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

HSIANG-SHANG KO, Institute of Information Science, Academia Sinica, Taiwan
SHIN-CHENG MU, Institute of Information Science, Academia Sinica, Taiwan

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 Mu's proof.

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 $h : \mathsf{List}\ A \to B$ such that $h\ xs$ depends on values of h at all the immediate sublists of *xs*.[1] More formally, assuming that the function

$$\mathsf{subs} : \mathsf{List}\ A \to \mathsf{List}\ (\mathsf{List}\ A)$$

computes the immediate sublists of a list, to compute $h\ xs$ we can decompose *xs* into $\mathsf{subs}\ xs : \mathsf{List}\ (\mathsf{List}\ A)$, then apply $\mathsf{map}\ h : \mathsf{List}\ (\mathsf{List}\ A) \to \mathsf{List}\ B$ recursively to get a list of sub-results for the immediate sublists of *xs*, and finally invoke a given function $f : \mathsf{List}\ B \to B$ to combine the sub-results into a result for *xs*. That is, h is specified by the equation

$$h\ xs = f\ (\mathsf{map}\ h\ (\mathsf{subs}\ xs)) \tag{1}$$

The problem is derived from Bird's [2008] study of the relationship between top-down and bottom-up algorithms. Equation (1) expresses a top-down strategy, which, if executed directly, results in lots of re-computation. See Figure 1, for example: h "ab" is computed twice for h "abc" and h "abd", and h "ac" twice for h "abc" and h "acd". A bottom-up strategy that avoids re-computation is shown in Figure 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. Bird represented each level using a tip-valued binary tree defined by[2]

$$\begin{array}{l} \textbf{data}\ \mathsf{BT}\ (A : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\ \quad \textbf{tip} : A \to \mathsf{BT}\ A \\ \quad \textbf{bin} : \mathsf{BT}\ A \to \mathsf{BT}\ A \to \mathsf{BT}\ A \end{array}$$

equipped with (overloaded) functions

$$\begin{array}{l} \mathsf{map} : (A \to B) \to \mathsf{BT}\ A \to \mathsf{BT}\ B \\ \mathsf{zipWith} : (A \to B \to C) \to \mathsf{BT}\ A \to \mathsf{BT}\ B \to \mathsf{BT}\ C \end{array}$$

[1]This form of recursive computation arises when, for example, solving an optimisation problem over the permutations of a list and decomposing the problem recursively by considering which element should be the first one in the output permutation. Mu [2024] mentioned some more examples near the end of his Section 1.

[2]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.

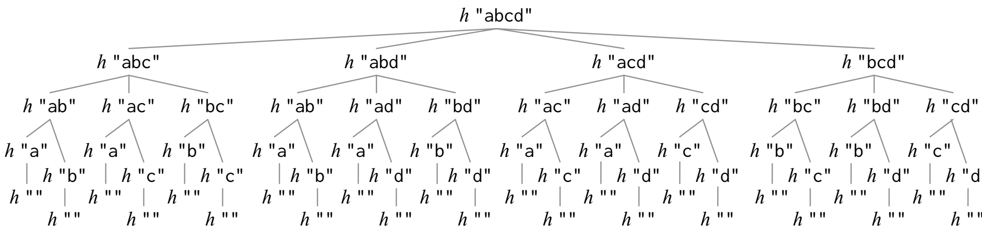

Fig. 1. Computing h "abcd" top-down.

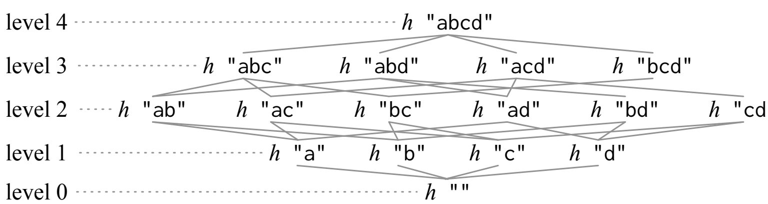

Fig. 2. Computing h "abcd" bottom-up.

