Towards an induction principle for nested data types

Peng Fu and Peter Selinger

Dalhousie University

Abstract

A well-known problem in the theory of dependent types is how to handle so-called *nested data types*. These data types are difficult to program and to reason about in total dependently typed languages such as Agda and Coq. In particular, it is not easy to derive a canonical induction principle for such types. Working towards a solution to this problem, we introduce *dependently typed folds* for nested data types. Using the nested data type `Bush` as a guiding example, we show how to derive its dependently typed fold and induction principle. We also discuss the relationship between dependently typed folds and the more traditional higher-order folds.

1 Introduction

Consider the following list data type and its fold function in Agda [1].

```
data List (a : Set) : Set where
  nil : List a
  cons : a -> List a -> List a

foldList : ∀ {a p : Set} -> p -> (a -> p -> p) -> List a -> p
foldList base step nil = base
foldList base step (cons x xs) = step x (foldList base step xs)
```

The keyword `Set` is a kind that classifies types. The function `foldList` has two implicitly quantified type variables `a` and `p`. In Agda, implicit arguments are indicated by braces (e.g., `{a}`), and can be omitted.

The function `foldList` is defined by structural recursion and is therefore terminating. Agda's termination checker automatically checks this. Once `foldList` is defined, we can use it to define other terminating functions such as the following `mapList` and `sumList`. This is similar to using the iterator to define terminating arithmetic functions in System **T** [6, §7].

```
mapList : ∀ {a b : Set} -> (a -> b) -> List a -> List b
mapList f ℓ = foldList nil (λ a r -> cons (f a) r) ℓ

sumList : List Nat -> Nat
sumList ℓ = foldList zero (λ x r -> add x r) ℓ
```

When defining the `mapList` function, if the input list is empty, then we just return `nil`, so the first argument for `foldList` is `nil`. If the input list is of the form `cons a as`, we return `cons (f a) (mapList f as)`, so the second argument for `foldList` is `(λ a r -> cons (f a) r)`, where `r` represents the result of the recursive call `mapList f as`. The function `sumList` is defined similarly, assuming a natural numbers type `Nat` with zero and addition.

We can generalize the type of `foldList` to obtain the following induction principle for lists.

```
indList : ∀ {a : Set} {p : List a -> Set} ->
          (base : p nil) ->
          (step : (x : a) -> (xs : List a) -> p xs -> p (cons x xs)) ->
          (ℓ : List a) -> p ℓ
indList base step nil = base
indList base step (cons x xs) = step x xs (indList base step xs)
```

We can see that the definition of `indList` is almost the same as that of `foldList`. Compared to the type of `foldList`, the type of `indList` is more general as the kind of `p` is generalized from `Set` to `List a -> Set`. We call `p` a *property* of lists. The induction principle `indList` states that to prove a property `p` for all lists, one must first prove that `nil` has the property `p`, and then assuming that `p` holds for any list `xs` as the induction hypothesis, prove that `p` holds for `cons x xs` for any `x`.

We can now use the induction principle `indList` to prove that `mapList` has the same behavior as the usual recursively defined `mapList'` function.

```
mapList' : ∀ {a b : Set} -> (a -> b) -> List a -> List b
mapList' f nil = nil
mapList' f (cons x xs) = cons (f x) (mapList' f xs)

lemma-mapList : ∀ {a b : Set} -> (f : a -> b) -> (ℓ : List a) ->
                mapList f ℓ == mapList' f ℓ
lemma-mapList f ℓ =
  indList {p = λ y -> mapList f y == mapList' f y} refl
    (λ x xs ih -> cong (cons (f x)) ih) ℓ
```

In the proof of `lemma-mapList`, we use `refl` to construct a proof by reflexivity and `cong` to construct a proof by congruence. The latter is defined such that `cong f` is a proof of `x == y -> f x == f y`. The key to using the induction principle `indList` is to specify which property of lists we want to prove. In this case the property is `(λ y -> mapList f y == mapList' f y)`.

