning

All the abstract nonsense took me some time. But I still don't know whether string diagrams will actually help me to understand the two algorithms (Section 2.4). It's time to find out.

I'm not confident enough to work with the full recursive definitions straight away, so I take the special case td $s\ e\ g\ 3$ of the top-down algorithm and unfold it into a deeply nested expression $g \cdot \mathsf{map}_{\mathsf{BT}}\ (\ldots) \cdot \mathsf{blank}_2^3$ (as on the left of Figure 4). The rightmost component blank_2^3 has type const $\top \rightrightarrows \mathsf{BT}_2^3$ (const $\top$), which — after eliding the consts, as I've just decided to do — is drawn as the bottom Y-junction in the string diagram (as on the right of Figure 4), transforming the input $\top$-wire at the bottom into two wires BT_2^3 and $\top$. Then the $\mathsf{map}_{\mathsf{BT}}$ means that the left wire BT_2^3 is skipped over and unchanged. The right $\top$-wire continues to be transformed by blank_1^2 and so on, and eventually becomes an s-wire. Finally, I apply g to the BT_2^3-wire I skipped over and the s-wire to produce the output s-wire at the top.

I step back and compare the expression and the diagram (Figure 4). All the $\mathsf{map}_{\mathsf{BT}}$s are gone in the diagram, because I can directly apply a transformation to the intended layers/wires, rather than count awkwardly how many outer layers I have to skip, using $\mathsf{map}_{\mathsf{BT}}$ one layer at a time. Functoriality is also transparent in the diagram, so it's slightly easier to see that td has two phases (which I have separated by a dashed line): the first phase constructs deeply nested blank tables, and the second phase fills and demolishes the tables inside out.

Functoriality is already somewhat transparent in the traditional expression though, thanks to the infix notation of function composition. So I suppose I don't absolutely need the string diagram to see that td has two phases, although the required rewriting (Figure 5) is not as perspicuous as just seeing the two phases in the diagram. Moreover, there's nothing I can meaningfully move in the diagram — all the transformations here are lifted after all.

Hm. Maybe I'll have more luck with the bottom-up algorithm, which has 'real' natural transformations? I go on to expand bu $s\ e\ g\ 3$. The loop in the expression unfolds into a sequence of functions, alternating between retabulate and $\mathsf{map}_{\mathsf{BT}}\ g$.

'A sequence…' I mutter. I shouldn't have expected anything else from unfolding a loop. But the sequential structure is so different from the deeply nested structure of td.

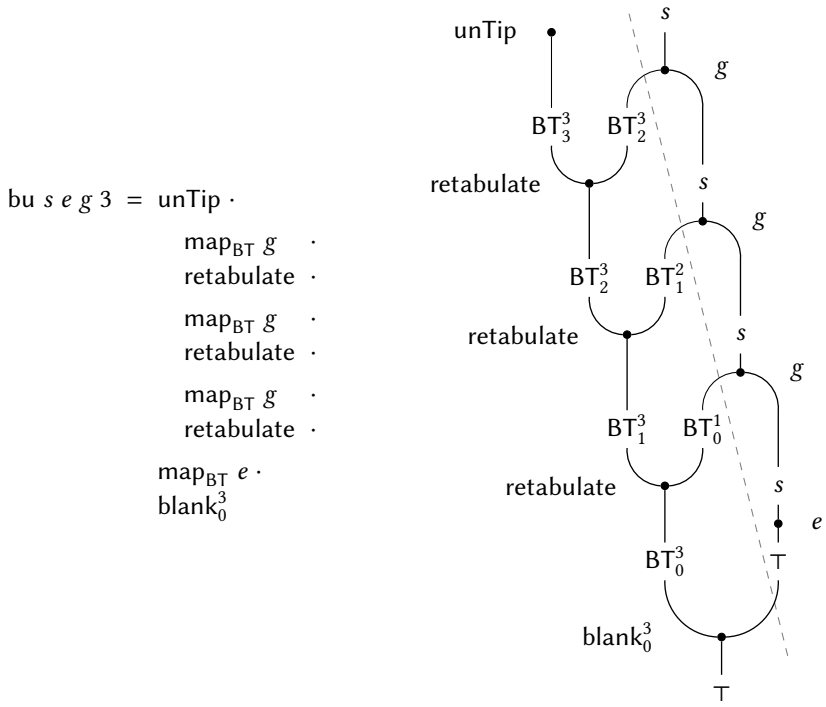


Fig. 6. A special case of the bottom-up algorithm as a string diagram.

