Dependently Typed Folds for Nested Data Types

PENG FU, Dalhousie University
PETER SELINGER, Dalhousie University

We present an approach to develop folds for nested data types using dependent types. We call such folds *dependently typed folds*, they have the following properties. (1) Dependently typed folds are defined by well-founded recursion and they can be defined in a total dependently typed language. (2) Dependently typed folds do not depend on maps, map functions and many terminating functions can be defined using dependently typed folds. (3) The induction principles for nested data types follow from the definitions of dependently typed folds and the programs defined by dependently typed folds can be formally verified. (4) Dependently typed folds exist for any nested data types and they can be specialized to the traditional *higher-order folds*. Using various of examples, we show how to program and reason about dependently typed folds. We also show how to obtain dependently typed folds in general and how to specialize them to the corresponding higher-order folds.

Additional Key Words and Phrases: Dependently typed folds, Nested data types, Induction principles, Dependent types, Theorem proving, Formal verification

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

fold : {a p : Set} -> p -> (a -> p -> p) -> List a -> p
fold {a} {p} base step Nil = base
fold {a} {p} base step (Cons x xs) = step x (fold {a} {p} base step xs)
```

The key word `Set` is a kind that classifies types. The function `fold` has two implicit type arguments, they correspond to the implicitly quantified type variables `a` and `p` in the type of `fold`. In Agda, implicit arguments can be supplied explicitly using braces (e.g. `{a}`), sometimes they can be omitted.

The function `fold` is defined by structural recursion and it is terminating. Once `fold` is defined, we can use it to define terminating functions such as the following `map` and `sum`. This is similar to using the iterator to define terminating arithmetic functions in System **T** [Girard et al. 1989, §7].

```
map : {a b : Set} -> (a -> b) -> List a -> List b
map f l = fold Nil (\ a r -> Cons (f a) r) l
sum : List Nat -> Nat
sum l = fold Z (\ x r -> add x r) l
```

When defining the `map` function, if the input list `l` is empty, then we just return `Nil`, so the first argument for `fold` is `Nil`. If the input list `l` is of the form `Cons a as`, then we want to return

[1] Agda documentation: https://agda.readthedocs.io/en/v2.5.3/.

Authors' addresses: Peng Fu, Department of Mathematics and Statistics, Dalhousie University, peng.frank.fu@gmail.com; Peter Selinger, Department of Mathematics and Statistics, Dalhousie University, selinger@mathstat.dal.ca.

2018. XXXX-XXXX/2018/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

`Cons (f a) (map f as)`, so the second argument for `fold` is `\ a r -> Cons (f a) r`, where `r` represents the result of the recursive call `map f as`. The function `sum` is defined similarly.

We can generalize the type of `fold` to obtain the following induction principle for list.

```
ind : {a : Set} {p : List a -> Set} -> p Nil ->
        ((x : a) -> {xs : List a} -> p xs -> p (Cons x xs)) -> (l : List a) -> p l
ind {a} {p} base step Nil = base
ind {a} {p} base step (Cons x xs) = step x {xs} (ind {a} {p} base step xs)
```

Ignoring the implicit arguments, the definition of `ind` is the same as `fold`. Compared to the type of `fold`, the type of `ind` is more general as the kind of `p` is generalized from `Set` to `List a -> Set`, we call such `p` a *property* of the list. The second argument `step` for `ind` has an implicit term argument `{xs : List a}`. The induction principle `ind` states that to prove a property `p` holds for any list `l`, one has to first prove that `Nil` has the property `p`, and then assuming `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 `ind` to prove that `map` behaves the same as the usual recursively defined `map'` (`lemma1`).

```
map' : {a b : Set} -> (a -> b) -> List a -> List b
map' f Nil = Nil
map' f (Cons x xs) = Cons (f x) (map' f xs)

lemma1 : {a b : Set} -> (f : a -> b) -> (l : List a) -> map f l == map' f l
lemma1 {a} {b} f l =
  ind {a} {\ y -> map f y == map' f y} refl (\ x {xs} ih -> cong (Cons (f x)) ih) l

refl : {a : Set} -> {x : a} -> x == x
cong : {a b : Set} -> {m n : a} -> (f : a -> b) -> m == n -> f m == f n
```

In the proof of `lemma1`, we use a notion of equality `==` with the tactic `refl` to construct a reflexivity proof and the tactic `cong` to construct a congruence proof. The key to use the induction principle `ind` is to specify what kind of property on list we want to prove. In this case the property we have in mind is `\ y -> map f y == map' f y`. Thus the type of `ind {a} {\ y -> map f y == map' f y }` is the following.

```
map f Nil == Nil ->
((x : a) -> {xs : List a} -> map f xs == map' f xs ->
      map f (Cons x xs) == Cons (f x) (map' f xs)) ->
(l : List a) -> map f l == map' f l
```

The first two arguments for the above type correspond to the base case and the step case in the inductive proof. For the base case, we just need `refl`. For the step case, the induction hypothesis `ih` has the type `map f xs == map' f xs`, we just need to show `map f (Cons x xs) == Cons (f x) (map' f xs)`. Since `map f (Cons x xs)` can be evaluated to `Cons (f x) (map f xs)`, we finish the proof by a congruence on the induction hypothesis, so `cong (Cons (f x)) ih` is the proof for `map f (Cons x xs) == Cons (f x) (map' f xs)`.

To summarize, the fold functions for *regular* 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. Unfortunately, these properties of folds for regular data types does not directly carry over to nested data types. Consider the following nested data type.

```
data Bush (a : Set) : Set where
  NilB : Bush a
  ConsB :  a -> Bush (Bush a) -> Bush a
```

According to Bird and Meertens [Bird and Meertens 1998, §1], at each step down the list, entries are *bushed*. For example, a value of type `Bush Nat` can be visualized as the following.

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

We can use general recursion to define the following map function and fold function.

