JFP, 12 pages, 2025.
doi:10.1017/xxxxx

FUNCTIONAL PEARL

Bottom-up computation using trees of sublists: A dependently typed approach

HSIANG-SHANG KO
Institute of Information Science, Academia Sinica, Taipei, Taiwan
(e-mail: joshko@iis.sinica.edu.tw)

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

Abstract

We revisit the problem of implementing a recursion scheme over immediate sublists studied by Mu (2024), and provide a dependently typed solution in Agda. The recursion scheme can be implemented as either a top-down algorithm, which has a straightforward definition but results in lots of re-computation, or a bottom-up algorithm, which has a puzzling definition but avoids re-computation. We show that the types can be made precise to guide and understand the developments of the algorithms. In particular, a precisely typed version of the key data structure (binomial trees) can be derived from the problem specification. The precise types also allow us to prove that the two algorithms are extensionally equal using parametricity. Despite apparent dissimilarities, our proof can be compared to Mu's equational proof, and be understood as a more economical version of the latter.

1 Introduction

The *immediate sublists* of a list xs are those lists obtained by removing exactly one element from xs. For example, the four immediate sublists of `"abcd"` are `"abc"`, `"abd"`, `"acd"`, and `"bcd"`. Mu (2024) considered the problem of computing a function h such that $h\,xs$ depends on values of h at all the immediate sublists of xs. More formally, given $f : \mathsf{List}\,B \rightarrow B$, compute $h : \mathsf{List}\,A \rightarrow B$ with such a top-down specification

$$h\,xs = f\,(\mathsf{map}\,h\,(\mathsf{subs}\,xs)) \tag{1.1}$$

where

$$\mathsf{subs} : \mathsf{List}\,A \rightarrow \mathsf{List}\,(\mathsf{List}\,A)$$

computes the immediate sublists of a list. Naively executing the specification results in lots of re-computation. See Fig. 1, for example: h `"ab"` is computed twice for h `"abc"` and h `"abd"`, and h `"ac"` twice for h `"abc"` and h `"acd"`.

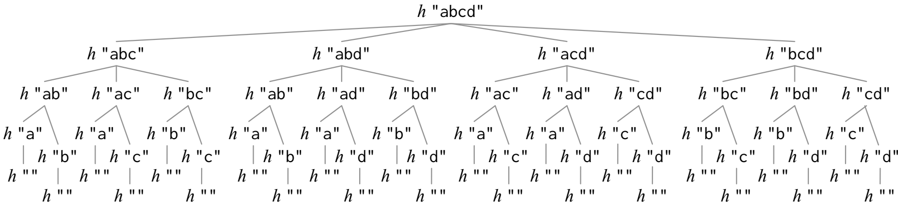

Fig. 1. Computing h "abcd" top-down.

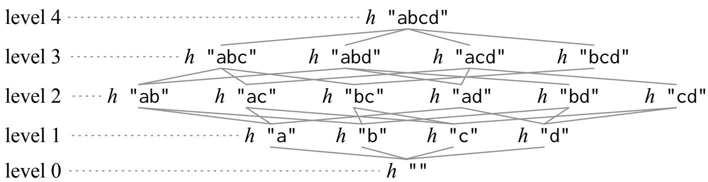

Fig. 2. Computing h "abcd" bottom-up.

The problem is derived from Bird's (2008) study of the relationship between top-down and bottom-up algorithms. A bottom-up strategy that avoids re-computation is shown in Fig. 2. Values of h on inputs of length n are stored in level n to be reused. Each level $n+1$ is computed from level n, until we reach the top. It may appear that this bottom-up strategy can be implemented by representing each level as a list, but this turns out to be impossible. Instead, Bird represented each level using a tip-valued binary tree defined by[1]

```
data BT (A : Set) : Set where
  tip : A            → BT A
  bin : BT A → BT A → BT A
```

equipped with (overloaded) functions map : $(A \to B) \to$ BT $A \to$ BT B and zipWith : $(A \to B \to C) \to$ BT $A \to$ BT $B \to$ BT C, respectively the mapping and zipping functions of BT, having expected definitions. Let t be a tree representing level n. To compute level $n+1$, we need a function upgrade : BT $A \to$ BT (List A), a natural transformation copying and rearranging elements in t, such that map f (upgrade t) represents level $n+1$. Bird suggested the following definition of upgrade (which is directly translated into Agda notation from Bird's Haskell program, and is not valid Agda):[2]

```
upgrade : BT A → BT (List A)
upgrade (bin (tip x) (tip y)) = tip (x :: y :: [])
upgrade (bin t       (tip y)) = bin (upgrade t) (map (_:: [ y ]) t)
upgrade (bin (tip x) u      ) = let tip xs = upgrade u in tip (x :: xs)
upgrade (bin t       u      ) = bin (upgrade t) (zipWith _::_ t (upgrade u))
```

If you feel puzzled by upgrade, so were we. Being the last example in the paper, Bird did not offer much explanation. The function upgrade is as concise as it is cryptic. The trees

[1] We use Agda in this pearl, while both Bird (2008) and Mu (2024) used Haskell; some of their definitions are quoted in this section but translated into Agda notation for consistency.