And then, something unexpected yet familiar appears in the transcribed diagram (Figure 6).

There are also two phases for table construction and demolition, and the gs and e in the demolition phase are *exactly the same* as in td!

The string diagram is truly helpful this time. Now I see, as Richard hinted, that I could rewrite the traditional expression using the naturality of unTip and retabulate to push g and e to the left of the sequence and separate the two phases (Figure 7). But in the string diagram, all those rewritings amount to nothing more than gradually pulling the two phases apart, eventually making the dashed line horizontal. In fact I don't even bother to pull, because on this diagram I can already see simultaneously both the sequence (the dots appearing one by one vertically) and the result of rewriting the sequence using naturality.

* * *

So, modulo naturality, the two algorithms have the same table demolition phase but different table construction phases. If I can prove that their table construction phases are equal, then I'll have another proof that the two algorithms are equal, in addition to the parametricity-based proof (Section 2.5). For td, the construction phase is a right-leaning tree on the diagram, whereas for bu it's a left-leaning tree. Maybe what I need is an equation about blank and retabulate that can help me to rotate a tree…?

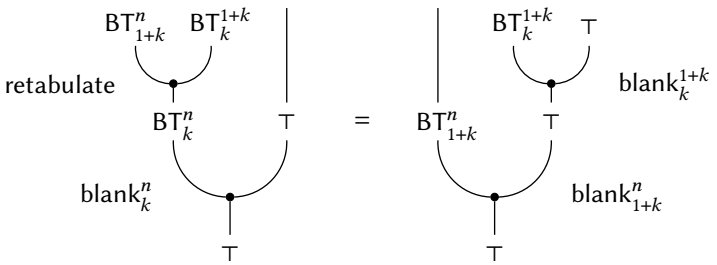
```
cd (choose k xs) = mapB (flatten . choose k) (choose (k+1) xs)
```

The equation (∗) flashes through my mind. Of course it has to be this equation — I used it as a specification for cd, the Haskell progenitor of retabulate. How else would I introduce retabulate

$$
\begin{aligned}
& \mathsf{bu}\ s\ e\ g\ 3 \\
= \quad & \{\text{- definition -}\} \\
& \mathsf{unTip} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}_0^3 \\
= \quad & \{\text{- naturality of unTip -}\} \\
& g \cdot \\
& \mathsf{unTip} \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}_0^3 \\
= \quad & \{\text{- naturality of retabulate -}\} \\
& g \cdot \\
& \mathsf{unTip} \cdot \mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ g) \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}_0^3 \\
= \quad & \{\text{- naturality of unTip again -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \\
& \mathsf{unTip} \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{retabulate} \cdot \mathsf{map}_{\mathsf{BT}}\ e \cdot \mathsf{blank}_0^3 \\
= \quad & \{\text{- similarly -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ g \cdot \mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ g) \cdot \mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ e)) \cdot \\
& \mathsf{unTip} \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{blank}_0^3 \\
= \quad & \{\text{- functoriality -}\} \\
& g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ (g \cdot \mathsf{map}_{\mathsf{BT}}\ e)) \cdot \\
& \mathsf{unTip} \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{retabulate} \cdot \mathsf{blank}_0^3
\end{aligned}
$$

Fig. 7. Rewriting bu $s\ e\ g\ 3$ into two phases using naturality.

into the picture? But first let me update this for my dependently typed string diagrams:



(**)

That's a tree rotation all right! So I should do an induction that uses this equation to rotate the right-leaning tree in td and obtain the left-leaning tree in bu (Figure 8). And then I'll need to prove the equation, meaning that I'll need to go through the definitions of retabulate and blank… Oh hell, that's a lot of work.

But wait a minute — do I really need to go through all this?

The three functors at the top of the diagrams (**) catch my attention. In Agda, they expand to the type $\mathsf{BT}^n_{1+k}\ (\mathsf{BT}^{1+k}_k\ (\mathsf{const}\ \top))\ xs$. An inhabitant of this type is a table of *blank* tables, so there is no choice of table entries; and moreover the structures of all the tables are completely determined by the indices… the type has a unique inhabitant! So the equation is actually trivial to prove: the two sides are forced by the type to construct the same table. And I don't need to look into the definitions of retabulate and blank at all!