To summarize, the fold functions for *ordinary* data types (i.e., non-nested inductive data types such as `List` and `Nat`) are well-behaved in the following sense. (1) The fold functions are defined by well-founded recursion. (2) The fold functions can be used to define a range of terminating functions (including maps). (3) The types of the fold functions can be generalized to the corresponding induction principles.

Nested data types [2] are a class of data types that one can define in most functional programming languages (OCaml, Haskell, Agda). They were initially studied by Bird and Meertens [2]. They have since been used to represent de Bruijn notation for lambda terms [3], and to give an efficient implementation of persistent sequences [7]. In this paper, we will consider the following nested data type.

```
data Bush (a : Set) : Set where
  leaf : Bush a
  cons : a -> Bush (Bush a) -> Bush a
```

According to Bird and Meertens [2], the type `Bush` a is similar to a list where at each step down the list, entries are *bushed*. For example, a value of type `Bush Nat` can be visualized as follows.

```
 bush1 = [ 4,                                       -- Nat
           [ 8, [ 5 ], [ [ 3 ] ] ],             -- Bush Nat
           [ [ 7 ], [ ], [ [ [ 7 ] ] ] ], -- Bush (Bush Nat)
           [ [ [ ], [ [ 0 ] ] ] ]   -- Bush (Bush (Bush Nat))
         ]
```

Here, for readability, we have written `[`x_1`,...,`x_n`]` instead of `cons` x_1 `(cons` x_2 `(...(cons` x_n `leaf)))`.

Unlike ordinary data types such as lists, nested data types are difficult to program with in total functional programming languages. For example, in the dependently typed proof assistant Coq, the `Bush` data type is not definable at all, since it does not pass Coq's strict positivity test. In Agda, `Bush` can be defined as a data type, but writing functions that use this type is not trivial. For example, we must use general recursion (rather than structural recursion) to define the following `hmap` function.

```
hmap : ∀ {b c : Set} -> (b -> c) -> Bush b -> Bush c
hmap f leaf = leaf
hmap f (cons x xs) = cons (f x) (hmap (hmap f) xs)
```

Note that, in contrast to the `mapList'` function for lists, this definition is not structurally recursive because the inner `hmap` is not applied to a subterm of `cons x xs`. Therefore, Agda's termination checker will reject this definition as potentially non-terminating, unless we specify the unsafe `-no-termination` flag.

The following function `hfold` for `Bush` is called a *higher-order fold* in the literature (e.g., [4], [8]). Its definition uses `hmap`.

```
hfold : (b : Set -> Set) ->
        (ℓ : (a : Set) -> b a) ->
        (c : (a : Set) -> a -> b (b a) -> b a) ->
        (a : Set) -> Bush a -> b a
hfold b ℓ c a leaf = ℓ a
hfold b ℓ c a (cons x xs) =
  c a x (hfold b ℓ c (b a) (hmap (hfold b ℓ c a) xs))
```

Observe that the type variable `b` in `hfold` has kind `Set -> Set`, unlike the type variable `p` in `foldList`, which has type `Set`. The higher-order fold `hfold` presents the following challenges. (1) The definition of `hfold` requires the auxiliary function `hmap`, and `hmap` cannot easily be defined from `hfold`. (2) The definition of `hfold`, like that of `hmap`, is not structurally recursive and Agda's termination checker cannot prove it to be total. (3) Although it is possible (see below), it is fairly difficult to define functions such as summation on `Bush`. (4) Unlike the induction principle for lists, it is not clear how to obtain an induction principle for `Bush` from the higher-order fold `hfold`.

