

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

```
subs : List A → List (List A)
```

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

$$h \ xs = f \ (\text{map } h \ (\text{subs } xs)) \quad (1)$$

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 by²

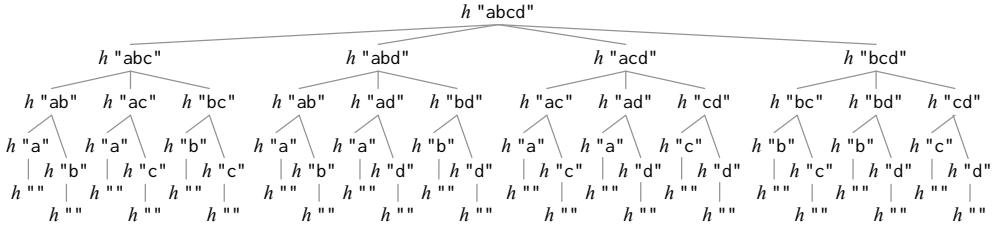
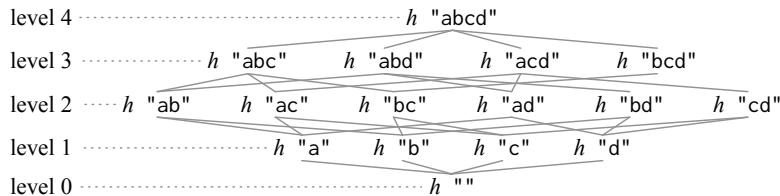
```
data BT (A : Set) : Set where
  tip : A → BT A
  bin : BT A → BT A → BT A
```

equipped with (overloaded) functions

```
map      : (A → B) → BT A → BT B
zipWith : (A → B → C) → BT A → BT B → BT C
```

¹This 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.

²We 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 $h \text{ "abcd"}$ top-down.Fig. 2. Computing $h \text{ "abcd"}$ 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 $\text{upgrade} : \text{BT } A \rightarrow \text{BT} (\text{List } A)$, a natural transformation copying and rearranging elements in t , such that map f ($\text{upgrade } t$) represents level $n + 1$. Bird suggested the following definition of upgrade:³

```

74   upgrade : BT A → BT (List A)
75   upgrade (bin (tip x) (tip y)) = tip (x :: y :: [])
76   upgrade (bin t      (tip y)) = bin (upgrade t) (map (_:: [ y ]) t)
77   upgrade (bin (tip x) u      ) = let tip xs = upgrade u in tip (x :: xs)
78   upgrade (bin t      u      ) = bin (upgrade t) (zipWith _::_ t (upgrade u))
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98

```

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?

³This 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.

99 2 THE INDUCTION PRINCIPLE AND ITS REPRESENTATIONS

100 Since we are computing a recursive function $h : \text{List } A \rightarrow B$ given $f : \text{List } B \rightarrow B$, we are dealing
 101 with a *recursion scheme* [Yang and Wu 2022] of type

$$102 \quad (103 \quad (\text{List } B \rightarrow B) \rightarrow \text{List } A \rightarrow B \quad (2)$$

104 In a dependently typed setting, recursion schemes become *elimination* or *induction principles*,
 105 and we will refine type (2) to an induction principle. The first step is refining $\text{List } A \rightarrow B$ to
 106 $(xs : \text{List } A) \rightarrow P xs$ where $P : \text{List } A \rightarrow \text{Set}$ is the induction *motive* [McBride 2002]. And
 107 then we should refine the premise $\text{List } B \rightarrow B$ to an induction *method*, whose type states that the
 108 motive should be established and preserved in a way that follows the recursive structure of the
 109 computation — or more specifically, whenever P holds for all the immediate sublists of a list ys ,
 110 P should hold for ys as well.

