

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

$$\text{subs} : \text{List } A \rightarrow \text{List } (\text{List } A)$$

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 $\text{map } h : \text{List } (\text{List } A) \rightarrow \text{List } B$ 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

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

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 by²

data $\text{BT } (A : \text{Set}) : \text{Set}$ **where**

tip : $A \rightarrow \text{BT } A$

bin : $\text{BT } A \rightarrow \text{BT } A \rightarrow \text{BT } A$

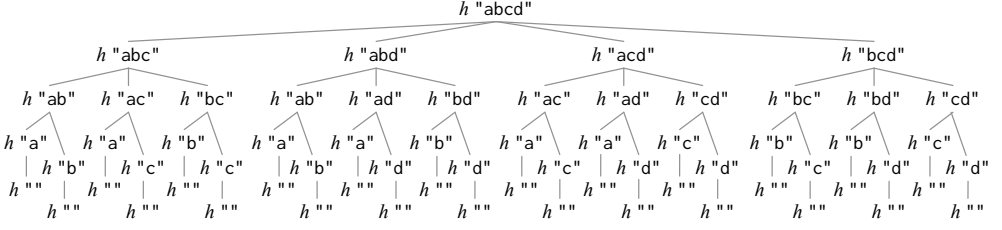
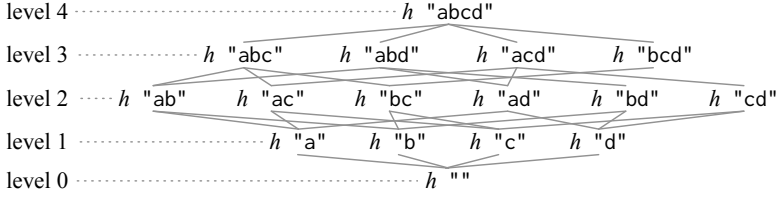
equipped with (overloaded) functions

map : $(A \rightarrow B) \rightarrow \text{BT } A \rightarrow \text{BT } B$

zipWith : $(A \rightarrow B \rightarrow C) \rightarrow \text{BT } A \rightarrow \text{BT } B \rightarrow \text{BT } C$

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 "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 $\text{upgrade} : \text{BT } A \rightarrow \text{BT (List } A)$, a natural transformation copying and rearranging elements in t , such that $\text{map } f (\text{upgrade } t)$ 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 (λ_:: [ y ]) t)
upgrade (bin (tip x) u      ) = let tip xs = upgrade u in tip (x :: xs)
upgrade (bin t      u      ) = bin (upgrade t) (zipWith λ_::_ 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?

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.

2 THE INDUCTION PRINCIPLE AND ITS REPRESENTATIONS

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

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

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 $\text{List } A \rightarrow B$ to $(xs : \text{List } A) \rightarrow P \text{ } xs$ where $P : \text{List } A \rightarrow \text{Set}$ is the induction *motive* [McBride 2002]. And then we should refine the premise $\text{List } B \rightarrow B$ 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:

```

data DropR : ℕ → List A → List A → Set where
  return :                               DropR zero  xs      xs
  drop   : DropR      n xs ys → DropR (suc n) (x :: xs) ys
  keep   : DropR (suc n) xs ys → DropR (suc n) (x :: xs) (x :: ys)

```

Dropping **zero** elements from any list xs is just returning xs itself; when dropping **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 **suc** n elements from xs . With the help of Drop^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 :

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

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:

```

{A : Set} (P : List A → Set)
(f : ∀ {ys} → (∀ {zs} → DropR 1 ys zs → P zs) → P ys)
(xs : List A) → P xs

```

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 $\text{List } B$, a first-order data structure. Below we derive an indexed data type $\text{Drop } n \text{ } P \text{ } xs$ that represents universal quantification over all the sublists obtained by dropping n elements from xs ; in particular, $\text{Drop } 1 \text{ } P \text{ } ys$ 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 :

```

mzero : Nondet A
mplus : Nondet A → Nondet A → Nondet A

```

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  $\rightarrow$  return (x :: ys))

```

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

```

with the standard definitions of return and bind:

```

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

```

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:

```

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

```

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  $\rightarrow$  Nondet A  $\rightarrow$  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.

$\text{drop}^L : \mathbb{N} \rightarrow \text{List } A \rightarrow \text{List } (\text{List } A)$
 $\text{drop}^L n \text{ xs} = \text{drop } n \text{ xs} \{ \{ \text{monoid } _ \oplus _ [] \} \} (_ :: [])$

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

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

$\text{Drop}^R n \text{ xs } \text{ys} \cong \text{drop } n \text{ xs} \{ \{ \text{monoid } _ \sqcup _ \perp \} \} (_ \equiv \text{ys})$

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

$\text{Drop } n P \text{ xs} \cong \text{drop } n \text{ xs} \{ \{ \text{monoid } _ \times _ \top \} \} P$