[2] The name upgrade was given by Mu (2024), while the same function was called cd by Bird (2008).

appear to obey some shape constraints — Bird called them *binomial trees*, hence the name BT, but neither the constraints nor how upgrade maintains them was explicitly stated.

Fascinated by the definition, Mu (2024) offered a specification of upgrade and a derivation of the definition, and then proved that the bottom-up algorithm is extensionally equal to the top-down specification/algorithm, all using traditional equational reasoning. As an interlude, Mu also showed (in his Section 4.3) a dependently typed version of upgrade, which used an indexed version of BT that encoded the shape constraint on binomial trees, although Mu did not explore the direction further. In this pearl, we go down the road not (thoroughly) taken and see how far it leads. In a dependently typed setting, can we derive the binomial trees by formalising in their types what we intend to compute? How effectively can the type information help us to implement the top-down and bottom-up algorithms correctly? And does the type information help us to prove that the two algorithms are extensionally equal?

2 The induction principle and its representations

Since we are computing a recursive function $h : \mathsf{List}\,A \to B$ given $f : \mathsf{List}\,B \to B$, we are dealing with a *recursion scheme* (Yang and Wu, 2022) of type

$$(\mathsf{List}\,B \to B) \to \mathsf{List}\,A \to B \tag{2.1}$$

In a dependently typed setting, recursion schemes become *elimination* or *induction principles*. Instead of ending type (2.1) with $\mathsf{List}\,A \to B$, we should aim for $(xs : \mathsf{List}\,A) \to P\,xs$ and make it an induction principle, of which $P : \mathsf{List}\,A \to \mathsf{Set}$ is the motive (McBride, 2002). Like all induction principles, the motive should be established and preserved in a way that follows the recursive structure of the computation: whenever P holds for all the immediate sublists of a list ys, it should hold for ys as well.

To define the induction principle formally, first we need to define immediate sublists — in fact we will just give a more general definition of sublists since we will need to refer to all of them during the course of the computation. Recall that an immediate sublist of xs is a list obtained by dropping one element from xs; more generally, a sublist can be obtained by dropping some number of elements. Element dropping can be written as an inductively defined relation:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Drop}^{\mathsf{R}} : \mathbb{N} \to \mathsf{List}\,A \to \mathsf{List}\,A \to \mathsf{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\textbf{return} & : & \qquad\qquad\qquad\qquad\quad \mathsf{Drop}^{\mathsf{R}}\ \textbf{zero}\ \ xs \qquad xs \\
\textbf{drop} & : & \mathsf{Drop}^{\mathsf{R}} \qquad\ \ n\ \ xs\,ys \to \mathsf{Drop}^{\mathsf{R}}\ (\textbf{suc}\,n)\ (x :: xs)\ ys \\
\textbf{keep} & : & \mathsf{Drop}^{\mathsf{R}}\ (\textbf{suc}\,n)\ xs\,ys \to \mathsf{Drop}^{\mathsf{R}}\ (\textbf{suc}\,n)\ (x :: xs)\ (x :: ys)
\end{array}
\end{array}
$$

Dropping $\textbf{zero}$ elements from any list xs is just returning xs itself; when dropping $\textbf{suc}\,n$ elements, the relation is defined only for non-empty lists $x :: xs$, and we may choose to drop x and continue to drop n elements from xs, or to keep x and continue to drop $\textbf{suc}\,n$ elements from xs. With the help of $\mathsf{Drop}^{\mathsf{R}}$ we can quantify over sublists; in particular, we can state that a motive P holds for all the immediate sublists zs of a list ys:

$$\forall\,\{zs\} \to \mathsf{Drop}^{\mathsf{R}}\ 1\ ys\ zs \to P\,zs \tag{2.2}$$

If this implies that P holds for any ys (as stated in the type of f below), then the induction principle concludes that P holds for all lists:

$$\begin{array}{l}\{A : \mathsf{Set}\}\ (P : \mathsf{List}\ A \to \mathsf{Set})\\ (f : \forall\,\{ys\} \to (\forall\,\{zs\} \to \mathsf{Drop}^{\mathsf{R}}\ 1\ ys\ zs \to P\ zs) \to P\ ys)\\ (xs : \mathsf{List}\ A) \to P\ xs\end{array}$$

Notice that the induction hypotheses are represented as a function of type (2.2), making the type of f higher-order, whereas type (2.1) uses a list, a first-order data structure. Below we derive an indexed data type $\mathsf{Drop}\ n\ P\ xs$ that represents universal quantification over all the sublists obtained by dropping n elements from xs; in particular, $\mathsf{Drop}\ 1\ P\ ys$ will be equivalent to type (2.2).