111 To write down the induction principle formally (in particular the type of the induction method),
 112 first we need to define immediate sublists — in fact we will give a more general definition of sublists,
 113 following Mu's [2024] insight that we will need to refer to all of the sublists during the course of
 114 the computation. Recall that an immediate sublist of xs is a list obtained by dropping one element
 115 from xs ; more generally, a sublist can be obtained by dropping some number of elements. Element
 116 dropping can be written as an inductively defined relation:

$$\begin{aligned} 117 \quad \text{data } \text{Drop}^R : \mathbb{N} &\rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \text{Set} \text{ where} \\ 118 \quad \text{return} &: \quad \text{Drop}^R \text{ zero } xs \quad xs \\ 119 \quad \text{drop} &: \text{Drop}^R \quad n \quad xs \quad ys \rightarrow \text{Drop}^R (\text{suc } n) (x :: xs) \quad ys \\ 120 \quad \text{keep} &: \text{Drop}^R (\text{suc } n) \quad xs \quad ys \rightarrow \text{Drop}^R (\text{suc } n) (x :: xs) \quad (x :: ys) \end{aligned}$$

122 Dropping **zero** elements from any list xs is just returning xs itself; when dropping **suc** n elements,
 123 the relation is defined only for nonempty lists $x :: xs$, and we may choose to drop x and continue to
 124 drop n elements from xs , or to keep x and continue to drop **suc** n elements from xs . With the help
 125 of Drop^R we can quantify over sublists; in particular, we can state that a motive P holds for all the
 126 immediate sublists zs of a list ys :

$$128 \quad \forall \{zs\} \rightarrow \text{Drop}^R 1 \quad ys \quad zs \rightarrow P \quad zs \quad (3)$$

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

$$\begin{aligned} 131 \quad \{A : \text{Set}\} \quad (P : \text{List } A \rightarrow \text{Set}) \\ 132 \quad (f : \forall \{ys\} \rightarrow (\forall \{zs\} \rightarrow \text{Drop}^R 1 \quad ys \quad zs \rightarrow P \quad zs) \rightarrow P \quad ys) \\ 133 \quad (xs : \text{List } A) \rightarrow P \quad xs \end{aligned}$$

136 This is one possible refinement of type (2). But notice that the induction hypotheses are represented
 137 as a function of type (3), whereas type (2) uses $\text{List } B$, a first-order data structure. Below we derive
 138 an indexed data type $\text{Drop } n \ P \ xs$ that represents universal quantification over all the sublists
 139 obtained by dropping n elements from xs ; in particular, $\text{Drop } 1 \ P \ ys$ will be equivalent to type (3).

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

$$\begin{aligned} 143 \quad \text{mzero} &: \text{Nondet } A \\ 144 \quad \text{mplus} &: \text{Nondet } A \rightarrow \text{Nondet } A \rightarrow \text{Nondet } A \end{aligned}$$

146 Then element dropping can be defined monadically by

```

148 drop :  $\mathbb{N} \rightarrow \text{List } A \rightarrow \text{Nondet}(\text{List } A)$ 
149 drop zero xs = return xs
150 drop (suc n) [] = mzero
151 drop (suc n) (x :: xs) = mplus (drop n xs) (drop (suc n) xs  $\gg= \lambda ys \rightarrow \text{return}(x :: ys)$ )
152

```

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

```

155 Nondet : Set  $\rightarrow \text{Set}_\omega$ 
156 Nondet A =  $\forall \{\ell\} \{M : \text{Set } \ell\} \rightarrow \{\{\text{Monoid } M\}\} \rightarrow (A \rightarrow M) \rightarrow M$ 
157

```

with the standard definitions of return and bind:

```

159 return : A  $\rightarrow \text{Nondet } A$ 
160 return x =  $\lambda k \rightarrow k x$ 
161 _ $\gg=$ _ : Nondet A  $\rightarrow (A \rightarrow \text{Nondet } B) \rightarrow \text{Nondet } B$ 
162 mx  $\gg= f = \lambda k \rightarrow mx(\lambda x \rightarrow f x k)$ 
163

```

In the definition of Nondet, there is an additional $\{\{\text{Monoid } M\}\}$ 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:

```

167 record Monoid (M : Set  $\ell$ ) : Set  $\ell$  where
168   constructor monoid
169   field
170     _ $\oplus$ _ : M  $\rightarrow M \rightarrow M$ 
171      $\emptyset$  : M
172