respectively the mapping and zipping functions of BT that one would expect. Let t be a tree representing level n. To compute level $n + 1$, we need a function upgrade : BT $A \rightarrow$ 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:[3]

```
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 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?

[3]This definition of upgrade is translated into Agda notation from Bird's Haskell program; the name upgrade was given by Mu [2024], while the same function was called cd by Bird [2008]. To be clear, the definition is not valid Agda: the two sets of patterns at the top level and in the **let**-expression fail the coverage check; moreover, Agda does not actually allow pattern matching with data constructors in **let**-expressions.

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

$$(\mathsf{List}\ B \to B) \to \mathsf{List}\ A \to B \tag{2}$$

In a dependently typed setting, recursion schemes become *elimination* or *induction principles*, and we will refine type (2) to an induction principle. The first step is refining $\mathsf{List}\ A \to B$ to $(xs : \mathsf{List}\ A) \to P\ xs$ where $P : \mathsf{List}\ A \to \mathsf{Set}$ is the induction *motive* [McBride 2002]. And then we should refine the premise $\mathsf{List}\ B \to B$ to an induction *method*, whose type states that the motive should be established and preserved in a way that follows the recursive structure of the computation — or more specifically, whenever P holds for all the immediate sublists of a list ys, P should hold for ys as well.

To write down the induction principle formally (in particular the type of the induction method), first we need to define immediate sublists — in fact we will give a more general definition of sublists, following Mu's [2024] insight that we will need to refer to all of the sublists 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:

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

Dropping **zero** elements from any list xs is just returning xs itself; when dropping $\mathbf{suc}\ n$ elements, the relation is defined only for nonempty 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 $\mathbf{suc}\ n$ 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:

$$\forall\ \{zs\} \to \mathsf{Drop}^{\mathsf{R}}\ 1\ ys\ zs \to P\ zs \tag{3}$$

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:

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

This is one possible refinement of type (2). But notice that the induction hypotheses are represented as a function of type (3), whereas type (2) uses $\mathsf{List}\ B$, 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 (3).

We start by (re)defining element dropping as a familiar nondeterministic function. Suppose that Nondet is a nondeterminism monad equipped with a fail operation mzero and nondeterministic choice mplus:

$$
\begin{array}{l}
\mathsf{mzero} : \mathsf{Nondet}\ A \\
\mathsf{mplus} : \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A
\end{array}
$$

Then element dropping can be defined monadically by

$$
\begin{aligned}
&\mathsf{drop} \;:\; \mathbb{N} \to \mathsf{List}\ A \to \mathsf{Nondet}\ (\mathsf{List}\ A)\\
&\mathsf{drop}\ \ \mathbf{zero}\ \ \ xs \qquad\quad = \mathsf{return}\ xs\\
&\mathsf{drop}\ (\mathbf{suc}\ n)\ []\qquad\quad = \mathsf{mzero}\\
&\mathsf{drop}\ (\mathbf{suc}\ n)\ (x :: xs) \;=\; \mathsf{mplus}\ (\mathsf{drop}\ n\ xs)\ (\mathsf{drop}\ (\mathbf{suc}\ n)\ xs \gg\!= \lambda\ ys \to \mathsf{return}\ (x :: ys))
\end{aligned}
$$

To instantiate Nondet so that drop can compute types (and allow us to derive the indexed data type Drop eventually), one way is to instantiate Nondet to a continuation monad,[4]

$$
\begin{aligned}
&\mathsf{Nondet} \;:\; \mathsf{Set} \to \mathsf{Set}_\omega\\
&\mathsf{Nondet}\ A \;=\; \forall\ \{\ell\}\ \{M \;:\; \mathsf{Set}\ \ell\} \to \{\!\{\ \mathsf{Monoid}\ M\ \}\!\} \to (A \to M) \to M
\end{aligned}
$$

with the standard definitions of return and bind:

$$
\begin{aligned}
&\mathsf{return} \;:\; A \to \mathsf{Nondet}\ A\\
&\mathsf{return}\ x \;=\; \lambda\ k \to k\ x
\end{aligned}
$$

$$
\begin{aligned}
&\_\!\gg\!=\!\_ \;:\; \mathsf{Nondet}\ A \to (A \to \mathsf{Nondet}\ B) \to \mathsf{Nondet}\ B\\
&mx \gg\!= f \;=\; \lambda\ k \to mx\ (\lambda\ x \to f\ x\ k)
\end{aligned}
$$

In the definition of Nondet, there is an additional $\{\!\{\ \mathsf{Monoid}\ M\ \}\!\}$ argument (wrapped in double brackets); this is an instance argument [Devriese and Piessens 2011], which is comparable to type classes in Haskell. In effect, the result type M is required to support the usual monoid operations:

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

With these operations on M, we can implement mzero by ignoring the current continuation and returning $\emptyset$,

$$
\begin{aligned}
&\mathsf{mzero} \;:\; \mathsf{Nondet}\ A\\
&\mathsf{mzero} \;=\; \lambda\ k \to \emptyset
\end{aligned}
$$

and implement mplus by running its two branches (both with the current continuation) and merging the results using the $\_\oplus\_$ operation:

$$
\begin{aligned}
&\mathsf{mplus} \;:\; \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A \to \mathsf{Nondet}\ A\\
&\mathsf{mplus}\ mx\ my \;=\; \lambda\ k \to mx\ k \oplus my\ k
\end{aligned}
$$

(Some readers have probably recognised that this is essentially the technique of representing monads in continuation-passing style [Filinski 1994; Hinze 2012].) If we expand these definitions in drop, we get

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

which we can instantiate to various forms. For example, we can instantiate drop to compute all the sublists of a particular length, supplying the list monoid as the result type:

[4] Technically, Nondet is not an endofunctor and thus not a monad, but it is a relative monad [Altenkirch et al. 2010], for which return and bind still make sense.

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

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

More importantly, we want to instantiate 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 $\mathsf{drop}\ n\ xs\ \{\!\{\ \mathbf{monoid}\ \_\uplus\_\ \bot\ \}\!\} \;:\; (\mathsf{List}\ A \to \mathsf{Set}) \to \mathsf{Set}$ amounts to existential quantification over sublists: an input predicate $P \;:\; \mathsf{List}\ A \to \mathsf{Set}$ is only required to hold in one of the branches at every nondeterministic choice (since we instantiate the monoid operation $\_\oplus\_$ to disjunction $\_\uplus\_$), so eventually P is only required to hold for one sublist. To obtain universal quantification, we supply the conjunction $\mathbf{monoid}\ \_\times\_\ \top$, requiring P to hold for all the branches and thus all the sublists:

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

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

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

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

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

Comparing ImmediateSublistInduction with type (2), 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 **bin** ∘ **tip**), so ImmediateSublistInduction is a faithful refinement of type (2). Moreover, we will see that Drop is a refinement of BT (Section 1) with an additional **nil** constructor — we will reimplement the upgrade function used in the bottom-up algorithm to operate on Drop trees, and the same computation patterns will emerge. The refinement from BT to Drop gives us a better idea of why Bird [2008] needed to use BT: paths in BT/Drop trees correspond to computation of sublists of a particular length, so working with BT/Drop trees allows us to figure out which sublist each element in the trees is associated with and put the elements at the right places; the associations are only implicitly assumed in BT, whereas in Drop they are explicitly recorded in the element types of the form $P\ xs$.

In 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 upgrade), and our implementations are by no means the only solutions.[5] The reader may want to try the exercises for themself, and is not obliged to go through the details of our programs. We will prove that the two algorithms are extensionally equal in Section 5; the proof will not depend on the implementation details of the two algorithms, but only on their shared dependent type.

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

3 THE TOP-DOWN ALGORITHM

Equation (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 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.