We start by (re)defining element dropping as a nondeterministic function:

$$\begin{array}{lll}\mathsf{drop} : \mathbb{N} \to \mathsf{List}\ A \to \mathsf{Nondet}\ (\mathsf{List}\ A) & & \\ \mathsf{drop}\ \ \mathbf{zero}\ \ xs & = & \mathsf{return}\ xs\\ \mathsf{drop}\ (\mathbf{suc}\ n)\ [\,] & = & \mathsf{mzero}\\ \mathsf{drop}\ (\mathbf{suc}\ n)\ (x :: xs) & = & \mathsf{mplus}\ (\mathsf{drop}\ n\ xs)\ (\mathsf{fmap}\ (x ::\_)\ (\mathsf{drop}\ (\mathbf{suc}\ n)\ xs))\end{array}$$

Nondet is a (relative) monad (Altenkirch et al., 2010) equipped with a fail operation ($\mathsf{mzero} : \mathsf{Nondet}\ A$) and nondeterministic choice ($\mathsf{mplus} : \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A$), and we choose the codensity representation (Filinski, 1994; Hinze, 2012)

$$\begin{array}{l}\mathsf{Nondet} : \mathsf{Set} \to \mathsf{Set}_\omega\\ \mathsf{Nondet}\ A = \forall\,\{\ell\}\ \{M : \mathsf{Set}\ \ell\} \to \{\!\{\mathsf{Monoid}\ M\}\!\} \to (A \to M) \to M\end{array}$$

where the result type M should be a monoid, defined as usual:

$$\begin{array}{l}\mathbf{record}\ \mathsf{Monoid}\ (M : \mathsf{Set}\ \ell) : \mathsf{Set}\ \ell\ \mathbf{where}\\ \quad \mathbf{constructor\ monoid}\\ \quad \mathbf{field}\\ \qquad \_\oplus\_ : M \to M \to M\\ \qquad \emptyset \quad\ : M\end{array}$$

(The monoid laws could be included but are not needed in our development.) If we expand the definitions of Nondet and its operations in drop, we get

$$\begin{array}{lll}\mathsf{drop} : \mathbb{N} \to \mathsf{List}\ A \to \{\!\{\mathsf{Monoid}\ M\}\!\} \to (\mathsf{List}\ A \to M) \to M & & \\ \mathsf{drop}\ \ \mathbf{zero}\ \ xs\ \ k & = & k\ xs\\ \mathsf{drop}\ (\mathbf{suc}\ n)\ [\,]\ \ k & = & \emptyset\\ \mathsf{drop}\ (\mathbf{suc}\ n)\ (x :: xs)\ k & = & \mathsf{drop}\ n\ xs\ k \oplus \mathsf{drop}\ (\mathbf{suc}\ n)\ xs\ (k \circ (x ::\_))\end{array}$$

which we can specialise to various forms. For example, we can specialise drop to compute all the sublists of a particular length using the list monad:

$$\begin{array}{l}\mathsf{drop}^{\mathsf{L}} : \mathbb{N} \to \mathsf{List}\ A \to \mathsf{List}\ (\mathsf{List}\ A)\\ \mathsf{drop}^{\mathsf{L}}\ n\ xs = \mathsf{drop}\ n\ xs\ \{\!\{\mathbf{monoid}\ \_{+\!\!+}\_\ [\,]\}\!\}\ (\_::[\,])\end{array}$$

In particular, $\mathsf{subs} = \mathsf{drop}^{\mathsf{L}}\ 1$ computes immediate sublists.

More interestingly, we can also specialise drop to compute types. For example, $\mathsf{Drop}^{\mathsf{R}}$ can alternatively be defined in continuation-passing style by

$$\mathsf{Drop}^{\mathsf{R}}\ n\ xs\ ys\ \cong\ \mathsf{drop}\ n\ xs\ \{\!\{\mathbf{monoid}\ \_\uplus\_\ \bot\}\!\}\ (\_\equiv ys)$$

where $\mathsf{drop}\ n\ xs\ \{\!\{\mathbf{monoid}\ \_\uplus\_\ \bot\}\!\}$ amounts to existential quantification over sublists. To obtain universal quantification, we supply the dual monoid:

$$\mathsf{Drop}\ n\ P\ xs\ \cong\ \mathsf{drop}\ n\ xs\ \{\!\{\mathbf{monoid}\ \_\times\_\ \top\}\!\}\ P$$

Rewriting the function definition as a data type definition (by turning each clause into a constructor), we get

$$\begin{array}{l}
\mathbf{data}\ \mathsf{Drop}\ :\ \mathbb{N} \to (\mathsf{List}\ A \to \mathsf{Set}) \to \mathsf{List}\ A \to \mathsf{Set}\ \mathbf{where} \\
\quad \mathbf{tip}\ :\ P\ xs \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathsf{Drop}\ \ \mathbf{zero}\ \ \ P\ xs \\
\quad \mathbf{nil}\ \ : \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{Drop}\ (\mathbf{suc}\ n)\ P\ [\,] \\
\quad \mathbf{bin}\ :\ \mathsf{Drop}\ n\ P\ xs \to \mathsf{Drop}\ (\mathbf{suc}\ n)\ (P \circ (x :: _))\ xs \to \mathsf{Drop}\ (\mathbf{suc}\ n)\ P\ (x :: xs)
\end{array}$$

which we will use to represent the induction hypotheses in the induction principle:

$$\begin{array}{l}
\mathsf{ImmediateSublistInduction}\ :\ \mathsf{Set}_1 \\
\mathsf{ImmediateSublistInduction}\ =\ \{A\ :\ \mathsf{Set}\}\ (P\ :\ \mathsf{List}\ A \to \mathsf{Set}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (f\ :\ \forall\,\{ys\} \to \mathsf{Drop}\ 1\ P\ ys \to P\ ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (xs\ :\ \mathsf{List}\ A) \to P\ xs
\end{array}$$