```

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

```

176 mzero : Nondet A
177 mzero =  $\lambda k \rightarrow \emptyset$ 

```

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

```

181 mplus : Nondet A  $\rightarrow \text{Nondet } A \rightarrow \text{Nondet } A$ 
182 mplus mx my =  $\lambda k \rightarrow mx k \oplus my k$ 

```

(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

```

187 drop :  $\mathbb{N} \rightarrow \text{List } A \rightarrow \{\{\text{Monoid } M\}\} \rightarrow (\text{List } A \rightarrow M) \rightarrow M$ 
188 drop zero xs k = k xs
189 drop (suc n) [] k =  $\emptyset$ 
190 drop (suc n) (x :: xs) k = drop n xs k  $\oplus$  drop (suc n) xs (k  $\circ$  (x :: _))

```

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:

⁴Technically, 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.

```

197 dropL :  $\mathbb{N} \rightarrow \text{List } A \rightarrow \text{List } (\text{List } A)$ 
198 dropL n xs = drop n xs {{ monoid  $\_+\_$  [] }} ( $\_::$  [])
199

```

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

201 More importantly, we want to instantiate drop to compute types. For example, Drop^R can
202 alternatively be defined in continuation-passing style by

```

203 DropR n xs ys  $\cong$  drop n xs {{ monoid  $\_+\_ \perp$  }} ( $\_ = ys$ )
204

```

205 where $\text{drop } n \text{ xs } \{\{\text{monoid } _+_ \perp\}\} : (\text{List } A \rightarrow \text{Set}) \rightarrow \text{Set}$ amounts to existential quantification
206 over sublists: an input predicate $P : \text{List } A \rightarrow \text{Set}$ is only required to hold in one of the branches at
207 every nondeterministic choice (since we instantiate the monoid operation $_+_$ to disjunction $_+_$), so eventually
208 P is only required to hold for one sublist. To obtain universal quantification, we
209 supply the conjunction $\text{monoid } _x_\top$, requiring P to hold for all the branches and thus all the
210 sublists:

```

211 Drop n P xs  $\cong$  drop n xs {{ monoid  $\_x\_\top$  }} P
212

```

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

```

215 data Drop :  $\mathbb{N} \rightarrow (\text{List } A \rightarrow \text{Set}) \rightarrow \text{List } A \rightarrow \text{Set}$  where
216   tip : P xs
217   nil :
218   bin : Drop n P xs  $\rightarrow$  Drop (suc n) ( $P \circ (x :: \_)$ ) xs  $\rightarrow$  Drop (suc n) P ( $x :: xs$ )
219

```

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

```

221 ImmediateSublistInduction : Set1
222 ImmediateSublistInduction = {A : Set} (P : List A  $\rightarrow$  Set)
223           (f :  $\forall \{ys\} \rightarrow \text{Drop } 1 \text{ P } ys \rightarrow P \text{ ys}$ )
224           (xs : List A)  $\rightarrow$  P xs
225

```

226 Comparing `ImmediateSublistInduction` with type (2), a potentially drastic change is that the *list*
227 of induction hypotheses is replaced with a *tree* of type $\text{Drop } 1 \text{ P } ys$ here. However, such a tree
228 is actually list-shaped (constructed using `nil` and `bin` \circ `tip`), so `ImmediateSublistInduction` is a
229 faithful refinement of type (2). Moreover, we will see that `Drop` is a more informative version of `BT`
230 (Section 1) with an additional `nil` constructor — we will reimplement the upgrade function used in
231 the bottom-up algorithm to operate on `Drop` trees, and the same computation patterns will emerge.
232 This give us a better idea of why Bird [2008] needed to use `BT`: paths in `BT/Drop` trees correspond
233 to computation of sublists of a particular length, so working with `BT/Drop` trees allows us to figure
234 out which sublist each element in the trees is associated with; the associations are only implicitly
235 assumed in `BT`, whereas in `Drop` they are explicitly recorded in the element types of the form $P \text{ xs}$.
236

237 In Sections 3 and 4 we will implement the top-down and bottom-up algorithms as programs of type
238 `ImmediateSublistInduction`. These are fairly standard exercises in dependently typed programming
239 (except perhaps for upgrade), and our implementations are by no means the only solutions.⁵ The
240 reader may want to try the exercises for themselves, and is not obliged to go through the details of
241 our programs. We will prove that the two algorithms are extensionally equal in Section 5; the proof
242 will not depend on the implementation details of the two algorithms, but only on their shared
243 dependent type.