```
td : ImmediateSublistInduction
td {A} P f xs = td′ (length xs) xs refl
  where    -- lenSubs to be defined later
    td′ : (l : ℕ) (xs : List A) → length xs ≡ l → P xs
    td′ zero    [] eq = f nil
    td′ (suc l) xs eq = f (map (λ {ys} → td′ l ys) (lenSubs l xs eq))
```

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

```
lenSubs : (l : ℕ) (xs : List A) → length xs ≡ suc l
        → Drop 1 (λ ys → length ys ≡ l) xs
```

With these equality proofs, we can then invoke td′ inductively on every immediate sublist of *xs* with the help of the map function for Drop,

```
map : (∀ {ys} → P ys → Q ys) → Drop n P xs → 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 Figure 2. This 'basement' level contains results for those sublists obtained by removing **suc** (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, or more specifically, a proof that all the branches in the nondeterministic computation of drop end with failure):

```
base : (xs : List A) → Drop (suc (length xs)) P xs
```

The algorithm then enters a loop bu′ and constructs each level of the lattice from bottom up, that is, a tree of type Drop n P *xs* for each n, with n decreasing:

```
bu : ImmediateSublistInduction
bu P f = bu′ _ ∘ base
  where    -- unTip and retabulate to be defined later
    bu′ : (n : ℕ) → Drop n P xs → P xs
    bu′ zero    = unTip
    bu′ (suc n) = bu′ n ∘ map f ∘ retabulate
```

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

```
unTip : Drop zero P xs → P xs
unTip (tip p) = p
```