Relieved, I start to work on the proof. The precise notion I need here is (mere) propositions [The Univalent Foundations Program 2013, Chapter 3]:

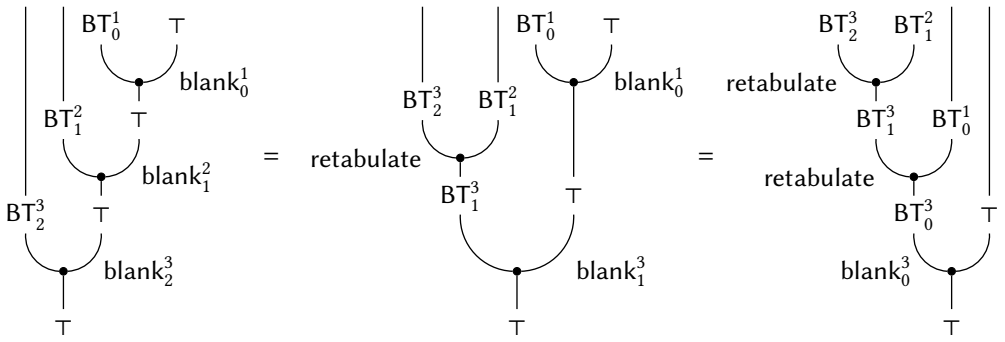


Fig. 8. Rewriting the table construction phase of td $s\ e\ g\ 3$ to that of bu $s\ e\ g\ 3$ using the rotation equation (∗∗).

$$\begin{aligned}&\mathsf{isProp}\ :\ \mathsf{Set} \to \mathsf{Set}\\ &\mathsf{isProp}\ a\ =\ \{x\ y\ :\ a\} \to x \equiv y\end{aligned}$$

The type $\mathsf{BT}^n_k\ p\ xs$ is propositional if the element types p are propositional — this is easy to prove by a straightforward double induction:

$$\mathsf{BT\text{-}isProp}\ :\ (\forall\ \{ys\} \to \mathsf{isProp}\ (p\ ys)) \to \mathsf{isProp}\ (\mathsf{BT}^n_k\ p\ xs)$$

And then the equation (∗∗) can be proved trivially by invoking BT-isProp twice:

$$\begin{aligned}&\mathsf{rotation}\ :\ \mathsf{retabulate}\ (\mathsf{blank}^n_k\ \mathbf{tt}) \equiv \mathsf{map}_{\mathsf{BT}}\ \mathsf{blank}^{1+k}_k\ (\mathsf{blank}^n_{1+k}\ \mathbf{tt})\\ &\mathsf{rotation}\ =\ \mathsf{BT\text{-}isProp}\ (\mathsf{BT\text{-}isProp}\ \mathbf{refl})\end{aligned}$$

The side conditions of retabulate and blank (omitted above) are all universally quantified. Usually they make proofs more complex, but not in this case because the proof doesn't look into any of the function definitions. As long as the type is blank nested tables, the two sides of an equation can be arbitrarily complicated, and I can still prove them equal just by using BT-isProp.

Wait, blank nested tables — aren't those what the construction phases of both algorithms produce?

I face-palm. It was a waste of time proving the rotation equation. The construction phases of both algorithms produce blank nested tables of the same type — $\mathsf{BT}^3_2\ (\mathsf{BT}^2_1\ (\mathsf{BT}^1_0\ (\mathsf{const}\ \top)))\ xs$ in the concrete examples I tried (Figures 4 and 6). So I can directly prove them equal using BT-isProp three times. There's no need to do any rotation.

Oh well, rotation is still interesting because it helps to explain how the two algorithms are related intensionally: they produce the same layers of tables but in opposite orders, and rotation helps to show how one order can be rewritten into the other (Figure 8). It's just that a rotation-based proof would be quite tedious, and I don't want to go through with that. A proof based on BT-isProp should be much simpler. Conceptually I've figured it all out: both algorithms have two phases modulo naturality; their table demolition phases are exactly the same, and their table construction phases are equal due to the BT-isProp reasoning. But the general proof is still going to take some work. If I want to stick to string diagrams, I'll need to transcribe the algorithms into inductively defined diagrams. Moreover, the BT-isProp reasoning is formally an induction (on the length of the input list), which needs to be worked out. And actually, compared with a diagrammatic but informal proof, I prefer a full Agda formalisation. That means I'll need to spell out a lot of detail, including functoriality and naturality rewriting (Figures 5 and 7). Whining, I finish the entire proof in Agda. But as usual, in the end there's a dopamine hit from seeing everything checked.