```
hmapB : {b c : Set} -> (b -> c) -> Bush b -> Bush c
hmapB f NilB = NilB
hmapB f (ConsB x xs) = ConsB (f x) (hmapB (hmapB f) xs)

hfoldB : {a : Set} -> {p : Set -> Set} ->
         ({b : Set} -> p b) -> ({b : Set} -> b -> p (p b) -> p b) -> Bush a -> p a
hfoldB base step NilB = base
hfoldB base step (ConsB x xs) =
  step x (hfoldB base step (hmapB (hfoldB base step) xs))
```

The fold function `hfoldB` for the nested data type `Bush` is called a *higher-order fold* in the literature (e.g. [Bird and Paterson 1999b], [Johann and Ghani 2007]). Observe that the type variable `p` in `hfoldB` is generalized to kind `Set -> Set`.

The higher-order fold `hfoldB` has the following problems. (1) The definition of `hfoldB` requires the map function `hmapB`, and `hmapB` can not be defined from `hfoldB`. (2) For both `hmapB` and `hfoldB`, Agda's termination checker fails because the use of recursion in both cases are not well-founded. For example, in the second case of `hmapB`, the outer recursive call of `hmapB` is on the structurally smaller argument `xs`, but there is no such indication for the inner recursive call `(hmapB f)`. (3) Although possible (see Section 6), it is not immediately clear how `hfoldB` can be used to define functions such as a summation of all the entries in `Bush Nat`. The most obvious way is to instantiate `p` with `\ x -> Nat` for `hfoldB`, thus `hfoldB {Nat} {\ x -> Nat}` has type `({b : Set} -> Nat) -> ({b : Set} -> b -> Nat -> Nat) -> Bush Nat -> Nat`. We will have to provide a function of the type `({b : Set} -> b -> Nat -> Nat)` as the second argument for `hfoldB {Nat} {\ x -> Nat}`. Such function would need to be defined for any type `b`, so it cannot be useful for defining the summation. (4) Unlike the induction principle for list, it is not clear how to obtain an induction principle for `Bush` from the higher-order fold `hfoldB`.

As a result, in the pioneer works of *generalized folds* for nested data types by Bird and Paterson ([Bird and Paterson 1999b], [Bird and Paterson 1999a]), they have to define generalized folds using maps, and both generalized folds and map functions are defined by general recursion. As for the formal verification, nested data types such as `Bush` can not be declared directly in the dependently typed language Coq[2]. In Agda, although we can declare nested data types such as `Bush`, we cannot easily program and reason about higher-order folds. This is because: (1) The Agda termination checker fails to recognize the termination of higher-order folds defined from general recursion. (2)

[2]Coq user manual: https://coq.inria.fr/refman/

Currently there is no natural formulation of induction principles for nested data types similar to the ones for regular data types.

1.1 Contributions of the paper

We present an approach to define fold functions for nested data types using dependent types inside the total dependently typed language Agda. We call such folds *dependently typed folds*. Dependently typed folds are defined by well-founded recursion, hence their termination is 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 `hfoldB`) are definable from the dependently typed folds. From the definitions of dependently typed folds, we can also obtain the corresponding induction principles. Thus we can formally reason about the programs involving nested data types in a total dependently typed language. In this paper, we illustrate these ideas by focusing on several concrete examples, we also show how to obtain dependently typed folds in general.

The main technical contents of the paper are the following: In Section 2, using the `Bush` data type, we develop the first example of dependently typed folds. In Section 3 and Section 4, as a case study, we show how to define, program and reason about dependently typed folds in Agda using two well-known nested data types from the literature. In Section 5, we give an example to show how to obtain dependently typed folds in general, and we show how to specialize dependently typed folds to the higher-order folds. In Section 6, we discuss related work. In Section 7, we discuss future work and conclude the paper. All the detailed programs for each section are available at https://github.com/Fermat/dependent-fold and checked by Agda 2.5.3.

2 A DEVELOPMENT OF DEPENDENTLY TYPED FOLDS IN A TOTAL TYPE THEORY VIA THE BUSH DATA TYPE

Let us continue the consideration of the `Bush` data type. The following is the result of evaluating `hmapB f bush1`, where `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))
]                 -- Bush (Bush (Bush (Bush (Bush b))))
```

In order to define the map function for `Bush Nat`, we need to already have the map functions defined for $\texttt{Bush}^n$ `Nat` for all $n \geq 0$, which seems paradoxical. A way out is to define a general map function for $\texttt{Bush}^n$, for all $n \geq 0$. First we define $\texttt{Bush}^n$ as the following `NBush`.

```
NTimes : (Set -> Set) -> Nat -> Set -> Set
NTimes p Z s = s
NTimes p (S n) s = p (NTimes p n s)
NBush : Nat -> Set -> Set
NBush = NTimes Bush
```

The function `NTimes` is a type level function defined by pattern-matching on the natural number `n`. The function call `NTimes p n a` returns a type of the form $\texttt{p}^n\ a$. We now define the following map function `mapB` for $\texttt{Bush}^n$, where `mapB (S Z)` corresponds to the map function for `Bush a`.

```
mapB : {a b : Set} -> (n : Nat) -> (a -> b) -> NBush n a -> NBush n b
mapB Z f x = f x
mapB (S n) f NilB = NilB
mapB (S n) f (ConsB x xs) = ConsB (mapB n f x) (mapB (S (S n)) f xs)
```

The recursive definition of mapB is well-founded as all the recursive calls are on the components of the constructor ConsB. The Agda termination checker accepts this definition of mapB. The definition of mapB confirms a general principle in theorem proving: proving a more general lemma may be easier than proving a concrete one. We will use this principle again and again when verifying programs involving nested data types.

From now on, instead of considering the concrete data type Bush, we will focus on its generalized counterpart NBush n. Looking at the definition of mapB, we can view NBush n as a kind of abstract indexed data type that has three constructors. The first constructor has type a -> NBush Z a, corresponding to the first case of mapB. The second constructor NilB has type NBush (S n) a, and the third constructor ConsB has type NBush n a -> NBush (S (S n)) a -> NBush (S n) a.

We now give the following dependently typed fold for the abstract indexed data type NBush n.

```
foldB : {a : Set} -> {p : Nat -> Set} ->
        (a -> p Z) ->
        ((n : Nat) -> p (S n)) ->
        ((n : Nat) -> p n -> p (S (S n)) -> p (S n)) ->
        (n : Nat) -> NBush n a -> p n
foldB base nil cons Z x = base x
foldB base nil cons (S n) NilB = nil n
foldB base nil cons (S n) (ConsB x xs) =
  cons n (foldB base nil cons n x) (foldB base nil cons (S (S n)) xs)
```

The dependently typed fold foldB captures the most general form of computing/traversal on the abstract data type NBush n a. The definition of foldB is well-founded because all the recursive calls of foldB are on the components of ConsB. Observe that the definition of foldB has the same structure as mapB. We can redefine mapB using foldB.

```
mapB : {a b : Set} -> (n : Nat) -> (a -> b) -> NBush n a -> NBush n b
mapB {a} {b} n f l =
  foldB {a} {\ n -> NBush n b} f (\ n -> NilB) (\ n -> ConsB) n l
```