Here is the definition of a function `sum` that sums up all natural numbers in a data structure of type `Bush Nat`. Although `sum` is not a polymorphic function, it requires an auxiliary function that is polymorphic and utilizes an argument `k` that is reminiscent of continuation passing style [9].

```
sumAux : (a : Set) -> Bush a -> (k : a -> Nat) -> Nat
sumAux =
  hfold (λ a -> (a -> Nat) -> Nat)
    (λ a k -> zero) (λ a x xs k -> add (k x) (xs (λ r -> r k)))

sum : Bush Nat -> Nat
sum ℓ = sumAux Nat ℓ (λ n -> n)
```

1.1 Contributions

We present a new approach to defining fold functions for nested data types, which we call *dependently typed folds*. For concreteness, we work within the dependently typed language Agda. Dependently typed folds are defined by well-founded recursion, hence their termination is easily confirmed by Agda. Map functions and many other terminating functions can be defined directly from the dependently typed folds. Moreover, the higher-order folds (such as `hfold`) are definable from the dependently typed folds. In addition, the definitions of dependently typed folds can easily be generalized to corresponding induction principles. Thus we can formally reason about programs involving nested data types in a total dependently typed language. While we illustrate these ideas by focusing on the `Bush` example, our approach also works for other kinds of nested data types; see Section 5 for an example.

2 Dependently typed fold for Bush

Let us continue the consideration of the `Bush` data type. The following is the result of evaluating `hmap f bush1`, where `bush1` is the data structure defined in the introduction, and `f : Nat -> b` for some type `b`.

```
  [ f 4,                                          -- b
    [ f 8, [ f 5 ], [ [ f 3 ] ] ],           -- Bush b
    [ [ f 7 ], [ ], [ [ [ f 7 ] ] ] ],  -- Bush (Bush b)
    [ [ [ ], [ [ f 0 ] ] ] ]     -- Bush (Bush (Bush b))
  ]
```

To motivate the definition of the dependently typed fold below, we first consider the simpler question of how to define a map function for `Bush` by structural recursion. The reason our definition of `hmap` in the introduction was not structural is that in order to define the map function for `Bush Nat`, we need to already have the map functions defined for $\texttt{Bush}^n$ `Nat` = `Bush` (`Bush` (... (`Bush Nat`))) for all $n \geq 0$, which seems paradoxical. Our solution is to define a general map function for $\texttt{Bush}^n$, for all $n \geq 0$. First we define a type-level function `NTimes` such that `NTimes n b` = $\texttt{b}^n$:

```
NTimes : (n : Nat) -> (b : Set -> Set) -> Set -> Set
NTimes zero b a = a
NTimes (succ n) b a = b (NTimes n b a)
```

We can now define the following map function for $\texttt{Bush}^n$:

```
nmap : ∀ {a b : Set} -> (n : Nat) -> (a -> b) ->
       NTimes n Bush a -> NTimes n Bush b
nmap zero f x = f x
nmap (succ n) f leaf = leaf
nmap (succ n) f (cons x xs) =
  cons (nmap n f x) (nmap (succ (succ n)) f xs)
```

Note that `nmap 1` corresponds to the map function for `Bush a`. The recursive definition of `nmap` is well-founded because all the recursive calls are on the components of the constructor `cons`. The Agda termination checker accepts this definition of `nmap`.

We are now ready to introduce the dependently typed fold. The idea is to define the fold over the type `NTimes n Bush` simultaneously for all n.

```
nfold : (p : Nat -> Set) ->
        (ℓ : (n : Nat) -> p (succ n)) ->
        (c : (n : Nat) -> p n -> p (succ (succ n)) -> p (succ n)) ->
        (a : Set) -> (z : a -> p zero) ->
        (n : Nat) -> NTimes n Bush a -> p n
nfold p ℓ c a z zero x = z x
nfold p ℓ c a z (succ n) leaf = ℓ n
nfold p ℓ c a z (succ n) (cons x xs) =
  c n (nfold p ℓ c a z n x) (nfold p ℓ c a z (succ (succ n)) xs)
```