* * *

Still, I can't help feeling that I've neglected a fundamental aspect of the problem: why the bottom-up algorithm is more efficient. After making all the effort adopting dependent types and string diagrams, do these state-of-the-art languages help me say something about efficiency too?

String diagrams make it easier for me to see that the table construction phases of both algorithms produce the same layers of tables but in opposite orders. Only the order used by the bottom-up algorithm allows table construction and demolition to be interleaved, and consequently the algorithm needs no more than two layers of tables at any time. That's the crucial difference between the two algorithms. Now I need to figure out what the difference means algorithmically.

More specifically, why is it good to keep *two* layers of tables and not more?

When there are multiple layers of tables of type BT^n_k with $k < n$, meaning that the input list is split into proper sublists multiple times, all the final sublists will appear (as indices in the element types) in the entire nested table multiple times — that is, overlapping subproblems will appear. Therefore, when I use g to fill in a nested table, I'm invoking g to compute duplicate solutions for overlapping subproblems, which is what I want to avoid. More precisely, 'using g to fill in a nested table' means applying g under at least two layers, for example $\mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ g)\ :\ \mathsf{BT}^3_2\ (\mathsf{BT}^2_1\ (\mathsf{BT}^1_0\ s)) \rightrightarrows \mathsf{BT}^3_2\ (\mathsf{BT}^2_1\ s)$, where the result is at least two layers of tables, so there need to be at least *three* layers of tables (to which $\mathsf{map}_{\mathsf{BT}}\ (\mathsf{map}_{\mathsf{BT}}\ g)$ is applied) for duplicate solutions of overlapping subproblems to be recomputed. The bottom-up algorithm never gets to three layers of tables, and therefore avoids recomputing solutions for overlapping subproblems.

That reasoning doesn't sound too bad, although it's clear that there's much more to be done. The whole argument is still too informal and lacks detail. It's easy to poke holes in the reasoning — for example, if the input list has duplicate elements, then the bottom-up algorithm won't be able to entirely avoid duplicate solutions of overlapping subproblems. To fix this, the algorithm will need a redesign. And of course it's tempting to explore more problem-decomposing strategies beyond immediate sublists. Eventually I may arrive at something general about dynamic programming, which was what Richard wanted to work out in his paper.

All those are for another day, however. I've had enough fun today. Mostly, what I did was transcribe programs into new languages, but that helped me to reason in new ways, using more convenient tools to tackle Richard's problem.

I wish Richard was still around so that I could show all these to him. He would've liked the new languages and the new ways of reasoning.

* * * * * * * * * * *

AFTERWORD

This work is presented as a kind of 'Socratic monologue', recording the thought processes of a functional programmer as they solve a programming mystery. We were inspired by the science fiction novel *Project Hail Mary* by Andy Weir, where the narrative masterfully weaves together intuitive presentations of scientific knowledge and the protagonist's application of that knowledge to solve the problems they are facing. We envisaged to do something similar in this paper, although it ends up being not as leisurely and entertaining as Weir's novel, because we need to cover more technical detail, and there is very little action in our story apart from sitting in front of a computer and racking one's brains. However, compared to the traditional rational reconstruction of a finished piece of work, we believe that this format helps both the writer and the reader to focus on currently available clues and how to make progress based on those clues by recreating the experience of solving a mystery. In fact, our telling largely follows our actual development (tracing what `cd` does in Section 1.2, generalising `B` and `B′` to `BT` in Section 2.2, revising `ImmediateSublistInduction` in

Section 2.4, realising that the BT-isProp argument works more generally after proving rotation in Section 3.2, etc) — that is, this paper is 'based on a true story'.

The format also works well with various decisions regarding what to include in the paper, and what to omit. We put emphasis on intuitive explanations, and give formal definitions, theorems, and proofs only when necessary: we usually rely on intuitive reasoning to tackle a problem at first, and do not hurry to write things down formally. The supplementary Agda code provides the omitted formal detail though. We have striven to keep the paper fairly self-contained: the reader should be able to get a sense of the main ideas just from the intuitive explanations. But we do not intend this paper to be a tutorial on dependently typed programming in Agda or on category theory — the paper is best thought of as a companion to such tutorials or textbooks, giving a larger but not overly complicated example, and applying the abstract tools to that example. To make the paper more accessible, we have also resisted the temptation to generalise or to answer every question: for example, we do not generalise ImmediateSublistInduction for dynamic programming more broadly (as Bird [2008] attempted to do); we leave the question of whether the type of retabulate uniquely determines the extensional behaviour of its inhabitants as a conjecture (Section 2.3); and we avoid digressions into topics such as how data types like BT can be derived systematically (Section 2.2) and whether BT is a graded comonad (Section 2.4).