The dependently typed fold foldB allows us to directly define other terminating functions such as the summation of all the entries in Bush Nat and the length function for Bush.

```
sumB : Bush Nat -> Nat
sumB = foldB {Nat} {\ n -> Nat} (\ x -> x) (\ n -> Z) (\ n -> add) (S Z)

lengthB : {a : Set} -> Bush a -> Nat
lengthB {a} = foldB {a} {\ n -> Nat} (\ x -> Z) (\ n -> Z) (\ n r1 r2 -> S r2) (S Z)
```

Comparing the dependently typed fold foldB and the higher-order fold hfoldB in Section 1, we can see that foldB does not depend on map, and mapB can be defined from foldB. The termination of foldB is obvious and it can be used to define other terminating functions. Moreover, the higher-order fold hfoldB is an instance of the dependently typed fold foldB, as we can define hfoldB using foldB.

```
hfoldB : {a : Set} -> {p : Set -> Set} ->
         ({b : Set} -> p b) -> ({b : Set} -> b -> p (p b) -> p b) -> Bush a -> p a
hfoldB {a} {p} base step =
  foldB {a} {\ n -> NTimes p n a} (\ x -> x) (\ n -> base) (\ n -> step) (S Z)
```

Last but not least, we can generalize the type of dependently typed fold `foldB` to obtain the following induction principle `indB`, just like how we obtain the induction principle for `List` from its fold function.

```
indB : {a : Set} -> {p : (n : Nat) -> NBush n a -> Set} ->
       ((x : a) -> p Z x) ->
       ((n : Nat) -> p (S n) NilB) ->
       ((n : Nat) -> {x : NBush n a} -> {xs : NBush (S (S n)) a} ->
               p n x -> p (S (S n)) xs -> p (S n) (ConsB x xs)) ->
       (n : Nat) -> (xs : NBush n a) -> p n xs
indB base nil cons Z xs = base xs
indB base nil cons (S n) NilB = nil n
indB base nil cons (S n) (ConsB x xs) =
  cons n (indB base nil cons n x) (indB base nil cons (S (S n)) xs)
```

Observe that the definition of `indB` is the same as `foldB`, and the type variable `p` is generalized to kind `(n : Nat) -> NBush n a -> Set`. The type of `indB` is specifying how to prove a property `p` holds for any `xs` of type `NBush n a` by induction. More specifically, for the first base case, we need to show `p` holds for any `x` of type `NBush Z a` (which equals `a`), hence `p Z x`. For the second base case, we need to show `p` holds for `NilB` of type `NBush (S n) a`. For the step case, we assume `p` holds for `x` of type `NBush n a` and `xs` of type `NBush (S (S n)) a` as the inducton hypotheses, we need to show `p` holds for `ConsB x xs`.

With `indB`, we can now prove properties about `mapB` and `foldB`. In the following we prove that `mapB` has the usual identity and composition properties.

```
identity : {a : Set} -> (n : Nat) -> (y : NBush n a) -> y == mapB n (\ x -> x) y
identity {a} n y =
  indB {a} {\ n v -> v == mapB n (\ x -> x) v} (\ x -> refl)
   (\ n -> refl) (\ n {x} {xs} ih1 ih2 -> cong2 ConsB ih1 ih2) n y

mapCompose : {a b c : Set} -> (n : Nat) -> (f : b -> c) -> (g : a -> b) ->
 (x : NBush n a) -> mapB n (compose f g) x == mapB n f (mapB n g x)
mapCompose {a} {b} {c} n f g x =
  indB {a} {\ n v -> mapB n (compose f g) v == mapB n f (mapB n g v)}
   (\ v -> refl) (\ n -> refl) (\ n {x1} {xs} ih1 ih2 -> cong2 ConsB ih1 ih2) n x

cong2 : {a b c : Set} -> {m1 n1 : a} -> {m2 n2 : b} ->
        (f : a -> b -> c) -> m1 == n1 -> m2 == n2  -> f m1 m2 == f n1 n2
```

We note that the usual way of proving things in Agda is by recursion, and relying on the Agda termination checker to prove termination. However, since our purpose is to show the strength of induction principles such as `indB`, we do not use recursion at all. All the proofs in this paper are by induction principles.

Let us take a closer look at `identity`. It is a general statement of `mapB n` (includes the special case `mapB (S Z)`). It is also about a property of `y : NBush n a`, i.e. `y` has the property of being equal to `mapB n (\ x -> x) y` for any `n`. So we instantiate the property `p` in `indB` with `\ n v -> v == mapB n (\ x -> x) v`. Thus the type of `indB {a} {\ n v -> v == mapB n (\ x -> x) v}` is the following.

```
((x : a) -> x == mapB Z (\ x -> x) x) ->
((n : Nat) -> NilB == mapB (S n) (\ x -> x) NilB) ->
((n : Nat) -> {x : NBush n a} -> {xs : NBush (S (S n)) a} ->
```

```
      x == mapB n (\ x -> x) x ->
      xs == mapB (S (S n)) (\ x -> x) xs ->
      ConsB x xs == mapB (S n) (\ x -> x) (ConsB x xs)) ->
(n : Nat) -> (xs : NBush n a) -> xs == mapB n (\ x -> x) xs
```

We need to provide three arguments for `indB {a} {\ n v -> v == mapB n (\ x -> x) v}` to obtain a proof of the theorem `(n : Nat) -> (xs : NBush n a) -> xs == mapB n (\ x -> x) xs`. These three arguments correspond to the three cases in the inductive proof, i.e. a case for `NBush Z a`, a case for `NilB` and a case for `ConsB x xs`. In the third case for `ConsB x xs`, we have two induction hypotheses, i.e. `ih1 : x == mapB n (\ x -> x) x` and `ih2 : xs == mapB (S (S n)) (\ x -> x) xs`. A congruence on these two induction hypotheses, i.e. `cong2 ConsB ih1 ih2`, gives us the proof for `ConsB x xs == mapB (S n) (\ x -> x) (ConsB x xs)`. The proof of `mapCompose` is similar to the proof of `identity`.