The dependently typed fold `nfold` captures the most general form of computing/traversal on the type `NTimes n Bush a`. Similarly to `nmap`, the definition of `nfold` is well-founded. Note that unlike the `hfold` in the introduction, this definition of fold does not require a map function to be defined first. In fact, `nmap` is definable from `nfold`:

```
nmap : ∀ {a b : Set} -> (n : Nat) -> (a -> b) ->
       NTimes n Bush a -> NTimes n Bush b
nmap {a} {b} n f ℓ =
  nfold (λ n -> NTimes n Bush b) (λ n -> leaf) (λ n -> cons) a f n ℓ
```

We can also prove that `nmap 1` satisfies the defining properties of `hmap` from the introduction. Let `hmap'` = `nmap 1`.

```
lemma-nmap : ∀ {a b : Set} -> (f : a -> b) -> (m n : Nat) ->
             (x : NTimes (add m n) Bush a) ->
             nmap (add m n) f x == nmap m (nmap n f) x
lemma-nmap f zero n x = refl
lemma-nmap f (succ m) n leaf = refl
lemma-nmap f (succ m) n (cons x xs) =
  cong2 cons (lemma-nmap f m n x) (lemma-nmap f (succ (succ m)) n xs)

hmap-leaf : ∀ {a b : Set} -> (f : a -> b) -> hmap’ f leaf == leaf
hmap-leaf f = refl

hmap-cons : ∀ {a b : Set} -> (f : a -> b) -> (x : a) ->
            (xs : Bush (Bush a)) ->
            hmap’ f (cons x xs) == cons (f x) (hmap’ (hmap’ f) xs)
hmap-cons f x xs = cong (cons (f x)) (lemma-nmap f 1 1 xs)
```

Many other terminating functions can also be conveniently defined in term of `nfold`. For example, the summation of all the entries in `Bush Nat` and the length function for `Bush` can be defined as follows:

```
sum : Bush Nat -> Nat
sum =
  nfold (λ n -> Nat) (λ n -> zero) (λ n -> add) Nat (λ x -> x) 1

length : (a : Set) -> Bush a -> Nat
length a =
  nfold (λ n -> Nat) (λ n -> zero) (λ n r1 r2 -> succ r2)
    a (λ x -> zero) 1
```

Note that this definition of `sum` is much more natural and straightforward than the one we gave in the introduction.

3 Induction principle for Bush

While there is no obvious induction principle corresponding to the higher-order fold `hfold`, we can easily generalize the dependently typed fold `nfold` to obtain an induction principle for `Bush`. The following function `ind` is related to `nfold` in the same way that the induction principle for `List` is related to its fold function.

```
ind : ∀ {a : Set} -> {p : (n : Nat) -> NTimes n Bush a -> Set} ->
      (base : (x : a) -> p zero x) ->
      (ℓ : (n : Nat) -> p (succ n) leaf) ->
      (c : (n : Nat) -> (x : NTimes n Bush a) ->
        (xs : NTimes (succ (succ n)) Bush a) ->
        p n x -> p (succ (succ n)) xs -> p (succ n) (cons x xs)) ->
      (n : Nat) -> (xs : NTimes n Bush a) -> p n xs
ind base ℓ c zero xs = base xs
ind base ℓ c (succ n) leaf = ℓ n
ind base ℓ c (succ n) (cons x xs) =
  c n x xs (ind base ℓ c n x) (ind base ℓ c (succ (succ n)) xs)
```

Observe that `ind` follows the same structure as `nfold`. The type variable `p` is generalized to a predicate of kind `(n : Nat) -> NTimes n Bush a -> Set`. The type of `ind` specifies how to prove by induction that a property `p` holds for all members of the type `NTimes n Bush a`. More specifically, for the base case, we must show that `p` holds for any `x` of type `NTimes zero Bush a` (which equals `a`), hence `p zero x`. For the leaf case, we must show that `p` holds for `leaf` of type `NTimes (succ n) Bush a`. For the cons case, we assume as the induction hypotheses that `p` holds for some `x` of type `NTimes n Bush a` and some `xs` of type `NTimes (succ (succ n)) Bush a`, and then we must show that `p` holds for `cons x xs`.