This work gives a dependently typed treatment of the sublists problem. The problem has been studied in other settings. It was one of the examples used by Bird and Hinze [2003] when studying a technique of function memoisation using trees of shared nodes, which they called *nexuses*. Bird [2008] went on to study top-down and bottom-up algorithms, where the sublists problem was the final example. To cover all the examples in the paper, Bird's generic bottom-up algorithm also employed a form of nexus, but it is not needed for the sublists problem and thus omitted in our work. Mu [2023] derived `cd` from the specification (∗) and proved the equality between `td` and `bu` using traditional equational reasoning. Neither Bird and Hinze [2003] nor Bird [2008] discussed applications of the sublists problem, but Mu [2023] observed that reduction to it is a standard technique in the algorithms community. None of these papers used dependent types.

The general message we want to deliver is that we can discover, explain, and prove things by writing them down in appropriate languages. More specifically, dependent types, category theory, and string diagrams are some of those languages, and they should be in the toolbox of the mathematically inclined functional programmer. In the case of dependent types, they can be expressive enough to replace traditional (equational) specifications and proofs. For example, in place of Mu's [2023] derivation, retabulate can be constructed by assigning it a type having sufficient information (Section 2.3), and the dependently typed `td` and `bu` can be proved equal simply by showing that they have the same, uniquely inhabited type (Section 2.5). This approach to program correctness and program equality is still under-explored, and has potential to reduce proof burdens drastically. As for category theory, even though we use it only in a lightweight manner, it still offers a somewhat useful abstraction for managing more complex (in our case, indexed) definitions as if they were simply typed (Section 3.1). More importantly, the categorical abstraction enables the use of string diagrams to simplify proofs about functoriality and naturality. These properties are only the simplest ones that string diagrams can handle — for other kinds of properties [Coecke and Kissinger 2017; Hinze and Marsden 2023] the proof simplification can be even more dramatic, although many of those properties are highly abstract. Our comparison between diagrammatic and traditional equational reasoning (Figures 4 and 5, and Figures 6 and 7) should be a good, albeit modest, demonstration of the power of string diagrams in a more practical, algorithmic scenario.

ACKNOWLEDGMENTS

We would like to thank Liang-Ting Chen for offering helpful suggestions about the development; Julie Summers, Royal Literary Fund Fellow at Kellogg College, Oxford, for commenting on an early draft; and Gene Tsai and Zhixuan Yang for proofreading a draft.

REFERENCES

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free: Parametricity for Dependent Types. *Journal of Functional Programming* 22, 2 (2012), 107–152. https://doi.org/10.1017/S0956796812000056

Richard Bird and Ralf Hinze. 2003. Functional Pearl: Trouble Shared is Trouble Halved. In *Workshop on Haskell*. ACM, 1–6. https://doi.org/10.1145/871895.871896

Richard S. Bird. 2008. Zippy Tabulations of Recursive Functions. In *International Conference on Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 5133)*. Springer, 92–109. https://doi.org/10.1007/978-3-540-70594-9_7

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *International Conference on Functional Programming (ICFP)*. ACM, 3–14. https://doi.org/10.1145/1863543.1863547

Bob Coecke and Aleks Kissinger. 2017. *Picturing Quantum Processes*. Cambridge University Press. https://doi.org/10.1017/9781316219317

Peter Dybjer. 1994. Inductive Families. *Formal Aspects of Computing* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects Via Grading. In *International Conference on Functional Programming (ICFP)*. ACM, 476–489. https://doi.org/10.1145/2951913.2951939

Ralf Hinze and Dan Marsden. 2023. *Introducing String Diagrams: The Art of Category Theory*. Cambridge University Press. https://doi.org/10.1017/9781009317825

Shin-Cheng Mu. 2023. Bottom-Up Computation Using Trees of Sublists (Functional Pearl). https://doi.org/10.48550/arXiv.2311.18528

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. https://homotopytypetheory.org/book/

Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. 2024. Internal and Observational Parametricity for Cubical Agda. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 8:1–32. https://doi.org/10.1145/3632850

Janis Voigtländer. 2009. Bidirectionalization for Free!. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 165–176. https://doi.org/10.1145/1480881.1480904