244 ⁵Even the induction principle has alternative formulations, one of which was explored by Ko et al. [2025].
245

246 **3 THE TOP-DOWN ALGORITHM**

247 Equation (1) is essentially an executable definition of the top-down algorithm. This definition would
 248 not pass Agda's termination check though, because the immediate sublists in `subs xs` would not be
 249 recognised as structurally smaller than `xs`. One way to make termination evident is to make the
 250 length of `xs` explicit and perform induction on the length. The following function `td` does this by
 251 invoking `td'`, which takes as additional arguments a natural number `l` and an equality proof stating
 252 that the length of `xs` is `l`. The function `td'` then performs induction on `l` and does the real work.
 253

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

260

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

```
264    lenSubs : (l : ℕ) (xs : List A) → length xs ≡ suc l
  265       → Drop 1 (λ ys → length ys ≡ l) xs
  266
```

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

```
269    map : (forall {ys} → P ys → Q ys) → Drop n P xs → Drop n Q xs
```

271 and again use `f` to compute the final result.

273 **4 THE BOTTOM-UP ALGORITHM**

274 Given an input list `xs`, the bottom-up algorithm `bu` first creates a tree representing ‘level –1’ below
 275 the lattice in Figure 2. This ‘basement’ level contains results for those sublists obtained by removing
 276 `suc (length xs)` elements from `xs`; there are no such sublists, so the tree contains no elements,
 277 although the tree itself still exists (representing a proof of a vacuous universal quantification, or
 278 more specifically, a proof that all the branches in the nondeterministic computation of `drop` end
 279 with failure):
 280

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

282 The algorithm then enters a loop `bu'` and constructs each level of the lattice from bottom up, that
 283 is, a tree of type `Drop n P xs` for each `n`, with `n` decreasing:
 284

```
285    bu : ImmediateSublistInduction
  286    bu P f = bu' _ ∘ base
  287    where -- unTip and rebabulate to be defined later
  288       bu' : (n : ℕ) → Drop n P xs → P xs
  289       bu' zero = unTip
  290       bu' (suc n) = bu' n ∘ map f ∘ rebabulate
```

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

```
295 unTip : Drop zero P xs → P xs
296 unTip (tip p) = p
297
```

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:

```
303 retabulate : Drop (suc n) P xs → Drop n (Drop 1 P) xs
304
```

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`.

```
311 retabulate : Drop (suc n) P xs → Drop n (Drop 1 P) xs
312 retabulate nil = underground
313 retabulate t@(bin (tip _) _) = tip t
314 retabulate (bin nil nil) = bin underground nil
315 retabulate (bin t@(bin _ _) u) = bin (retabulate t)
316                                         (zipWith (bin ∘ tip) t (retabulate u))
317
```

The auxiliary function `underground` is defined by

```
320 underground : Drop n (Drop 1 P) []
321 underground {n = zero} = tip nil
322 underground {n = suc _} = nil
323
```

(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:

```
330 zipWith : (∀ {ys} → P ys → Q ys → R ys)
331           → Drop n P xs → Drop n Q xs → Drop n R xs
332
```

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.