With `ind`, we can now prove properties of `nmap`. For example, the following is a proof that `nmap` has the usual identity property of functors.

```
nmap-id : ∀ {a : Set} -> (n : Nat) -> (y : NTimes n Bush a) ->
          nmap n (id a) y == y
nmap-id {a} n y =
  ind {a} {λ n xs -> nmap n (id a) xs == xs} (λ x -> refl) (λ n -> refl)
    (λ n x xs ih1 ih2 -> cong2 cons ih1 ih2) n y
```

We note that the usual way of proving things in Agda is by recursion, relying on the Agda termination checker to prove termination. Our purpose here, of course, is to illustrate that our induction principle is strong enough to prove many properties without needing Agda's recursion. Nevertheless, the above proof is equivalent to the following proof by well-founded recursion.

```
nmap-id' : ∀ {a : Set} -> (n : Nat) -> (y : NTimes n Bush a) ->
           nmap n (id a) y == y
nmap-id' zero y = refl
nmap-id' (succ n) leaf = refl
nmap-id' (succ n) (cons x y) =
  cong2 cons (nmap-id' n x) (nmap-id' (succ (succ n)) y)
```

The first two clauses of `nmap-id'` correspond to the two arguments `(λ n -> refl)` for `nmap-id`. The recursive calls `nmap-id' n x` and `nmap-id' (succ (succ n)) y` in the definition of `nmap-id'` correspond to the inductive hypotheses `ih1` and `ih2` in `nmap-id`.

4 Higher-order folds and dependently typed folds

Comparing `nfold`, the dependently typed fold that was defined in Section 2, to `hfold`, the higher-order fold defined in the introduction, we saw that `nfold` does not depend on `nmap`, and `nmap` can be defined from `nfold`. We also saw that the termination of `nfold` is obvious and that it can be used to define other terminating functions.

In this section, we will show the `hfold` is actually equivalent to `nfold` in the sense that they are definable from each other.

4.1 Defining `hfold` from `nfold`

Using `nfold`, it is straightforward to define `hfold`, because the latter is essentially the former instantiated to the case $n = 1$.

```
hfold : (b : Set -> Set) ->
        (ℓ : (a : Set) -> b a) ->
        (c : (a : Set) -> a -> b (b a) -> b a) ->
        (a : Set) -> Bush a -> b a
hfold b ℓ c a x =
  nfold (λ n -> NTimes n b a) (λ n -> ℓ (NTimes n b a))
    (λ n -> c (NTimes n b a)) a (λ x -> x) 1 x
```

We can prove that this version of `hfold` satisfies the defining properties of the version of `hfold` that was defined in the introduction (and therefore the two definitions agree). Since the proof of `hfold-cons` is rather long, we have omitted it, but the full machine-checkable proof can be found at [5].

```
hfold-leaf : (a : Set) -> (p : Set -> Set) ->
             (ℓ : (b : Set) -> p b) ->
             (c : (b : Set) -> b -> p (p b) -> p b) ->
              hfold p ℓ c a leaf == ℓ a
hfold-leaf a p ℓ c = refl

hfold-cons : (a : Set) -> (p : Set -> Set) ->
             (ℓ : (b : Set) -> p b) ->
             (c : (b : Set) -> b -> p (p b) -> p b) ->
             (x : a) -> (xs : Bush (Bush a)) ->
             hfold p ℓ c a (cons x xs)
             == c a x (hfold p ℓ c (p a) (hmap (hfold p ℓ c a) xs))
hfold-cons a p ℓ c x xs = ...
```