Recall that Agda does not accept the general recursive definition of `hmapB` in Section 1. Now that we understand that `hmapB` is just `mapB (S Z)`, we can use the induction principle `indB` to show that `mapB (S Z)` has the same computational behavior as `hmapB`.

```
mapNilB : forall {a b : Set} -> (f : a -> b) -> mapB (S Z) f NilB == NilB
mapNilB {a} {b} f = refl

mapConsB : {a b : Set} -> (f : a -> b) -> (x : a) -> (xs : Bush (Bush a)) ->
          mapB (S Z) f (ConsB x xs) == ConsB (f x) (mapB (S Z) (mapB (S Z) f) xs)
mapConsB {a} {b} f x xs = cong (ConsB (f x)) (addMap {a} {b} (S Z) f xs)

addMap : {a b : Set} -> (n : Nat) -> (f : a -> b) -> (x : NBush (add n n) a) ->
          mapB (add n n) f x == mapB n (mapB n f) x
addMap {a} {b} n f x  =
  indB {NBush n a} {\ m v -> mapB (add m n) f v == mapB m (mapB n f) v}
       (\ x -> refl) (\ n -> refl)
       (\ n {x} {xs} ih1 ih2 -> cong2 ConsB ih1 ih2) n x
```

The theorem `mapNilB` corresponds to the first case in the general recursive definition of `hmapB`, the theorem `mapConsB` corresponds to the second case. To prove `mapConsB`, we need to prove a more general lemma `addMap`. The proof of lemma `addMap` is by standard induction, however, coming up with the correct lemma `addMap` requires some effort.

Similarly, we can use `indB` to show that the `hfoldB` defined from `foldB` behaves the same as the one defined by general recursion. The following theorem `foldBNilB` corresponds to the first case in the general recursive definition, and theorem `foldBConsB` corresponds to the second case. We elide the nontrivial proof of the lemma `lemmConsB` (which uses `indB`).

```
foldBNilB : {a : Set} -> {p : Set -> Set} -> (base : {b : Set} -> p b) ->
            (step : {b : Set} -> b -> p (p b) -> p b) ->
            hfoldB {a} {p} base step NilB == base
foldBNilB base step = refl

foldBConsB : {a : Set} -> {p : Set -> Set} -> (base : {b : Set} -> p b) ->
             (step : {b : Set} -> b -> p (p b) -> p b) ->
             (x : a) -> (xs : Bush (Bush a)) ->
             hfoldB base step (ConsB x xs) ==
             step x (hfoldB base step (mapB (S Z) (hfoldB base step) xs))
foldBConsB {a} {p} base step x xs =
```

```
  cong (step x) (lemmConsB {a} {p} (S Z) base step xs)
```

Finally, we use the induction principle `indB` to prove that for any function `f`, if `f` behaves according to `foldBNilB` and `foldBConsB`, i.e. `f base step NilB == base` and `f base step (ConsB x xs) == step x (f base step (mapB (S Z) (f base step) xs)`, then `f` is equal to `hfoldB`. This statement can be formalized as the following.

```
uniqueness :  (f : {a : Set} -> {p : Set -> Set} ->
                   ({b : Set} -> p b) ->
                   ({b : Set} -> b -> p (p b) -> p b) -> Bush a -> p a) ->

              (hp1 : {a : Set} {p : Set -> Set} -> (base : {b : Set} -> p b) ->
                     (step : {b : Set} -> b -> p (p b) -> p b) ->
                      f {a} {p} base step NilB == base) ->

              (hp2 : {a : Set} {p : Set -> Set} -> (base : {b : Set} -> p b) ->
                     (step : {b : Set} -> b -> p (p b) -> p b) ->
                     (x : a) -> (xs : Bush (Bush a)) ->
                     f base step (ConsB x xs) ==
                     step x (f base step (mapB (S Z) (f base step) xs))) ->

               {a : Set} -> {p : Set -> Set} ->
               (base : {b : Set} -> p b) ->
               (step : {b : Set} -> b -> p (p b) -> p b) -> (bush : Bush a) ->
               f {a} {p} base step bush == hfoldB {a} {p} base step bush
uniqueness f hp1 hp2 {a} {p} base step bush =
  indB {a} {\ n v -> lift n (f base step) v == lift n (hfoldB base step) v} ...

lift : {a : Set} {p : Set -> Set}
       (n : Nat) -> (g : {a : Set} -> Bush a -> p a) -> NBush n a -> NTimes p n a
lift {a} {p} Z g x = x
lift {a} {p} (S n) g x = g (mapB (S Z) (lift {a} {p} n g) x)
```

The proof of `uniqueness` is nontrivial and requires all the lemmas and theorems about `foldB` and `mapB` we seen so far together with the helper function `lift`. The complete proof can be found in the supplementary material. The key idea of the proof is that instead of proving the concrete theorem `(bush : Bush a) -> f base step bush == hfoldB base step bush`, we use `lift` to prove a more general one, namely, `(n : Nat) -> (bush : NBush n a) -> lift n (f base step) bush == lift n (hfoldB base step) bush`.

2.1 The indexed representations

We now show that the nested data type `Bush` is in fact definable even in a core type theory without nested data types. Indeed, we can define `NBush` directly as a non-nested indexed data type `BushN` (called the *indexed representation*). The nested data type `Bush` is recoverable as `BushN (S Z)`. This opens the possibility of a user defined nested data type in a surface language, then it can be automatically desugared to a non-nested definition in the underlying type theory while still providing the fold function and induction principle. For example, the total dependently typed language Coq does not accept nested data types such as `Bush` due to the failure of Coq's strict positivity check. So in this case we can work with the indexed representations instead.

Consider the following indexed data type BushN.

```
data BushN : Nat -> Set -> Set where
  Base : {a : Set} -> a -> BushN Z a
  NilBN : {a : Set} -> {n : Nat} -> BushN (S n) a
  ConsBN : {a : Set} -> {n : Nat} ->
           BushN n a -> BushN (S (S n)) a -> BushN (S n) a
```

The BushN data type is indexed by the natural numbers. A value of type `BushN Z a` is of the form `Base x`, where `x : a`. A value of type `BushN (S n) a` can be either a `NilBN`, or `ConsBN x xs` with `x : BushN n a` and `xs : BushN (S (S n)) a`.