If the loop counter is **suc** n, we create a new tree of type Drop n P xs that is one level higher than the current tree of type Drop (**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, follow the types, and most of the program writes itself. We will not go through the construction of the program in detail because we only aim to show the correctness of bu, which depends only on the type of retabulate.

```
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 near the end of 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 Figure 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 easier to deal with than the conditions though.

5 EXTENSIONAL EQUALITY BETWEEN THE TWO ALGORITHMS

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

Actually, is it possible to write programs of type ImmediateSublistInduction to compute different results in Agda? It may help to consider a simpler example: induction on natural numbers,

$$\begin{array}{l} \mathbb{N}\mathsf{Induction} \;:\; \mathsf{Set}_1 \\ \mathbb{N}\mathsf{Induction} \;=\; (P \;:\; \mathbb{N} \to \mathsf{Set}) \\ \qquad\qquad\qquad (pz \;:\; P\;\mathbf{zero})\;(ps \;:\; \forall\,\{n\} \to P\;n \to P\;(\mathbf{suc}\;n)) \\ \qquad\qquad\qquad (n \;:\; \mathbb{N}) \to P\;n \end{array}$$

which has a standard implementation:

$$\begin{array}{l} \mathsf{ind}\mathbb{N} \;:\; \mathbb{N}\mathsf{Induction} \\ \mathsf{ind}\mathbb{N}\;P\;pz\;ps\;\;\mathbf{zero}\quad\;\; = \;pz \\ \mathsf{ind}\mathbb{N}\;P\;pz\;ps\;(\mathbf{suc}\;n) \;=\; ps\;(\mathsf{ind}\mathbb{N}\;P\;pz\;ps\;n) \end{array}$$

There are other implementations of ℕInduction (a tail-recursive one, for example). But since ℕInduction is parametric in P, on which the only given operations are pz and ps, any implementation can only compute a result of type $P\ n$ using pz and ps; moreover, the index n determines that result — it has to be n applications of ps to pz. For ImmediateSublistInduction we can reason similarly, and conclude that 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, first for the simpler ℕInduction. The following is the unary parametricity statement of ℕInduction derived using Bernardy et al.'s [2012] translation, which becomes a predicate on programs of type ℕInduction:

$$\begin{array}{l} \mathbb{N}\mathsf{InductionUnaryParametricity} \;:\; \mathbb{N}\mathsf{Induction} \to \mathsf{Set}_1 \\ \mathbb{N}\mathsf{InductionUnaryParametricity}\;ind \;= \\ \quad \{P \;\;:\; \mathbb{N} \to \mathsf{Set}\} \qquad\qquad\qquad\; (Q \;\;:\; \forall\,\{n\} \to P\;n \to \mathsf{Set}) \\ \quad \{pz \;:\; P\;\mathbf{zero}\} \qquad\qquad\qquad\;\;\; (qz \;:\; Q\;pz) \\ \quad \{ps \;:\; \forall\,\{n\} \to P\;n \to P\;(\mathbf{suc}\;n)\}\;(qs \;:\; \forall\,\{n\}\,\{p \;:\; P\;n\} \to Q\;p \to Q\;(ps\;p)) \\ \quad \{n \;\;:\; \mathbb{N}\} \qquad\qquad\qquad\qquad\quad \to Q\;(ind\;P\;pz\;ps\;n) \end{array}$$

(The typesetting helps to distinguish the original arguments in ℕInduction on the left column from the entities added by the parametricity translation on the right.) Unary parametricity can be understood in terms of invariant preservation: state an invariant Q on values of type of the form $P\ n$, prove (by supplying qz and qs) that Q is satisfied by pz and preserved by ps, and then the results computed by $ind\ P\ pz\ ps$ will satisfy Q (intuitively because ind can only construct its result using pz and ps). Given a program ind : ℕInduction, we can obtain a proof of its unary parametricity for free,

$$param \;:\; \mathbb{N}\mathsf{InductionUnaryParametricity}\;ind$$

for example using Bernardy et al.'s translation again or internal parametricity [Van Muylder et al. 2024]. For any P, pz, and ps, we invoke the parametricity proof with the invariant

$$\lambda\,\{n\}\;p \to p \equiv \mathsf{ind}\mathbb{N}\;P\;pz\;ps\;n \;:\; \forall\,\{n\} \to P\;n \to \mathsf{Set}$$