344 **5 EXTENSIONAL EQUALITY BETWEEN THE TWO ALGORITHMS**

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

347 Actually, is it possible to write programs of type `ImmediateSublistInduction` to compute different
 348 results in Agda? It may help to consider a simpler example: induction on natural numbers,
 349

```
350   NInduction : Set1
351   NInduction = (P : N → Set)
352     (pz : P zero) (ps : ∀ {n} → P n → P (suc n))
353     (n : N) → P n
```

354 which has a standard implementation:

```
355   indN : NInduction
356   indN P pz ps zero = pz
357   indN P pz ps (suc n) = ps (indN P pz ps n)
```

358 There are other implementations of `NInduction` (a tail-recursive one, for example). But since
 359 `NInduction` is parametric in *P*, on which the only given operations are *pz* and *ps*, any implementation
 360 can only compute a result of type *P n* using *pz* and *ps*; moreover, the index *n* determines that result
 361 — it has to be *n* applications of *ps* to *pz*. For `ImmediateSublistInduction` we can reason similarly,
 362 and conclude that `td` and `bu` have to compute the same results simply because they have the same
 363 —and special— type!

364 To prove this formally, we use parametricity, first for the simpler `NInduction`. The following is
 365 the unary parametricity statement of `NInduction` derived using Bernardy et al.’s [2012] translation,
 366 which becomes a predicate on programs of type `NInduction`:

```
367   NInductionUnaryParametricity : NInduction → Set1
368   NInductionUnaryParametricity ind =
369     {P : N → Set}           (Q : ∀ {n} → P n → Set)
370     {pz : P zero}          (qz : Q pz)
371     {ps : ∀ {n} → P n → P (suc n)} (qs : ∀ {n} {p : P n} → Q p → Q (ps p))
372     {n : N}                 → Q (ind P pz ps n)
```

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

```
380   param : NInductionUnaryParametricity ind
```

381 for example using Bernardy et al.’s translation again or internal parametricity [Van Muylder et al.
 382 2024]. For any *P*, *pz*, and *ps*, we invoke the parametricity proof with the invariant

```
383   λ {n} p → p ≡ indN P pz ps n : ∀ {n} → P n → Set
```

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

387

```

393 param (λ {n} p → p ≡ indN P pz ps n) refl (λ {refl → refl})
394   : {n : N} → ind P pz ps n ≡ indN P pz ps n
395

```

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

```

398 UnaryParametricity : ImmediateSublistInduction → Set1
399 UnaryParametricity ind =
400   {A : Set} {P : List A → Set} (Q : ∀ {ys} → P ys → Set)
401     {f : ∀ {ys} → Drop 1 P ys → P ys} (g : ∀ {ys} {ps : Drop 1 P ys}
402       → All Q ps → Q (f ps))
403     {xs : List A} → Q (ind P f xs)
404

```

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:

```

407 All : (∀ {ys} → P ys → Set) → Drop n P xs → Set
408 All Q (tip p) = Q p
409 All Q nil = ⊤
410 All Q (bin t u) = All Q t × All Q u
411

```

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

```

412 buParam (λ {ys} p → td P f ys ≡ p) tdComp : {xs : List A} → td P f xs ≡ bu P f xs (4)
413

```

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

```

414   ∀ {ys} {ps : Drop 1 P ys} → All (λ {zs} p → td P f zs ≡ p) ps → td P f ys ≡ f ps
415

```

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`,

```

416 ComputationRule : ImmediateSublistInduction → Set1
417 ComputationRule ind =
418   {A : Set} {P : List A → Set} {f : ∀ {ys} → Drop 1 P ys → P ys} {xs : List A}
419     {ps : Drop 1 P xs} → All (λ {ys} p → ind P f ys ≡ p) ps → ind P f xs ≡ f ps
420

```

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:

```

421 uniqueness :
422   (ind ind' : ImmediateSublistInduction)
423   → ComputationRule ind → UnaryParametricity ind'
424   → {A : Set} (P : List A → Set) (f : ∀ {ys} → Drop 1 P ys → P ys) (xs : List A)
425

```

442 $\rightarrow \text{ind } P f xs \equiv \text{ind}' P f xs$
 443 uniqueness $\text{ind } \text{ind}' \text{ comp param}' P f xs = \text{param}' (\lambda \{ys\} p \rightarrow \text{ind } P f ys \equiv p) \text{ comp}$
 444
 445