The following are the fold function and the induction principle for BushN.

```
foldBN : {a : Set} -> {p : Nat -> Set} ->
         (a -> p Z) ->
         ((n : Nat) -> p (S n)) ->
         ((n : Nat) -> p n -> p (S (S n)) -> p (S n)) ->
         (n : Nat) -> BushN n a -> p n
foldBN base nil cons Z (Base x) = base x
foldBN base nil cons (S n) NilBN = nil n
foldBN base nil cons (S n) (ConsBN x xs) =
  cons n (foldBN base nil cons n x) (foldBN base nil cons (S (S n)) xs)

indBN : {a : Set} -> {p : (n : Nat) -> BushN n a -> Set} ->
        ((x : a) -> p Z (Base x)) ->
        ((n : Nat) -> p (S n) NilBN) ->
        ((n : Nat) -> {x : BushN n a} -> {xs : BushN (S (S n)) a} ->
              p n x -> p (S (S n)) xs -> p (S n) (ConsBN x xs)) ->
        (n : Nat) -> (xs : BushN n a) -> p n xs
```

The definition of `indNB` is exactly the same as `foldBN`. Notice that `foldBN` is almost the same as `foldB` except it actually pattern-matches on all the constructors of the index data type `BushN n a`. We can convert back and forth between `NBush n a` and `BushN n a`.

```
to : {a : Set} -> (n : Nat) -> NBush n a -> BushN n a
to {a} n s =
  foldB {a} {\ n -> BushN n a} Base (\ n -> NilBN) (\ n -> ConsBN) n s
from : {a : Set} -> (n : Nat) -> BushN n a -> NBush n a
from {a} n s =
  foldBN {a} {\ n -> NBush n a} (\ x -> x) (\ n -> NilB) (\ n -> ConsB) n s

toFrom : {a : Set} -> (n : Nat) -> (x : NBush n a) -> from n (to n x) == x
fromTo : {a : Set} -> (n : Nat) -> (x : BushN n a) -> to n (from n x) == x
```

In principle, all the programs and theorems about `NBush n a` can be converted to `BushN n a`. Please see the supplementary material for an example.

2.2 The Church encodings of the indexed representations

We can work with the indexed representations via their Church encodings in the Calculus of Constructions [Coquand and Huet 1988], a minimal total dependent type system that does not provide primitive data types and recursion. As an example, we now define CNBush, the Church-encoded version of BushN.

```
CNBush : Nat -> Set -> Set
CNBush n a = {p : Nat -> Set} ->
             (a -> p Z) ->
             ((n : Nat) -> p (S n)) ->
             ((n : Nat) -> p n -> p (S (S n)) -> p (S n)) -> p n
```

The definition of `CNBush` is impredicative. The kind of `CNBush` should be `Nat -> Set ->`$\texttt{Set}_1$, not `Nat -> Set -> Set` (recall that `Set` in Agda is a shorthand for $\texttt{Set}_0$), due to the quantification of `p : Nat -> Set`. Since Agda does not support impredicative polymorphism, we enable this feature by using the unsafe `--type-in-type` flag. In a language that supports impredicativity (e.g. Coq), defining the Church-encoded `CNBush` is not a problem, we also provide the Coq version of `CNBush` in the supplementary material.

To obtain the Church-encoded `BushN`, we first identify the type `CNBush n a` with the type of `foldBN`, then we define the three constructors of `CNBush` by imitating the three cases of `foldBN`.

```
cbase : {a : Set} -> a -> CNBush Z a
cbase x = \ base nil cons -> base x

cnil : {a : Set} -> (n : Nat) -> CNBush (S n) a
cnil n = \ base nil cons -> nil n

ccons : {a : Set} -> (n : Nat) -> CNBush n a -> CNBush (S (S n)) a -> CNBush (S n) a
ccons n x xs = \ base nil cons -> cons n (x base nil cons) (xs base nil cons)
```

Since the principle of fold is already encoded in the constructors, the following definition of `cfoldB` is essentially an identity function.

```
cfoldB : {a : Set} -> {p : Nat -> Set} ->
         (a -> p Z) ->
         ((n : Nat) -> p (S n)) ->
         ((n : Nat) -> p n -> p (S (S n)) -> p (S n)) ->
         (n : Nat) -> CNBush n a -> p n
cfoldB base nil cons n b = b base nil cons
```

Programming with the Church-encoded `CNBush` is just like programming with `foldBN` for `BushN`. Instead of using `foldBN`, we use `cfoldB`. For example, the following is the map function for `CNBush`.

```
cmapB : {a b : Set} -> (n : Nat) -> (a -> b) -> CNBush n a -> CNBush n b
cmapB {a} {b} n f =
  cfoldB {a} {\ n -> CNBush n b} (\ x -> cbase (f x)) cnil ccons n
```

Note that we do not use any recursion in the definitions of `cfoldB` and `cmapB`.

Although we can program with the Church-encoded `CNBush` using `cfoldB`, we cannot obtain the induction principle from `cfoldB`, as it is well-known that induction is not derivable in the Calculus of Construction ([Coquand 1989], [Geuvers 2001]).

3 CASE STUDY I: DE BRUIJN NOTATION AS THE NESTED DATA TYPE `TERM`

One place in the literature where nested data types are useful is the representation of de Bruijn lambda terms. In this section and the next section, we give an extended case study of two nested data types that are used to represent the de Bruijn lambda terms. The case study demonstrates that the dependently typed folds is sufficient for the purpose of programming and reasoning about nested data types.

Recall that the idea of de Bruijn notation is to use a number to represent a bound variable. The number is the number of binders between the bound variable and its binding site [De Bruijn 1972]. For example, the lambda term $\lambda x.x\ (\lambda y.x\ y\ (\lambda z.x\ y\ z))$ is represented as $\lambda.0\ (\lambda.1\ 0\ (\lambda.2\ 1\ 0))$. This idea can be captured by the following data types (from [Bird and Paterson 1999a]).

```
data Incr (a : Set) : Set where
  Zero : Incr a
  Succ : a -> Incr a

data Term (a : Set) : Set where
  Var : a -> Term a
  App : Term a -> Term a -> Term a
  Lam : Term (Incr a) -> Term a
```

The data type `Term` is a nested data type because the constructor `Lam` requires an argument of *larger* type `Term (Incr a)`, instead of `Term a`. At each level down the constructor `Lam`, a term will gain an additional `Incr` in its type. For example, the following are the representations of the terms $\lambda.0\ (\lambda.1\ 0\ (\lambda.2\ 1\ 0))$ and $\lambda.\lambda.1\ 0\ (S(S`W`))$.

```
term1 : Term Char
term1 = Lam (App (Var Zero)                  -- Var Zero : Term (Incr Char)
                 (Lam (App (App (Var (Succ Zero))
                              (Var Zero))  -- Var Zero : Term (Incr (Incr Char))
                            (Lam (App (App (Var (Succ (Succ Zero)))
                                           (Var (Succ Zero)))
                                      (Var Zero))))))

term2 : Term Char
term2 = Lam (Lam (App (App (Var (Succ Zero)) (Var Zero))
                       (Var (Succ (Succ W)))))
                           -- Var (Succ (Succ W)) : Term (Incr (Incr Char))
```

Notice that each variable in `term1`, `term2` has a type of the form `Term` (`Incr`^{n} `Char`). In a term of the type `Term Char`, the maximum number of `Succ` in a bound variable is strictly less than the number of `Incr` in its type, while the number of `Succ` in a free variable is equal to the number of `Incr` in its type.