saying that any $p\ :\ P\ n$ can only be the result computed by indℕ $P\ pz\ ps\ n$ (that is, n applications of ps to pz). This invariant can be easily proved to hold for pz and be preserved by ps, so we get a proof that ind has to be extensionally equal to indℕ:

$$\begin{array}{l} \quad \mathit{param}\ (\lambda\ \{n\}\ p \to p \equiv \mathsf{ind}\mathbb{N}\ P\ \mathit{pz}\ \mathit{ps}\ n)\ \mathbf{refl}\ (\lambda\ \{\mathbf{refl} \to \mathbf{refl}\}) \\ \qquad : \{n : \mathbb{N}\} \to \mathit{ind}\ P\ \mathit{pz}\ \mathit{ps}\ n \equiv \mathsf{ind}\mathbb{N}\ P\ \mathit{pz}\ \mathit{ps}\ n \end{array}$$

The proof can be readily adapted for ImmediateSublistInduction. The unary parametricity statement with respect to P (whereas A is treated merely as a fixed parameter) is

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

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:

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

Then a proof of the extensional equality between td and bu can be obtained similarly from a parametricity proof buParam : UnaryParametricity bu.

$$\mathsf{buParam}\ (\lambda\ \{ys\}\ p \to \mathsf{td}\ P\ f\ ys \equiv p)\ \mathsf{tdComp} : \{xs : \mathsf{List}\ A\} \to \mathsf{td}\ P\ f\ xs \equiv \mathsf{bu}\ P\ f\ xs \quad (4)$$

The argument tdComp proving that f preserves the invariant is worth taking a closer look. Its type is

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

which says that computing td $P\ f\ ys$ is the same as applying f to ps where every p in ps is already a result computed by td $P\ f$ — this has the same computational content as equation (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.) Incidentally, this explains why it was easy to discharge similar proof obligations in the proof for ℕInduction: indℕ satisfies the computation rules of ℕInduction definitionally.

Therefore, behind equality (4) is a theorem with a bit more structure and generality. If we formulate the computation rule for any implementation *ind* of ImmediateSublistInduction,

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

then we can generalise equality (4) 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:

$$\begin{array}{l} \mathsf{uniqueness} : \\ \qquad (\mathit{ind}\ \mathit{ind}' : \mathsf{ImmediateSublistInduction}) \\ \quad \to \mathsf{ComputationRule}\ \mathit{ind} \to \mathsf{UnaryParametricity}\ \mathit{ind}' \\ \quad \to \{A : \mathsf{Set}\}\ (P : \mathsf{List}\ A \to \mathsf{Set})\ (f : \forall\ \{ys\} \to \mathsf{Drop}\ 1\ P\ ys \to P\ ys)\ (xs : \mathsf{List}\ A) \end{array}$$

$$\rightarrow \mathit{ind}\ P\ f\ \mathit{xs} \equiv \mathit{ind}'\ P\ f\ \mathit{xs}$$
$$\text{uniqueness}\ \mathit{ind}\ \mathit{ind}'\ \mathit{comp}\ \mathit{param}'\ P\ f\ \mathit{xs}\ =\ \mathit{param}'\ (\lambda\ \{\mathit{ys}\}\ p \rightarrow \mathit{ind}\ P\ f\ \mathit{ys} \equiv p)\ \mathit{comp}$$

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

$$\begin{aligned}
&\ (\mathit{ind}\ \mathit{ind}'\ :\ \text{ImmediateSublistInduction})\\
\rightarrow&\ \text{ComputationRule}\ \mathit{ind} \rightarrow \text{ComputationRule}\ \mathit{ind}'\\
\rightarrow&\ \{A\ :\ \text{Set}\}\ (P\ :\ \text{List}\ A \rightarrow \text{Set})\ (f\ :\ \forall\ \{\mathit{ys}\} \rightarrow \text{Drop}\ 1\ P\ \mathit{ys} \rightarrow P\ \mathit{ys})\ (\mathit{xs}\ :\ \text{List}\ A)\\
\rightarrow&\ \mathit{ind}\ P\ f\ \mathit{xs} \equiv \mathit{ind}'\ P\ f\ \mathit{xs}
\end{aligned}$$

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 $h\ :\ \text{List}\ A\ \rightarrow\ B$ given $f\ :\ \text{List}\ B\ \rightarrow\ B$. The main property Mu needed was his Lemma 1, which can be roughly translated into our setting as

$$(\text{map}\ f \circ \text{upgrade})^{k}\ (\text{base}'\ \mathit{xs})\ =\ \text{map}\ h\ (\text{drop}^{\text{BT}}\ (\textbf{suc}\ (\text{length}\ \mathit{xs}) - k)\ \mathit{xs}) \tag{5}$$