6 METHODOLOGICAL DISCUSSIONS

6.1 Proving uniqueness of induction principle implementations from parametricity

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

451 (ind ind' : ImmediateSublistInduction)
 452 $\rightarrow \text{ComputationRule ind} \rightarrow \text{ComputationRule ind}'$
 453 $\rightarrow \{A : \text{Set}\} (P : \text{List } A \rightarrow \text{Set}) (f : \forall \{ys\} \rightarrow \text{Drop } 1 P ys \rightarrow P ys) (xs : \text{List } A)$
 454 $\rightarrow \text{ind } P f xs \equiv \text{ind}' P f xs$

455
 456 The uniqueness theorem in Section 5 demonstrates (in terms of `ImmediateSublistInduction`) that
 457 we can alternatively assume that one implementation, say ind' , satisfies unary parametricity instead,
 458 and we will still have a proof. This is useful when ind can be easily proved to satisfy the set of
 459 computation rules whereas ind' cannot. In our case, even though our `td` in Section 3 does not
 460 satisfy the computation rule definitionally (because it performs a different form of induction on
 461 the length of the input list, to make termination evident to Agda), a proof of `ComputationRule td`
 462 still takes only a small amount of work. It would be more difficult to prove that `bu` satisfies the
 463 computation rule, whereas a parametricity proof for `bu` is always mechanical—if not automatic—
 464 to derive, so switching to the latter greatly reduces the proof burden. In general, this trick may be
 465 useful for porting recursion schemes or inventing efficient implementations of induction principles
 466 in a dependently typed setting.

6.2 Establishing invariants using indexed data types and parametricity

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

$$475 \quad (\text{map } f \circ \text{upgrade})^k (\text{base}' xs) = \text{map } h (\text{drop}^{\text{BT}} (\text{suc} (\text{length } xs) - k) xs) \quad (5)$$

476
 477 This is an old-school way of saying that the bottom-up algorithm maintains an invariant. The left-
 478 hand side is the value computed by the bottom-up algorithm after k iterations: xs is the initial input;
 479 base' plays a similar role as base in Section 4 and prepares an initial tree, on which $\text{map } f \circ \text{upgrade}$,
 480 the loop body of the bottom-up algorithm, is performed k times. The invariant is that the value
 481 must equal the right-hand side: a tree containing values $h ys$ for all the sublists ys of xs having
 482 k elements—that is, those sublists obtained by dropping $\text{suc} (\text{length } xs) - k$ elements from xs ;
 483 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
 484 determines the position of each $h ys$ in the tree. By contrast, this pearl uses (i) the indexed data
 485 type `Drop` to enforce tree shapes and sublist positions and (ii) parametricity to establish that the
 486 trees contain values of `td`.

487 Using indexed data types to enforce shape constraints is a well known technique, which in
 488 particular was briefly employed by Mu [2024, Section 4.3]. But program specifications are often not
 489 just about shapes. For example, to prove equation (5), Mu gave a specification of `upgrade`, from

491 which the derivation of upgrade's definition was the main challenge for Mu:

$$493 \quad \text{upgrade} (\text{drop}^{\text{BT}} (\text{suc } k) \text{ xs}) = \text{map subs} (\text{drop}^{\text{BT}} k \text{ xs})$$

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

504 Equation (5) also says that each iteration of the bottom-up algorithm produces the same results
 505 as those computed by h , and Mu [2024] proved equation (5) by induction on k . What is the
 506 relationship between Mu's inductive proof and ours based on `UnaryParametricity bu` (Section 5)?
 507 Mu's induction on k coincides with the looping structure of the bottom-up algorithm. On the
 508 other hand, while `UnaryParametricity` could in principle be proved mechanically once-and-for-all
 509 for all functions having the right type, if one had to prove `UnaryParametricity bu` manually, the
 510 proof would also follow the structure of `bu`. Therefore the proof of `bu`'s unary parametricity would
 511 essentially be the proof of equation (5) generalised to all invariants. Then the uniqueness proof
 512 only needs to plug in the key part of the proof (namely the preservation of our chosen invariant)
 513 and does not need to go through the definition of `bu`. Finally, note that this opportunity to invoke
 514 parametricity emerges because we switch to dependent types and reformulate the recursion scheme
 515 as an induction principle: knowing that a result p has the indexed type $P \{ ys \}$ allows us to state the
 516 invariant $Q \{ ys \} p = \text{td } P f \{ ys \} \equiv p$, whereas the non-indexed result type B in type (2) does not
 517 provide enough information for stating that.
 518