3.1 Dependently typed folds for `Incr` and `Term`

Since we will need to manipulate both bound and free variables, we define the following dependently typed fold `foldI` and `mapIncr` function for `Incr`^{n} a.

```
NIncr : Nat -> Set -> Set
NIncr = NTimes Incr

foldI : {a : Set} -> {p : Nat -> Set} -> (n : Nat) ->
        (a -> p Z) ->
        ((m : Nat) -> p (S m)) ->
        ((m : Nat) -> p m -> p (S m)) -> NIncr n a -> p n
foldI Z base zero succ x = base x
foldI (S n) base zero succ Zero = zero n
foldI (S n) base zero succ (Succ x) = succ n (foldI n base zero succ x)
```

```
mapIncr : {a b : Set} -> (n : Nat) -> (a -> b) -> NIncr n a -> NIncr n b
mapIncr {a} {b} n f y =
  foldI {a} {\ n -> NIncr n b} n f (\ m -> Zero) (\ m -> Succ) y
```

Of course, the usual fold function for the regular data type `I` is an instance of `foldI`.

```
foldI' : {a : Set} -> {p : Set} -> p -> (a -> p) -> Incr a -> p
foldI' {a} {p} zero succ = foldI {a} {\ n -> p} (S Z) succ (\ m -> zero) (\ m x -> x)
```

Let a be a fixed constant type. We say `y : NIncr n a` is *closed* if `y` only consists of `Zero` and `Succ`, otherwise we say `y` is *open*. If `y : NIncr n a` is open, then it must be of the form $\mathtt{Succ}^n$ `x`, where `x : a`. The behavior of `mapIncr` is subtle. Suppose `y : NIncr n a` is open, then the result of `mapIncr l Succ y` (where $l \leq n$) will be `Succ y`. If `y : NIncr n a` is closed, then `y` must be of the form $\mathtt{Succ}^m$ `Zero`, where $m < n$. In this case `mapIncr l Succ y` will be evaluated to `y` if $m < l \leq n$, or evaluated to `Succ y` if $l \leq m$. We can test this by comparing the following `num1` and `num2`. The program `num1` is evaluated to `Succ (Succ Zero)`, while `num2` is evaluated to `Succ (Succ (Succ Zero))`.

```
num0 : NIncr (S (S (S (S (S Z))))) Char
num0 = Succ (Succ Zero)
num1 : NIncr (S (S (S (S (S (S Z)))))) Char
num1 = mapIncr {Incr (Incr Char)} {Incr (Incr (Incr Char))}
       (S (S (S Z))) Succ num0
num2 : NIncr (S (S (S (S (S (S Z)))))) Char
num2 = mapIncr {Incr (Incr (Incr Char))} {Incr (Incr (Incr (Incr Char)))}
       (S (S Z)) Succ num0
```

We now define the following dependently typed fold for `Term` ($\mathtt{Incr}^n\ a$).

```
foldT : {a : Set} -> {p : Nat -> Set} -> (n : Nat) ->
        ((m : Nat) -> NIncr m a -> p m) ->
        ((m : Nat) -> p m -> p m -> p m) ->
        ((m : Nat) -> p (S m) -> p m) -> Term (NIncr n a) -> p n
foldT {a} {p} n var app lam (Var x) = var n x
foldT {a} {p} n var app lam (App x1 x2) =
   app n (foldT {a} {p} n var app lam x1) (foldT {a} {p} n var app lam x2)
foldT {a} {p} n var app lam (Lam x) = lam n (foldT {a} {p} (S n) var app lam x)
```

The function `foldT` traverses the structure of `Term`, replaces the constructors `Var`, `App` and `Lam` with `var`, `app` and `lam`, increases the number `n` only when traversing under the `Lam` constructor.

The higher-order fold for `Term` is an instance of the dependently typed fold `foldT`.

```
hfoldT : {a : Set} -> {p : Set -> Set} ->
         ({a : Set} -> p a) ->
         ({a : Set} -> p a -> p a -> p a) ->
         ({a : Set} -> p (Incr a) -> p a) -> Term a -> p a
hfoldT {a} {p} var app lam =
  foldT {a} {\ n -> p (NTimes Incr n a)} Z (\ m a -> var) (\ m -> app) (\ m -> lam)
```

The following is the map function defined from `foldT` and `mapIncr`.

```
mapT : {a b : Set} -> (n : Nat) -> (a -> b) -> Term (NIncr n a) -> Term (NIncr n b)
mapT {a} {b} n f y =
  foldT {a} {\ n -> Term (NIncr n b)} n
    (\ n v -> Var (mapIncr n f v)) (\ n -> App) (\ n -> Lam) y
```

When defining the function `mapT`, we use `foldT` to traverse the abstract data type `Term (NIncr n a)`, and perform action at the leaves, where we apply the `mapIncr` function.

For a variable `Var x`, we say it is *open*/*closed* iff `x` is *open*/*closed*. Note that an open variable is necessarily a free variable, but a closed variable can be either bound or free. For example, the variable `Var Zero : Term (Incr Char)` is a free and closed variable because it is not bound by any `Lam` constructor, while the variable `Var Zero : Term (Incr Char)` in `Lam (Var Zero) : Term Char` is closed and bound. Let `y` be a term of type `Term (NIncr n a)` for some fixed constant type `a`. The function call `mapT n f y` will map `f` to all the open variables in `y` and leave the closed variables (including free and closed variables) unchanged. On the other hand, the function call `mapT Z f y` will map the function `f` to all the free variables (open or closed) in `y`, while leaving the bound variables unchanged. When `n = Z` and `y : Term (Incr Z a)`, there are no closed free variables in `y`, thus the free variables coincide with the open variables, the bound variables coincide with the closed variables.

3.2 Programming with dependently typed folds and maps

The following programs are the printing functions defined from `foldT` and `foldI`.

```
showT : Term String -> String
showT y = foldT {String} {\ n -> String} Z
                showI
                (\ m x y -> ` LP ` ++ x ++ ` EMP ` ++ y ++ ` RP `)
                (\ m x -> ` L ` ++ x) y
showI : (m : Nat) -> NIncr m String -> String
showI m y = foldI {String} {\ n -> String} m
            (\ x -> x) (\ m -> ` Ze `) (\ m x -> ` Su ` ++ x) y
showTC : Term Char -> String
showTC x = showT (mapT Z (\ x -> Cons x Nil) x)
```

In the definition of `showTC`, we first convert all the free variables in the term `x` to `String`[3], and then apply `showT` to the resulting term.