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

data $\text{Drop} : \mathbb{N} \rightarrow (\text{List } A \rightarrow \text{Set}) \rightarrow \text{List } A \rightarrow \text{Set}$ **where**
 $\text{tip} : P \text{ xs} \rightarrow \text{Drop } \text{zero} P \text{ xs}$
 $\text{nil} : \text{Drop } (\text{suc } n) P []$
 $\text{bin} : \text{Drop } n P \text{ xs} \rightarrow \text{Drop } (\text{suc } n) (P \circ (x :: _)) \text{ xs} \rightarrow \text{Drop } (\text{suc } n) P (x :: \text{xs})$

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

$\text{ImmediateSublistInduction} : \text{Set}_1$
 $\text{ImmediateSublistInduction} = \{ A : \text{Set} \} (P : \text{List } A \rightarrow \text{Set})$
 $(f : \forall \{ \text{ys} \} \rightarrow \text{Drop } 1 P \text{ ys} \rightarrow P \text{ ys})$
 $(\text{xs} : \text{List } A) \rightarrow P \text{ xs}$

Comparing $\text{ImmediateSublistInduction}$ with type (2), a potentially drastic change is that the *list* of induction hypotheses is replaced with a *tree* of type $\text{Drop } 1 P \text{ ys}$ here. However, such a tree is actually list-shaped (constructed using nil and $\text{bin} \circ \text{tip}$), so $\text{ImmediateSublistInduction}$ is a faithful refinement of type (2). Moreover, we will see that Drop is a more informative version 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. This give 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; the associations are only implicitly assumed in BT, whereas in Drop they are explicitly recorded in the element types of the form $P \text{ xs}$.

In Sections 3 and 4 we will implement the top-down and bottom-up algorithms as programs of type $\text{ImmediateSublistInduction}$. 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 themselves, 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.

⁵Even 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.

```

246  td : ImmediateSublistInduction
247  td {A} P f xs = td' (length xs) xs refl
248  where -- lenSubs to be defined later
249      td' : (l : ℕ) (xs : List A) → length xs ≡ l → P xs
250      td' zero [] eq = f nil
251      td' (suc l) xs eq = f (map (λ {ys} → td' l ys) (lenSubs l xs eq))
252
253
254

```

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 :

```

255  lenSubs : (l : ℕ) (xs : List A) → length xs ≡ suc l
256          → Drop 1 (λ ys → length ys ≡ l) xs
257
258
259

```

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`,

```

260  map : (∀ {ys} → P ys → Q ys) → Drop n P xs → Drop n Q xs
261
262
263

```

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):

```

264  base : (xs : List A) → Drop (suc (length xs)) P xs
265
266
267

```

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:

```

268  bu : ImmediateSublistInduction
269  bu P f = bu' _ ∘ base
270  where -- unTip and retabulate to be defined later
271      bu' : (n : ℕ) → Drop n P xs → P xs
272      bu' zero = unTip
273      bu' (suc n) = bu' n ∘ map f ∘ retabulate
274
275
276

```

When the loop counter reaches `zero`, the tree contains exactly the result for `xs`, which we can 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 $\text{Drop } n P \text{ xs}$ that is one level higher than
 298 the current tree of type $\text{Drop } (\text{suc } n) P \text{ xs}$. The type of the new tree says that it should contain
 299 results of type $P \text{ ys}$ for all the sublists ys at the higher level. The **retabulate** function, which plays
 300 the same role as **upgrade** (Section 1), does half of the work by copying and rearranging the elements
 301 of the current tree to construct an intermediate tree representing the higher level:
 302

```

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 $P \text{ ys}$ using f — that is,
 305 each element of the intermediate tree is a tree of type $\text{Drop } 1 P \text{ ys}$. Then $\text{map } f$ does the rest of the
 306 work and produces the desired new tree of type $\text{Drop } n P \text{ xs}$, and we enter the next iteration.

307 To implement **retabulate**, follow the types, and most of the program writes itself. We will not go
 308 through the construction of the program in detail because we only aim to show the correctness of
 309 **bu**, which depends only on the type of **retabulate**.
 310

```

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
 318

```

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

```

322 (It analyses the implicit argument n , which therefore needs to be present at runtime, so **retabulate**
 323 actually requires more information than the input tree to execute, unlike **upgrade**.) The last clause
 324 of **retabulate** is the most difficult one to conceive, but can be copied exactly from the last clause
 325 of **upgrade** except that the list **cons** is replaced by the **cons** function $\text{bin} \circ \text{tip}$ for $\text{Drop } 1$ trees
 326 (which, as mentioned near the end of Section 2, are list-shaped), and the type of **zipWith** needs to
 327 be updated:
 328

```

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