4.2 Defining `nfold` from `hfold`

The other direction is much trickier. In attempting to define `nfold` from `hfold`, the main difficulty is that we must supply a type function `b : Set -> Set` to `hfold`, and this `b` should somehow capture the quantification over natural numbers. Ideally, we would like to define `b` such that $\mathtt{b}^n\,\mathtt{a} = \mathtt{p}\,n$ for all n and some suitable `a`. However, this is clearly impossible, because `p` is an arbitrary type family, which can be defined so that $\mathtt{p}\,0 = \mathtt{p}\,1$ but $\mathtt{p}\,1 \neq \mathtt{p}\,2$. This would imply $\mathtt{a} = \mathtt{b}\,\mathtt{a}$ but $\mathtt{b}\,\mathtt{a} \neq \mathtt{b}^2\,\mathtt{a}$, a contradiction.

Surprisingly, it is possible to work around this by arranging things so that there is a canonical function $\mathtt{b}^n\,\mathtt{a} \to \mathtt{p}\,n$, rather than an equality. This is done by defining the following rather unintuitive type-level function `PS`.

```
PS : (p : Nat -> Set) -> Set -> Set
PS p A = (n : Nat) -> (A -> p n) -> p (succ n)
```

The type `PS p` is special because there is a map `NTimes n (PS p) a` $\to$ `p n`.

```
PS-to-P : (p : Nat -> Set) -> (a : Set) -> (z : a -> p zero) ->
          (n : Nat) -> NTimes n (PS p) a -> p n
PS-to-P p a z zero x = z x
PS-to-P p a z (succ n) hyp = hyp n ih
  where
    ih : NTimes n (PS p) a -> p n
    ih = PS-to-P p a z n
```

So if we set $\mathtt{b} = \mathtt{PS}\,\mathtt{p}$, we have the promised canonical map $\mathtt{b}^n\,\mathtt{a} \to \mathtt{p}\,n$. We can pass this `b` to `hfold` to go from `Bush a` to `PS p a`.

```
fold-PS : (p : Nat -> Set) ->
        (ℓ : (n : Nat) -> p (succ n)) ->
        (c : (n : Nat) -> p n -> p (succ (succ n)) -> p (succ n)) ->
        (a : Set) -> Bush a -> PS p a
fold-PS p ℓ c =
  hfold (PS p) (λ a n tr -> ℓ n)
    (λ a x xs n tr -> c n (tr x) (xs (succ n) (λ f -> f n tr)))
```

Now, provided that we are able to *lift* the function `Bush a -> PS p a` to its nth iteration, i.e., to a function of type `NTimes n Bush a -> NTimes n (PS p) a`, then we will be able to define the dependently typed fold via the following.

```
nfold' : (p : Nat -> Set) ->
         (ℓ : (n : Nat) -> p (succ n)) ->
         (c : (n : Nat) -> p n -> p (succ (succ n)) -> p (succ n)) ->
         (a : Set) -> (z : a -> p zero) ->
         (n : Nat) -> NTimes n Bush a -> p n
nfold' p ℓ c a z n x = PS-to-P p a z n (lift n x)
  where
    lift : (n : Nat) -> NTimes n Bush a -> NTimes n (PS p) a
    lift n x =
      liftNTimes Bush (PS p) (λ a b -> hmap) n (fold-PS p ℓ c) a x
```

The `liftNTimes` function can indeed be defined by induction on natural numbers.

```
liftNTimes : (b c : Set -> Set) ->
             (∀ x y -> (x -> y) -> (b x -> b y)) ->
             (n : Nat) -> (∀ a -> b a -> c a) ->
             (a : Set) -> NTimes n b a -> NTimes n c a
liftNTimes b c m zero f a x = x
liftNTimes b c m (succ n) f a x =
  f (NTimes n c a)
    (m (NTimes n b a) (NTimes n c a) (liftNTimes b c m n f a) x)
```

Finally, we can prove that the function `nfold'` that we just defined behaves identically to the `nfold` that was defined in Section 2. Again, since the proof is rather long and uses several lemmas, we do not reproduce it here. The machine-checkable proof can be found at [5].

```
theorem : ∀ p ℓ c a z n x ->
          nfold p ℓ c a z n x == nfold' p ℓ c a z n x
theorem p ℓ c a z n x = ...
```