We now define the abstraction function `abst` that takes a name x and a term t, returns the abstracted term $\lambda x.t$. We assume there is a generic comparison function[4] `cmp : {a : Set} -> a -> a -> Bool` and it satisfies `axiom1 : {a : Set} -> (x x1 : a) -> cmp x x1 == True -> x == x1`. We can express these two assumptions as postulates in Agda.

```
abst : {a : Set} -> a -> Term a -> Term a
abst x t = Lam (mapT Z (match x) t)
match : {a : Set} -> a -> a -> Incr a
match {a} a1 a2 = foldBool {Incr a} Zero (Succ a2) (cmp a1 a2)
foldBool : {p : Set} -> p -> p -> Bool -> p
foldBool x y True = x
foldBool x y False = y
```

The function `match` compares `a1` with `a2`, if they are equal, returns `Zero`, else returns `Succ a2`. So the function call `abst x t` will look at all the free variables in `t`, replace a free variable to `Zero` if it is equal to `x`, otherwise add an extra `Succ` to it.

A beta-redex is a term of the form `App (Lam t) s : Term a`. To perform a beta reduction for `App (Lam t) s`, we need to substitute the variables bound by `Lam` in `t` with `s`. The following actions are required to define the subsitution. (1) For each variable bound by the `Lam` inside `t`, we

[3]The data types `String` and `Char` are user-defined.

[4]The comparison function here is similar to the equality method in the `Eq` type class in Haskell.

need to replace it by `s`. (2) For each free variable in `t`, we need to decrease a `Succ` in it. (3) For each free variable in `s`, we need to increase a `Succ` for it whenever `s` is traversing under a binder in `t`. (4) All the other bound variables in `t` remain unchanged.

In the following we define the substitution function `subst` and the function `redex`.

```
redex : {a : Set} -> Term a -> Term a
redex (App (Lam t) s) = subst Z s t
redex t = t

subst : {a : Set} -> (n : Nat) -> Term a ->
        Term (NIncr n (Incr a)) -> Term (NIncr n a)
subst {a} n s t =
  foldT {Incr a} {\ n -> Term (NIncr n a)} n
       (\ m -> varcase m s) (\ m -> App) (\ m -> Lam) t

varcase : {a : Set} -> (n : Nat) -> Term a -> NIncr n (Incr a) -> Term (NIncr n a)
varcase {a} n s v =
  foldI {Incr a} {\ n -> Term (NIncr n a)} n
   (h s)
   (\ m -> Var Zero)              -- Action (4)
   (\ m r -> mapT Z Succ r)       -- Action (3)
   v
  where h : {a : Set} -> Term a -> Incr a -> Term a
        h s Zero = s              -- Action (1)
        h s (Succ b) = Var b      -- Action (2)
```

The substitution function `subst` is generalized to allow substitution for any term of the type `Term (NIncr n (Incr a))`, the definition of `redex` uses `subst Z`. In the definition of `subst`, all the actions happen in the variable case, we use `foldT` just for traversing to the leaves of `Term (NIncr n (Incr a))`. The `varcase` function inspects on the variable `v` and performs actions (1)–(4).

3.3 Reasoning with the induction principles `indI` and `indT`

Based on the dependently typed fold `foldT` and `foldI`, we can obtain the following induction principles. The induction principles follow the same definitions as `foldT` and `foldI`, except their types are more general. They provide a way to prove a property holds for any `Term (NIncr n a)` and `NIncr n a`.

```
indT : {a : Set} -> {p : (n : Nat) -> Term (NIncr n a) -> Set} ->
       (n : Nat) -> ((m : Nat) -> (v : NIncr m a) -> p m (Var v)) ->
       ((m : Nat) -> {v1 v2 : Term (NIncr m a)} ->
              p m v1 -> p m v2 -> p m (App v1 v2)) ->
       ((m : Nat) -> {v : Term (NIncr (S m) a)} -> p (S m) v -> p m (Lam v)) ->
       (v : Term (NIncr n a)) -> p n v

indI : {a : Set} -> {p : (n : Nat) -> NIncr n a -> Set} -> (n : Nat) ->
       ((x : a) -> p Z x) ->
       ((m : Nat) -> p (S m) Zero) ->
       ((m : Nat) -> {x : NIncr m a} -> p m x -> p (S m) (Succ x)) ->
       (v : NIncr n a) -> p n v
```

Now we can use induction to prove the following `thm1`, which states $(\lambda x.t)\ x =_\beta t$.

```
thm1 : {a : Set} -> (x : a) -> (t : Term a) ->
         redex (App (abst x t) (Var x)) == t
thm1 {a} x t =
    indT {a} {\ n v -> subst n (Var x) (mapT n (match x) v) == v} Z
      (thm0 x)
      (\ m {v1} {v2} ih1 ih2 -> cong2 App ih1 ih2)
      (\ m {v} ih -> cong Lam ih) t

thm0 : {a : Set} -> (x : a) -> (m : Nat) -> (v : NIncr m a) ->
         subst m (Var x) (mapT m (match x) (Var v)) == Var v
thm0 {a} x m v =
    indI {a} {\ n u -> subst n (Var x) (mapT n (match x) (Var u)) == Var u} m
      (\ x1 -> lemm0 (cmp x x1) (lemm1 x x1) (lemm2 x x1))
      (\ m -> refl) (\ m {x1} ih -> cong (mapT Z Succ) ih) v

lemm0 : {p : Set} -> (x : Bool) -> (x == True -> p) -> (x == False -> p) -> p
lemm1 : {a : Set} -> (x x1 : a) -> (cmp x x1 == True) ->
        subst Z (Var x) (mapT {a} {Incr a} Z (match {a} x) (Var x1)) == Var x1
lemm2 : {a : Set} -> (x x1 : a) -> (cmp x x1 == False) ->
        subst Z (Var x) (mapT {a} {Incr a} Z (match {a} x) (Var x1)) == Var x1
```

The inductive cases for `thm1` are straightforward, the only nontrivial case is the variable case, which needs `thm0`. For `thm0`, the only nontrivial case is the base case, where we need a proof of `(x x1 : a) -> subst Z (Var x) (mapT Z (match x) (Var x1)) == Var x1`. We prove this by considering both `cmp x x1 == True` (`lemm1`) and `cmp x x1 == False` (`lemm2`). The proofs of `lemm1` (requires `axiom1`), `lemm2` and `lemm0` are straightforward.