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 map $f \circ$ 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 $h\ \mathit{ys}$ for all the sublists *ys* of *xs* having k elements — that is, those sublists obtained by dropping **suc** (length *xs*) − k elements from *xs*; this tree has the same shape as the one built by $\text{drop}^{\text{BT}}\ :\ \mathbb{N} \rightarrow \text{List}\ A \rightarrow \text{BT}\ (\text{List}\ A)$, which also determines the position of each $h\ \mathit{ys}$ 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 (5), Mu gave a specification of upgrade, from which the derivation of upgrade's definition was the main challenge for Mu:

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

Shape-wise, this equation says that given a tree having the shape computed by $\text{drop}^{\text{BT}}\ (\mathbf{suc}\ k)\ xs$, upgrade produces a tree having the shape computed by $\text{drop}^{\text{BT}}\ k\ xs$. 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 $\text{drop}^{\text{BT}}\ (\mathbf{suc}\ k)\ xs$. This pearl demonstrates that it is possible to go beyond shapes and encode the full specification in the type of 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 retabulate can be developed in a type-driven manner, which is more economical than Mu's equational derivation.

Equation (5) also says that each iteration of the bottom-up algorithm produces the same results as those computed by h, and Mu [2024] proved equation (5) 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 (5) generalised to all invariants. Then the uniqueness proof only needs to plug in the key part of the proof (namely the preservation of our chosen invariant) and does not need to go through the definition of bu. 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 $Q\ \{ys\}\ p\ =\ \text{td}\ P\ f\ ys \equiv p$, whereas the non-indexed result type B in type (2) 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 (3) 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. The anonymous reviewers also provided valuable feedback for us to make the presentation more accessible. 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

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *International Conference on Foundations of Software Science and Computational Structures (FoSSaCS) (Lecture Notes in Computer Science, Vol. 6014)*. Springer, 297–311. https://doi.org/10.1007/978-3-642-12032-9_21

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 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

Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *International Conference on Functional Programming (ICFP)*. ACM, 143–155. https://doi.org/10.1145/2034773.2034796

Andrzej Filinski. 1994. Representing Monads. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 446–457. https://doi.org/10.1145/174675.178047

Ralf Hinze. 2012. Kan Extensions for Program Optimisation — Or: Art and Dan Explain an Old Trick. In *International Conference on Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, 7342)*. Springer, 324–362. https://doi.org/10.1007/978-3-642-31113-0_16

Hsiang-Shang Ko, Shin-Cheng Mu, and Jeremy Gibbons. 2025. Binomial Tabulation: A Short Story. arXiv:2503.04001. https://doi.org/10.48550/arXiv.2503.04001

Conor McBride. 2002. Elimination with a Motive. In *International Workshop on Types for Proofs and Programs (TYPES) (Lecture Notes in Computer Science, Vol. 2277)*. Springer, 197–216. https://doi.org/10.1007/3-540-45842-5_13

Shin-Cheng Mu. 2024. Bottom-Up Computation Using Trees of Sublists. *Journal of Functional Programming* 34 (2024), e14:1–16. https://doi.org/10.1017/S0956796824000145

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

Zhixuan Yang and Nicolas Wu. 2022. Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes. In *International Conference on Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 13544)*. Springer, 222–267. https://doi.org/10.1007/978-3-031-16912-0_9