5 Nested data types beyond Bush

So far, we have focused on the `Bush` type, but our approach works for arbitrary nested data types, including ones that are defined by mutual recursion. To illustrate this, consider the following pair of mutually recursive data types:

```
data Bob (a : Set) : Set
data Dylan (a b : Set) : Set

data Bob a where
  robert : a -> Bob a
  zimmerman : Dylan (Bob (Dylan a (Bob a))) (Bob a) -> Bob (Dylan a a) -> Bob a

data Dylan a b where
  duluth : Bob a -> Bob b -> Dylan a b
  minnesota : Dylan (Bob a) (Bob b) -> Dylan a b
```

As usual, the higher-order fold is easy to define. There are two separate such folds, one for `Bob` and one for `Dylan`:

```
hfold-bob : (bob : Set -> Set) ->
      (dylan : Set -> Set -> Set) ->
      (rob : ∀ a -> a -> bob a) ->
      (zim : ∀ a -> dylan (bob (dylan a (bob a))) (bob a) -> bob (dylan a a) -> bob a) ->
      (dul : ∀ a b -> bob a -> bob b -> dylan a b) ->
      (min : ∀ a b -> dylan (bob a) (bob b) -> dylan a b) ->
      ∀ a -> Bob a -> bob a

hfold-dylan : (bob : Set -> Set) ->
      (dylan : Set -> Set -> Set) ->
      (rob : ∀ a -> a -> bob a) ->
      (zim : ∀ a -> dylan (bob (dylan a (bob a))) (bob a) -> bob (dylan a a) -> bob a) ->
      (dul : ∀ a b -> bob a -> bob b -> dylan a b) ->
      (min : ∀ a b -> dylan (bob a) (bob b) -> dylan a b) ->
      ∀ a b -> Dylan a b -> dylan a b
```

The dependent fold requires some explanation. Recall that for `Bush`, the only type expressions of interest were of the form $\texttt{Bush}^n\,\texttt{a}$, so we used the natural number n to index these types. In the more general case, we must consider more complicated type expressions such as `Dylan (Bob a) (Dylan a b)`. Therefore, we need to replace the natural numbers with a custom type. We define a type `BobDylanIndex`, which represents expressions built up from type variables and the type constructors `Bob` and `Dylan`.

```
data BobDylanIndex : Set where
  varA : BobDylanIndex
  varB : BobDylanIndex
  BobC : BobDylanIndex -> BobDylanIndex
  DylanC : BobDylanIndex -> BobDylanIndex -> BobDylanIndex
```

We can then give an interpretation function for these type expressions. This plays the role that `NTimes` played in the `Bush` case:

```
I : (Set -> Set) -> (Set -> Set -> Set) -> Set -> Set -> BobDylanIndex -> Set
I bob dylan a b varA = a
I bob dylan a b varB = b
I bob dylan a b (BobC expr) = bob (I bob dylan a b expr)
I bob dylan a b (DylanC expr1 expr2) = dylan (I bob dylan a b expr1) (I bob dylan a b expr2)
```

For example, if

```
i = DylanC (BobC varA) (DylanC varA varB),
```

then

```
I bob dylan a b i = dylan (bob a) (dylan a b).
```

The dependent fold is defined simultaneously for `Bob` and `Dylan`, and in fact for all type expressions that are built from `Bob` and `Dylan`. Its type is the following:

```
nfold : (p : BobDylanIndex -> Set) ->
        (rob : ∀ a -> p a -> p (BobC a)) ->
        (zim : ∀ a -> p (DylanC (BobC (DylanC a (BobC a))) (BobC a))
                    -> p (BobC (DylanC a a)) -> p (BobC a)) ->
        (dul : ∀ a b -> p (BobC a) -> p (BobC b) -> p (DylanC a b)) ->
        (min : ∀ a b -> p (DylanC (BobC a) (BobC b)) -> p (DylanC a b)) ->
        (a b : Set) ->
        (baseA : a -> p varA) ->
        (baseB : b -> p varB) ->
        (∀ i -> I Bob Dylan a b i -> p i)
```