Note that Drop is an indexed version of BT (Section 1) that has an additional **nil** constructor. (We will see in Section 4 why it is beneficial to include **nil**.) Comparing type (2.1) with ImmediateSublistInduction, a potentially drastic change is that the list of induction hypotheses is replaced with a tree of type $\mathsf{Drop}\ 1\ P\ ys$ here. However, such a tree is actually list-shaped (constructed using **nil** and $\mathbf{bin} \circ \mathbf{tip}$), so ImmediateSublistInduction is really just a more informative version of type (2.1).

In the subsequent Sections 3 and 4 we will implement the top-down and bottom-up algorithms as programs of type ImmediateSublistInduction. These are fairly standard exercises in dependently typed programming (except perhaps for the upgrade function used in the bottom-up algorithm), and our implementations are by no means the only solutions.[3] The reader may want to try the exercises for themself, and is not obliged to go through the detail of our programs. We will prove that the two algorithms are extensionally equal in Section 5, to understand which it will *not* be necessary to know how the two algorithms are implemented.

3 The top-down algorithm

Equation (1.1) is essentially an executable definition of the top-down algorithm. This definition would not pass Agda's termination check though, because the immediate sublists in $\mathsf{subs}\ xs$ would not be recognised as structurally smaller than xs. One way to make termination evident is to make the length of xs explicit and perform induction on the length. The following function td does this by invoking td′, which takes as additional arguments a natural number l and an equality proof stating that the length of xs is l. The function td′ then performs induction on l and does the real work.

[3] Even the induction principle has alternative formulations, one of which was explored by Ko et al. (2025).

$$
\begin{aligned}
&\mathsf{td} : \mathsf{ImmediateSublistInduction}\\
&\mathsf{td}\ \{A\}\ P\ f\ xs\ =\ \mathsf{td'}\ (\mathsf{length}\ xs)\ xs\ \mathbf{refl}\\
&\quad \textbf{where}\\
&\qquad \mathsf{td'} : (l : \mathbb{N})\ (xs : \mathsf{List}\ A) \to \mathsf{length}\ xs \equiv l \to P\ xs\\
&\qquad \mathsf{td'}\ \ \mathbf{zero}\ \ []\ eq\ =\ f\ \mathbf{nil}\\
&\qquad \mathsf{td'}\ (\mathbf{suc}\ l)\ xs\ eq\ =\ f\ (\mathsf{map}\ (\lambda\ \{ys\} \to \mathsf{td'}\ l\ ys)\ (\mathsf{lenSubs}\ l\ xs\ eq))
\end{aligned}
$$

In the first case of $\mathsf{td'}$, where xs is $[]$, the final result is simply $f\ \mathbf{nil} : P\ []$. In the second case of $\mathsf{td'}$, where the length of xs is $\mathbf{suc}\ l$, the function subs is adapted to lenSubs, which constructs equality proofs that all the immediate sublists of xs have length l:

$$
\begin{aligned}
&\mathsf{lenSubs} : (l : \mathbb{N})\ (xs : \mathsf{List}\ A) \to \mathsf{length}\ xs \equiv \mathbf{suc}\ l\\
&\qquad\quad \to \mathsf{Drop}\ 1\ (\lambda\ ys \to \mathsf{length}\ ys \equiv l)\ xs
\end{aligned}
$$

With these equality proofs, we can then invoke $\mathsf{td'}$ inductively on every immediate sublist of xs with the help of the map function for Drop,

$$
\mathsf{map} : (\forall\ \{ys\} \to P\ ys \to Q\ ys) \to \mathsf{Drop}\ n\ P\ xs \to \mathsf{Drop}\ n\ Q\ xs
$$

and again use f to compute the final result.