519 ACKNOWLEDGEMENTS

520 Zhixuan Yang engaged in several discussions about induction principles, computation rules, and
 521 parametricity, leading to the current presentation of the parametricity-based proof. He also pointed
 522 out how `Nondet` is an instance of the codensity representation except that a dinaturality condition
 523 is omitted [Hinze 2012]. At the IFIP WG 2.1 meeting in April 2024, James McKinna suggested
 524 defining `retabulate` on the higher-order representation (3) instead. This definition of `retabulate` is
 525 extremely simple, but does not copy and reuse results on sublists, and therefore does not help to
 526 avoid re-computation. However, this perspective does make the relationship between binomial
 527 trees and proofs of universal quantification clear, and leads to the inclusion of the `nil` constructor in
 528 `Drop` (which helps to simplify our definition of `retabulate`). At the same meeting, Wouter Swierstra
 529 asked whether lists could be used instead of vectors in a previous definition of binomial trees [Ko
 530 et al. 2025]. There the definition of immediate sublists depends on the length of the input list, so
 531 it is more convenient to use vectors. However, this question leads us to consider a definition of
 532 immediate sublists that does not depend on list length, and ultimately to the simpler definition
 533 of `Drop` (which uses lists instead of vectors). Yen-Hao Liu previewed and provided feedback on a
 534 draft. The anonymous reviewers also provided valuable feedback for us to make the presentation
 535 more accessible. We would like to thank all of them.

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

540 REFERENCES

- 541 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *International
542 Conference on Foundations of Software Science and Computational Structures (FoSSaCS) (Lecture Notes in Computer Science,
543 Vol. 6014)*. Springer, 297–311. https://doi.org/10.1007/978-3-642-12032-9_21
- 544 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free: Parametricity for Dependent Types.
545 *Journal of Functional Programming* 22, 2 (2012), 107–152. <https://doi.org/10.1017/S0956796812000056>
- 546 Richard S. Bird. 2008. Zippy Tabulations of Recursive Functions. In *International Conference on Mathematics of Program
547 Construction (MPC) (Lecture Notes in Computer Science, Vol. 5133)*. Springer, 92–109. [https://doi.org/10.1007/978-3-540-70594-9_7](https://doi.org/10.1007/978-3-540-
70594-9_7)
- 548 Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In
549 *International Conference on Functional Programming (ICFP)*. ACM, 143–155. <https://doi.org/10.1145/2034773.2034796>
- 550 Andrzej Filinski. 1994. Representing Monads. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 446–457.
<https://doi.org/10.1145/174675.178047>
- 551 Ralf Hinze. 2012. Kan Extensions for Program Optimisation – Or: Art and Dan Explain an Old Trick. In *International
552 Conference on Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, 7342)*. Springer, 324–362.
553 https://doi.org/10.1007/978-3-642-31113-0_16
- 554 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>
- 555 Conor McBride. 2002. Elimination with a Motive. In *International Workshop on Types for Proofs and Programs (TYPES)
556 (Lecture Notes in Computer Science, Vol. 2277)*. Springer, 197–216. https://doi.org/10.1007/3-540-45842-5_13
- 557 Shin-Cheng Mu. 2024. Bottom-Up Computation Using Trees of Sublists. *Journal of Functional Programming* 34 (2024),
558 e14:1–16. <https://doi.org/10.1017/S0956796824000145>
- 559 Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. 2024. Internal and Observational Parametricity for Cubical
560 Agda. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 8:1–32. <https://doi.org/10.1145/3632850>
- 561 Zhixuan Yang and Nicolas Wu. 2022. Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes. In
562 *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
- 563
- 564
- 565
- 566
- 567
- 568
- 569
- 570
- 571
- 572
- 573
- 574
- 575
- 576
- 577
- 578
- 579
- 580
- 581
- 582
- 583
- 584
- 585
- 586
- 587
- 588