Note that although the types Bob and Dylan are complicated, the corresponding nfold can be systematically derived from their definition. Moreover, as in the case of Bush, the higher-order folds and the dependent fold are definable in terms of each other. In addition, the induction principle, which generalizes nfold, can be easily defined. Full details can be found in the accompanying code [5].

6 Discussion

We think that the equivalence of hfold and nfold is both surprising and useful. The reason it is surprising is because it was informally believed among researchers that hfold is too abstract for most useful programming tasks. The reason it is potentially useful is that in the context of some dependently typed programming languages or proof assistants (such as Coq), when the user writes a data type declaration, the system should automatically derive the appropriate folds and induction principles for the data type. In the case of nested data types, there is currently no universally good way to do this (which is presumably one of the reasons Coq does not support the Bush type). Now on the one hand, we have nfold, which is a practical programming primitive, but its type is not easy to generate from a user-defined data type declaration. For example, even stating the type of nfold requires a reference to an ancillary data type, which is Nat in the case of Bush but can be more complicated for a general nested type. On the other hand, we have hfold, which is not very practical, but its type can be easily read off from a data type declaration. The fact that we have shown nfold to be definable in terms of hfold suggests a solution to this problem: given a data type declaration, the system can generate its corresponding hfold, and then the user can follow a generic recipe to derive the more useful nfold.

7 Conclusion and future work

Using Bush as an example, we showed how to define dependently typed folds for nested data types. Unlike higher-order folds, dependently typed folds can be used to define maps and other terminating functions, and they have analogous induction principles, similar to the folds for ordinary data types. We showed how to reason about programs involving nested data types in Agda. Last but not least, we also showed that dependently typed folds and higher-order folds are mutually definable. This has some potential applications in implementations of dependent type theories, because given a user-defined nested data type, the corresponding higher-order fold can be automatically generated, and then the user can derive the more useful dependent fold by following a generic recipe. All of our proofs are done in Agda, without using any unsafe flag.

Our long term goal is to derive induction principles for *any* algebraic data type (nested or non-nested). There is still a lot of work to be done. In this paper, we only showed how to get the dependently typed fold and induction principle for the single example of Bush. Although our approach also works for other nested data types, we have not yet given a formal characterization of dependently typed folds and their induction principles in the general case. Another research direction is to study the direct relationship between the induction principles (derived from dependently typed folds) and higher-order folds. In the Bush example, it corresponds to asking if we can define ind from hfold, possibly with some extra properties that can also be read off from the data type definition.

Acknowledgements

We thank the referees for their thoughtful comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Air Force Office of Scientific Research under Award No. FA9550-21-1-0041.

References

[1] Agda documentation. https://agda.readthedocs.io/, accessed: 2022-02-15

[2] Bird, R., Meertens, L.: Nested datatypes. In: Mathematics of program construction. pp. 52–67. Springer (1998)

[3] Bird, R., Paterson, R.: De Bruijn notation as a nested datatype. Journal of functional programming **9**(1), 77–91 (1999)

[4] Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2), 200–222 (1999)

[5] Fu, P., Selinger, P.: Agda code accompanying this paper (2023), available as ancillary material from this paper's arXiv page

[6] Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and types, vol. 7. Cambridge University Press Cambridge (1989)

[7] Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of functional programming **16**(2), 197–217 (2006)

[8] Johann, P., Ghani, N.: Initial algebra semantics is enough! In: International Conference on Typed Lambda Calculi and Applications. pp. 207–222. Springer (2007)

[9] Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theor. Comput. Sci. **1**, 125–159 (1975)