```

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

334 The first and third clauses of **retabulate** involve **nil**, and have no counterparts in **upgrade**. **Drop**
 335 trees containing **nil** correspond to empty levels below the lattice in Figure 2 (which result from
 336 dropping too many elements from the input list). Mu [2024] avoided dealing with such empty
 337 levels by imposing conditions throughout his development — for example, see Mu’s Section 4.3
 338 and Appendix B for a version of the program (which is named up there) with conditions. We avoid
 339 those somewhat tedious conditions by including **nil** in **Drop** to represent the empty levels, and in
 340 exchange need to deal with these levels, which are easier to deal with than the conditions though.
 341

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,

```

NInduction : Set1
NInduction = (P : ℕ → Set)
              (pz : P zero) (ps : ∀ {n} → P n → P (suc n))
              (n : ℕ) → P n

```

which has a standard implementation:

```

indN : NInduction
indN P pz ps zero = pz
indN P pz ps (suc n) = ps (indN P pz ps n)

```

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`:

```

NInductionUnaryParametricity : NInduction → Set1
NInductionUnaryParametricity ind =
  { P : ℕ → Set }          ( Q : ∀ {n} → P n → Set )
  { pz : P zero }           ( qz : Q pz )
  { ps : ∀ {n} → P n → P (suc n) } ( qs : ∀ {n} {p : P n} → Q p → Q (ps p) )
  { n : ℕ }                 → Q (ind P pz ps n)

```

(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 : NInductionUnaryParametricity 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

```

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

$param (\lambda \{n\} p \rightarrow p \equiv ind \mathbb{N} P pz ps n) \text{ refl } (\lambda \{ \text{refl} \rightarrow \text{refl} \})$
 $: \{n : \mathbb{N}\} \rightarrow ind P pz ps n \equiv ind \mathbb{N} P pz ps n$

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

UnaryParametricity : ImmediateSublistInduction \rightarrow Set₁
 UnaryParametricity ind =
 $\{A : \text{Set}\} \{P : \text{List } A \rightarrow \text{Set}\} \quad (Q : \forall \{ys\} \rightarrow P \text{ } ys \rightarrow \text{Set})$
 $\{f : \forall \{ys\} \rightarrow \text{Drop } 1 P \text{ } ys \rightarrow P \text{ } ys\} \quad (g : \forall \{ys\} \{ps : \text{Drop } 1 P \text{ } ys\}$
 $\rightarrow \text{All } Q \text{ } ps \rightarrow Q (f \text{ } ps))$
 $\{xs : \text{List } A\} \quad \rightarrow Q (ind P f xs)$

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:

All : $(\forall \{ys\} \rightarrow P \text{ } ys \rightarrow \text{Set}) \rightarrow \text{Drop } n P \text{ } xs \rightarrow \text{Set}$
 All $Q (\text{tip } p) = Q p$
 All $Q \text{ nil} = \top$
 All $Q (\text{bin } t \text{ } u) = \text{All } Q t \times \text{All } Q u$

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

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

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

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

which says that computing $\text{td } P f \text{ } ys$ is the same as applying f to ps where every p in ps is already a result computed by $\text{td } P f$ — 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 \mathbb{N} Induction: $ind \mathbb{N}$ satisfies the computation rules of \mathbb{N} Induction 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,

ComputationRule : ImmediateSublistInduction \rightarrow Set₁
 ComputationRule ind =
 $\{A : \text{Set}\} \{P : \text{List } A \rightarrow \text{Set}\} \{f : \forall \{ys\} \rightarrow \text{Drop } 1 P \text{ } ys \rightarrow P \text{ } ys\} \{xs : \text{List } A\}$
 $\{ps : \text{Drop } 1 P \text{ } xs\} \rightarrow \text{All } (\lambda \{ys\} p \rightarrow ind P f \text{ } ys \equiv p) \text{ } ps \rightarrow ind P f \text{ } xs \equiv f \text{ } ps$

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:

uniqueness :
 $(ind \text{ } ind' : \text{ImmediateSublistInduction})$
 $\rightarrow \text{ComputationRule } ind \rightarrow \text{UnaryParametricity } ind'$
 $\rightarrow \{A : \text{Set}\} (P : \text{List } A \rightarrow \text{Set}) (f : \forall \{ys\} \rightarrow \text{Drop } 1 P \text{ } ys \rightarrow P \text{ } ys) (xs : \text{List } A)$

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

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

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

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

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

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 $\text{map } f \circ \text{upgrade}$, 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 $h \text{ } ys$ for all the sublists ys of xs having k elements – that is, those sublists obtained by dropping $\text{suc } (\text{length } xs) - k$ 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 $h \text{ } ys$ 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:

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

Shape-wise, this equation says that given a tree having the shape computed by $\text{drop}^{\text{BT}} (\text{suc } k) \text{ xs}$, `upgrade` produces a tree having the shape computed by $\text{drop}^{\text{BT}} k \text{ xs}$. 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 $\text{drop}^{\text{BT}} (\text{suc } k) \text{ xs}$. 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 $P \text{ ys}$ allows us to state the invariant $Q \{ \text{ys} \} p = \text{td } P f \text{ ys} \equiv p$, 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://arxiv.org/abs/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