4 The bottom-up algorithm

Given an input list xs, the bottom-up algorithm bu first creates a tree representing ‘level -1’ below the lattice in Fig. 2. This ‘basement’ level contains results for those sublists obtained by removing $\mathbf{suc}\ (\mathsf{length}\ xs)$ elements from xs; there are no such sublists, so the tree contains no elements, although the tree itself still exists (representing a proof of a vacuous universal quantification):

$$
\mathsf{base} : (xs : \mathsf{List}\ A) \to \mathsf{Drop}\ (\mathbf{suc}\ (\mathsf{length}\ xs))\ P\ xs
$$

The algorithm then enters a loop $\mathsf{bu'}$ and constructs each level of the lattice from bottom up, that is, a tree of type $\mathsf{Drop}\ n\ P\ xs$ for each n, with n decreasing:

$$
\begin{aligned}
&\mathsf{bu} : \mathsf{ImmediateSublistInduction}\\
&\mathsf{bu}\ P\ f\ =\ \mathsf{bu'}\ \_ \circ \mathsf{base}\\
&\quad \textbf{where}\\
&\qquad \mathsf{bu'} : (n : \mathbb{N}) \to \mathsf{Drop}\ n\ P\ xs \to P\ xs\\
&\qquad \mathsf{bu'}\ \ \mathbf{zero}\ \ =\ \mathsf{unTip}\\
&\qquad \mathsf{bu'}\ (\mathbf{suc}\ n)\ =\ \mathsf{bu'}\ n \circ \mathsf{map}\ f \circ \mathsf{retabulate}
\end{aligned}
$$

When the loop counter reaches **zero**, the tree contains exactly the result for xs, which we can extract using

$$
\begin{aligned}
&\mathsf{unTip} : \mathsf{Drop}\ \mathbf{zero}\ P\ xs \to P\ xs\\
&\mathsf{unTip}\ (\mathbf{tip}\ p)\ =\ p
\end{aligned}
$$

If the loop counter is $\mathbf{suc}\ n$, we create a new tree of type $\mathsf{Drop}\ n\ P\ xs$ that is one level higher than the current tree of type $\mathsf{Drop}\ (\mathbf{suc}\ n)\ P\ xs$. The type of the new tree says that it should contain results of type $P\ ys$ for all the sublists ys at the higher level. The retabulate function, which plays the same role as upgrade (Section 1), does half of the work by copying and rearranging the elements of the current tree to construct an intermediate tree representing the higher level:

```
retabulate : Drop (suc n) P xs → Drop n (Drop 1 P) xs
```

It assembles for each *ys* the induction hypotheses needed for computing *P ys* using *f* — that is, each element of the intermediate tree is a tree of type Drop 1 *P ys*. Then map *f* does the rest of the work and produces the desired new tree of type Drop *n P xs*, and we enter the next iteration.

To implement retabulate, just follow the types, and most of the program writes itself. (It is not particularly important to understand the program — in fact, any program works as long as it is type-correct.)

```
retabulate : Drop (suc n) P xs → Drop n (Drop 1 P) xs
retabulate     nil                         = underground
retabulate t@(bin    (tip _)     _  )  = tip t
retabulate    (bin     nil        nil)  = bin underground nil
retabulate    (bin t@(bin _ _) u   )  = bin (retabulate t)
                                              (zipWith (bin ∘ tip) t (retabulate u))
```

The auxiliary function underground is defined by

```
underground : Drop n (Drop 1 P) []
underground {n = zero  } = tip nil
underground {n = suc _} = nil
```

(It analyses the implicit argument *n*, which therefore needs to be present at runtime, so retabulate actually requires more information than the input tree to execute, unlike upgrade.) The last clause of retabulate is the most difficult one to conceive, but can be copied exactly from the last clause of upgrade except that the list cons is replaced by the cons function **bin** ∘ **tip** for Drop 1 trees (which, as mentioned in Section 2, are list-shaped), and the type of zipWith needs to be updated:

```
zipWith : (∀ {ys} → P ys → Q ys → R ys)
              → Drop n P xs → Drop n Q xs → Drop n R xs
```

It is a fruitful exercise to trace the constraints assumed and established throughout the construction (especially the last clause), which are now manifested as type information — see Ko et al.'s (2025) Section 2.3 for a solution to a similar version of the exercise.

The first and third clauses of retabulate involve **nil**, and have no counterparts in upgrade. Drop trees containing **nil** correspond to empty levels below the lattice in Fig. 2 (which result from dropping too many elements from the input list). Mu (2024) avoided dealing with such empty levels by imposing conditions throughout his development — for example, see Mu's Section 4.3 and Appendix B for a version of the program (which is named up there) with conditions. We avoid those somewhat tedious conditions by including **nil** in Drop to represent the empty levels, and in exchange need to deal with these levels, which are easy to deal with though.

5 Extensional equality between the two algorithms

Now we have two different implementations of $\mathsf{ImmediateSublistInduction}$, namely td and bu. How do we prove that they compute the same results?

Actually, is it possible to write programs of type $\mathsf{ImmediateSublistInduction}$ to compute different results in Agda? Since $\mathsf{ImmediateSublistInduction}$ is parametric in P, intuitively a program of this type can only compute a result of type $P\,xs$ using f, and moreover, the index xs determines how f needs to be applied to arrive at that result (to compute which f needs to be applied to sub-results of type $P\,ys$ for all the immediate sublists ys of xs, and all the sub-results can only be computed using f, and so on). So td and bu have to compute the same results simply because they have the same —and special— type!

To prove this formally, we use parametricity. The following is the unary parametricity statement of $\mathsf{ImmediateSublistInduction}$ with respect to P (whereas A is treated merely as a fixed parameter), derived using Bernardy et al.'s (2012) translation:

$$
\begin{array}{l}
\mathsf{UnaryParametricity} : \mathsf{ImmediateSublistInduction} \to \mathsf{Set}_1 \\
\mathsf{UnaryParametricity}\ ind\ = \\
\quad \{A : \mathsf{Set}\}\ \{P\ : \mathsf{List}\,A \to \mathsf{Set}\} \qquad\qquad\qquad (Q\ : \forall\,\{ys\} \to P\,ys \to \mathsf{Set}) \\
\qquad\qquad \{f\ \ : \forall\,\{ys\} \to \mathsf{Drop}\;1\;P\,ys \to P\,ys\}\ (g\ \ : \forall\,\{ys\}\,\{ps : \mathsf{Drop}\;1\;P\,ys\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathsf{All}\,Q\,ps \to Q\,(f\,ps)) \\
\qquad\qquad \{xs : \mathsf{List}\,A\} \to Q\,(ind\,P\,f\,xs)
\end{array}
$$

Unary parametricity can be understood in terms of invariant preservation: state an invariant Q on values of type of the form $P\,ys$, provide a proof g that Q is preserved by f, and then the results computed by $ind\,P\,f$ will satisfy Q. In the type of g, we need an auxiliary definition to formulate the premise that Q is satisfied by all the elements in a Drop tree:

$$
\begin{array}{l}
\mathsf{All} : (\forall\,\{ys\} \to P\,ys \to \mathsf{Set}) \to \mathsf{Drop}\,n\,P\,xs \to \mathsf{Set} \\
\mathsf{All}\,Q\,(\mathbf{tip}\,p) \quad = Q\,p \\
\mathsf{All}\,Q\ \ \mathbf{nil} \qquad = \top \\
\mathsf{All}\,Q\,(\mathbf{bin}\,t\,u)\ = \mathsf{All}\,Q\,t \times \mathsf{All}\,Q\,u
\end{array}
$$

Now the extensional equality between td and bu follows fairly straightforwardly from a proof of bu's unary parametricity

$$
\mathsf{buParam} : \mathsf{UnaryParametricity}\ \mathsf{bu}
$$

which can be obtained for free, for example using Bernardy et al.'s translation again or internal parametricity (Van Muylder et al., 2024). Given P and f, we invoke the parametricity proof with the invariant $\lambda\,\{ys\}\,p \to \mathsf{td}\,P\,f\,ys \equiv p$ saying that any $p : P\,ys$ can only be the result computed by $\mathsf{td}\,P\,f\,ys$ (corresponding to our intuition above), and supply an argument tdComp proving that f preserves the invariant, which takes only a small amount of work:

$$
\mathsf{buParam}\ (\lambda\,\{ys\}\,p \to \mathsf{td}\,P\,f\,ys \equiv p)\ \mathsf{tdComp} : \{xs : \mathsf{List}\,A\} \to \mathsf{td}\,P\,f\,xs \equiv \mathsf{bu}\,P\,f\,xs \tag{5.1}
$$

We have got the equality we want. But if we look at the argument tdComp in more detail, we will see that we can refactor the proof to gain a bit more structure and generality. The instantiated type of tdComp is

$$\forall\, \{ys\}\, \{ps : \mathsf{Drop}\ 1\ P\ ys\} \to \mathsf{All}\ (\lambda\, \{zs\}\, p \to \mathsf{td}\ P\ f\ zs \equiv p)\ ps \to \mathsf{td}\ P\ f\ ys \equiv f\ ps$$

This says that computing $\mathsf{td}\ P\ f\ ys$ is the same as applying f to ps where every p in ps is already a result computed by $\mathsf{td}\ P\ f$ — this has the same computational content as equation (1.1), and is a formulation of the *computation rule* of ImmediateSublistInduction, satisfied by td! (That is, computation rules can be formulated as a form of invariant preservation.) Therefore we can formulate the computation rule for any implementation *ind* of ImmediateSublistInduction,

$$
\begin{aligned}
&\mathsf{ComputationRule} : \mathsf{ImmediateSublistInduction} \to \mathsf{Set}_1\\
&\mathsf{ComputationRule}\ ind\ =\\
&\quad \{A : \mathsf{Set}\}\, \{P : \mathsf{List}\ A \to \mathsf{Set}\}\, \{f : \forall\, \{ys\} \to \mathsf{Drop}\ 1\ P\ ys \to P\ ys\}\, \{xs : \mathsf{List}\ A\}\\
&\quad \{ps : \mathsf{Drop}\ 1\ P\ xs\} \to \mathsf{All}\ (\lambda\, \{ys\}\, p \to ind\ P\ f\ ys \equiv p)\ ps \to ind\ P\ f\ xs \equiv f\ ps
\end{aligned}
$$

and then generalise equality (5.1) to a theorem that equates the extensional behaviour of any two implementations of the induction principle, where one implementation satisfies the computation rule and the other satisfies unary parametricity:

$$
\begin{aligned}
&\mathsf{uniqueness} :\\
&\qquad (ind\ ind' : \mathsf{ImmediateSublistInduction})\\
&\quad \to \mathsf{ComputationRule}\ ind \to \mathsf{UnaryParametricity}\ ind'\\
&\quad \to \{A : \mathsf{Set}\}\ (P : \mathsf{List}\ A \to \mathsf{Set})\ (f : \forall\, \{ys\} \to \mathsf{Drop}\ 1\ P\ ys \to P\ ys)\ (xs : \mathsf{List}\ A)\\
&\quad \to ind\ P\ f\ xs \equiv ind'\ P\ f\ xs\\
&\mathsf{uniqueness}\ ind\ ind'\ comp\ param'\ P\ f\ xs\ =\ param'\ (\lambda\, \{ys\}\, p \to ind\ P\ f\ ys \equiv p)\ comp
\end{aligned}
$$

6 Methodological discussions

6.1 Proving uniqueness of induction principle implementations from parametricity

Usually, we prove two implementations *ind* and *ind′* of an induction principle to be equal assuming that both *ind* and *ind′* satisfy the set of computation rules coming with the induction principle. For example, for ImmediateSublistInduction we can prove

$$
\begin{aligned}
&\quad (ind\ ind' : \mathsf{ImmediateSublistInduction})\\
&\to \mathsf{ComputationRule}\ ind \to \mathsf{ComputationRule}\ ind'\\
&\to \{A : \mathsf{Set}\}\ (P : \mathsf{List}\ A \to \mathsf{Set})\ (f : \forall\, \{ys\} \to \mathsf{Drop}\ 1\ P\ ys \to P\ ys)\ (xs : \mathsf{List}\ A)\\
&\to ind\ P\ f\ xs \equiv ind'\ P\ f\ xs
\end{aligned}
$$

The uniqueness theorem in Section 5 demonstrates (in terms of ImmediateSublistInduction) that we can alternatively assume that one implementation, say *ind′*, satisfies unary parametricity instead, and we will still have a proof. This is useful when *ind* can be easily proved to satisfy the set of computation rules whereas *ind′* cannot. In our case, even though our td in Section 3 does not satisfy the computation rule definitionally (because it performs a different form of induction on the length of the input list, to make termination evident to Agda), a proof of ComputationRule td still takes only a small amount of work. It would be more difficult to prove that bu satisfies the computation rule, whereas a parametricity proof for bu is always mechanical —if not automatic— to derive, so switching to the latter greatly reduces the proof burden. In general, this trick may be useful for porting recursion schemes or inventing efficient implementations of induction principles in a dependently typed setting.

6.2 Establishing invariants using indexed data types and parametricity

Mu (2024) took pains to prove that the two algorithms are extensionally equal, whereas in this pearl the equality seems to follow almost for free from parametricity. The trick is that the necessary properties are either enforced by types or established by parametricity. Recall that in Section 1 the top-down algorithm is computed by $h : \mathsf{List}\,A \to B$ given $f : \mathsf{List}\,B \to B$. The main property Mu needed was his Lemma 1, which can be roughly translated into our setting as

$$(\mathsf{map}\,f \circ \mathsf{upgrade})^{k}\ (\mathsf{base}'\ xs) \ = \ \mathsf{map}\,h\,(\mathsf{drop}^{\mathsf{BT}}\ (\mathbf{suc}\,(\mathsf{length}\,xs) - k)\ xs) \tag{6.1}$$

This is an old-school way of saying that the bottom-up algorithm maintains an invariant. The left-hand side is the value computed by the bottom-up algorithm after k iterations: xs is the initial input; base' plays a similar role as base in Section 4 and prepares an initial tree, on which $\mathsf{map}\,f \circ \mathsf{upgrade}$, the loop body of the bottom-up algorithm, is performed k times. The invariant is that the value must equal the right-hand side: a tree containing values $h\,ys$ for all the sublists ys of xs having k elements — that is, those sublists obtained by dropping $\mathbf{suc}\,(\mathsf{length}\,xs) - k$ elements from xs; this tree has the same shape as the one built by $\mathsf{drop}^{\mathsf{BT}} : \mathbb{N} \to \mathsf{List}\,A \to \mathsf{BT}\,(\mathsf{List}\,A)$, which also determines the position of each $h\,ys$ in the tree. By contrast, this pearl uses (i) the indexed data type Drop to enforce tree shapes and sublist positions and (ii) parametricity to establish that the trees contain values of td.

Using indexed data types to enforce shape constraints is a well known technique, which in particular was briefly employed by Mu (2024, Section 4.3). But program specifications are often not just about shapes. For example, to prove equation (6.1), Mu gave a specification of $\mathsf{upgrade}$, from which the derivation of $\mathsf{upgrade}$'s definition was the main challenge for Mu:

$$\mathsf{upgrade}\,(\mathsf{drop}^{\mathsf{BT}}\ (\mathbf{suc}\,k)\ xs) \ = \ \mathsf{map}\,\mathsf{subs}\,(\mathsf{drop}^{\mathsf{BT}}\ k\,xs)$$

Shape-wise, this equation says that given a tree having the shape computed by $\mathsf{drop}^{\mathsf{BT}}\ (\mathbf{suc}\,k)\ xs$, $\mathsf{upgrade}$ produces a tree having the shape computed by $\mathsf{drop}^{\mathsf{BT}}\ k\,xs$. But the equation also specifies how the natural transformation should rearrange the tree elements by saying what it should do in particular to the trees of sublists computed by $\mathsf{drop}^{\mathsf{BT}}\ (\mathbf{suc}\,k)\ xs$. This pearl demonstrates that it is possible to go beyond shapes and encode the full specification in the type of $\mathsf{retabulate}$ (Section 4) using the indexed data type Drop. The key is that the element types in Drop trees are indexed by sublists and therefore distinct in general, so the elements need to be placed at the right positions to be type-correct. Subsequently, the definition of $\mathsf{retabulate}$ can be developed in a type-driven manner, which is more economical than Mu's equational derivation.

Equation (6.1) also says that each iteration of the bottom-up algorithm produces the same results as those computed by h, and Mu (2024) proved equation (6.1) by induction on k. What is the relationship between Mu's inductive proof and ours based on

UnaryParametricity bu (Section 5)? Mu's induction on k coincides with the looping structure of the bottom-up algorithm. On the other hand, while UnaryParametricity could in principle be proved mechanically once-and-for-all for all functions having the right type, if one had to prove UnaryParametricity bu manually, the proof would also follow the structure of bu. Therefore the proof of bu's unary parametricity would essentially be the proof of equation (6.1) generalised to all invariants. Finally, note that this opportunity to invoke parametricity emerges because we switch to dependent types and reformulate the recursion scheme as an induction principle: knowing that a result p has the indexed type $P\,ys$ allows us to state the invariant $Q\,\{ys\}\,p\ =\ \mathsf{td}\,P\,f\,ys \equiv p$, whereas the non-indexed result type B in type (2.1) does not provide enough information for stating that.

Acknowledgements

Zhixuan Yang engaged in several discussions about induction principles, computation rules, and parametricity, leading to the current presentation of the parametricity-based proof. He also pointed out how Nondet is an instance of the codensity representation except that a dinaturality condition is omitted (Hinze, 2012). At the IFIP WG 2.1 meeting in April 2024, James McKinna suggested defining retabulate on the higher-order representation (2.2) instead. This definition of retabulate is extremely simple, but does not copy and reuse results on sublists, and therefore does not help to avoid re-computation. However, this perspective does make the relationship between binomial trees and proofs of universal quantification clear, and leads to the inclusion of the **nil** constructor in Drop (which helps to simplify our definition of retabulate). At the same meeting, Wouter Swierstra asked whether lists could be used instead of vectors in a previous definition of binomial trees (Ko et al., 2025). There the definition of immediate sublists depends on the length of the input list, so it is more convenient to use vectors. However, this question leads us to consider a definition of immediate sublists that does not depend on list length, and ultimately to the simpler definition of Drop (which uses lists instead of vectors).Yen-Hao Liu previewed and provided feedback on a draft. We would like to thank all of them.

The two authors are supported by the National Science and Technology Council of Taiwan under grant numbers NSTC 112-2221-E-001-003-MY3 and NSTC 113-2221-E-001-020-MY2 respectively.

References

Altenkirch, T., Chapman, J. & Uustalu, T. (2010) Monads need not be endofunctors. International Conference on Foundations of Software Science and Computational Structures (FoSSaCS). Springer. pp. 297–311.

Bernardy, J.-P., Jansson, P. & Paterson, R. (2012) Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*. **22**(2), 107–152.

Bird, R. S. (2008) Zippy tabulations of recursive functions. International Conference on Mathematics of Program Construction (MPC). Springer. pp. 92–109.

Filinski, A. (1994) Representing monads. Symposium on Principles of Programming Languages (POPL). ACM. pp. 446–457.

Hinze, R. (2012) Kan extensions for program optimisation — or: Art and Dan explain an old trick. International Conference on Mathematics of Program Construction (MPC). Springer. pp. 324–362.

Ko, H.-S., Mu, S.-C. & Gibbons, J. (2025) Binomial tabulation: A short story. arXiv:2503.04001.

McBride, C. (2002) Elimination with a motive. International Workshop on Types for Proofs and Programs (TYPES). Springer. pp. 197–216.

Mu, S.-C. (2024) Bottom-up computation using trees of sublists. *Journal of Functional Programming*. **34**, e14:1–16.

Van Muylder, A., Nuyts, A. & Devriese, D. (2024) Internal and observational parametricity for Cubical Agda. *Proceedings of the ACM on Programming Languages*. **8**(POPL), 8:1–32.

Yang, Z. & Wu, N. (2022) Fantastic morphisms and where to find them: A guide to recursion schemes. International Conference on Mathematics of Program Construction (MPC). Springer. pp. 222–267.