We now consider the theorems `mapTfuseZ` and `mapSubst`, which are properties of `mapT` and `subst` and will be needed for proving the theorem `cvtThm` in the Section 4.

The following `mapTfuseZ` theorem states that for any `s : Term (NIncr n a)`, the following two results are equal: (1) first add an extra `Succ` to all the free variables, and then add an extra `Succ` to all the open variables. (2) first add an extra `Succ` to all the open variables, and then add an extra `Succ` to all the free variables.

```
mapTfuseZ : {a : Set} -> (n : Nat) -> (s : Term (NIncr n a)) ->
            mapT {a} {Incr a} (S n) Succ
                 (mapT {NIncr n a} {Incr (NIncr n a)} Z Succ s) ==
            mapT {Incr (NIncr n a)} {Incr (Incr (NIncr n a))} Z Succ
                 (mapT {a} {Incr a} n Succ s)
mapTfuseZ {a} n s = mapTfuse {a} Z n s

mapTfuse : {a : Set} -> (m n : Nat) -> (s : Term (NIncr (add m n) a)) ->
            mapT {a} {Incr a} (S (add m n)) Succ
              (mapT {NIncr n a} {Incr (NIncr n a)} m Succ s) ==
            mapT {NIncr (S n) a} {Incr (NIncr (S n) a)} m Succ
              (mapT {a} {Incr a} (add m n) Succ s)
```

The `mapTfuseZ` theorem requires us to prove a more general lemma `mapTfuse`, finding this lemma takes a lot of effort, but it can be proved by standard induction using `indT` and `indI`.

The following `mapSubst` theorem states how `mapT Z Succ` commutes with `subst m`. Again, `mapSubst` requires a nontrivial general lemma `mapSubst'`. The lemma `mapSubst'` can be proved by induction using the lemma `mapTfuseZ`.

```
mapSubst : {a : Set} -> (m : Nat) -> (s : Term a) -> (t : Term (NIncr (S m) a)) ->
           mapT Z Succ (subst m s t) == subst (S m) s (mapT Z Succ t)
mapSubst {a} m s t = mapSubst' {a} Z m s t
mapSubst' : {a : Set} -> (n m : Nat) -> (s : Term a) ->
            (t : Term (NIncr (S (add n m)) a)) ->
            mapT {NIncr m a} {NIncr (S m) a} n Succ (subst (add n m) s t) ==
            subst (S (add n m)) s
               (mapT {NIncr (S m) a} {NIncr (S (S m)) a} n Succ t)
```

4 CASE STUDY II: DE BRUIJN NOTATION AS THE NESTED DATA TYPE TERME

In this section we consider representing the de Bruijn lambda terms using the following nested data type.

```
data TermE (a : Set) : Set where
  VarE : a -> TermE a
  AppE : TermE a -> TermE a -> TermE a
  LamE : TermE (Incr (TermE a)) -> TermE a
```

This data type is already motivated by Bird and Paterson [Bird and Paterson 1999a]. Recall that when substituting term s for x in term t, i.e. $[s/x]t$, we need to perform the actions (1)–(4) (Section 3). The action (3) requires traversing the term s to add an additional `Succ` when the substitution is going under a binder. This has two drawbacks, namely, traversing s takes additional time and prevents the sharing of s.

For example, the redex $(\lambda.0\ (\lambda.1\ 0\ (\lambda.2\ 1\ 0)))\ (\lambda.0\ (S\ 'W'))$ will be reduced to the following.

$$\underline{(\lambda.0\ (S\ 'W'))}\ (\lambda.\underline{(\lambda.0\ (S(S\ 'W')))}\ 0\ (\lambda.\underline{(\lambda.0\ (S(S(S\ 'W'))))}\ 1\ 0))$$

Not only we traverse the term $(\lambda.0\ (S\ 'W'))$ three times to add S, but also we have three different copies of $(\lambda.0\ (S\ 'W'))$ in the resulting term. A more efficient implementation would avoid such traversal and allow us to obtain the following term.

$$\texttt{term0} = \underline{(\lambda.0\ (S\ 'W'))}\ (\lambda.(S(\underline{\lambda.0\ (S\ 'W')}))\ 0\ (\lambda.(S(S(\underline{\lambda.0\ (S\ 'W')})))\ 1\ 0))$$

In `term0`, we have three same copies of $(\lambda.0\ (S\ 'W'))$. To enable such representation, we would need to allow `Succ` to be applied to a term, hence the type for the `LamE` constructor. The following `term1` is the concrete representation of $(\lambda.0\ (S\ 'W'))$ and `term2` is the representation of `term0`.

```
term1 : TermE Char
term1 = LamE (AppE (VarE Zero) (VarE (Succ (VarE W))))

term2 : TermE Char
term2 = AppE term1
          (LamE (AppE (AppE (VarE (Succ term1))
                            (VarE Zero))
                      (LamE (AppE (AppE (VarE (Succ (VarE (Succ term1))))
                                        (VarE (Succ (VarE Zero))))
                                  (VarE Zero)))))
```

4.1 The dependently typed fold `foldE`

At each level down the constructor `LamE`, a term gains an additional type-constructor `\ x -> Incr (TermE x)` in its type. So we will focus on the type of the form `TermE` (`IncrTermE`^{n} a), where `IncrTermE`^{n} means the n-th iteration of `\ x -> Incr (TermE x)`.

We define the following dependently typed fold `foldE` for `TermE` (`IncrTermE`^{n} a).

```
IncrTermE : Nat -> Set -> Set
IncrTermE = NTimes (\ x -> Incr (TermE x))

foldE : {a : Set} -> {p : Nat -> Set} -> (n : Nat) ->
         (a -> p Z) ->
         ((m : Nat) -> p (S m)) ->
         ((m : Nat) -> p m -> p (S m)) ->
         ((m : Nat) -> p m -> p m -> p m) ->
         ((m : Nat) -> p (S m) -> p m) -> TermE (IncrTermE n a) -> p n
foldE {a} {p} Z varBase varZero varSucc app abs (VarE x) = varBase x
foldE {a} {p} (S n) varBase varZero varSucc app abs (VarE Zero) = varZero n
foldE {a} {p} (S n) varBase varZero varSucc app abs (VarE (Succ x)) =
  varSucc n (foldE {a} {p} n varBase varZero varSucc app abs x)
foldE {a} {p} n varBase varZero varSucc app lam (LamE x) =
  lam n (foldE {a} {p} (S n) varBase varZero varSucc app lam x)
foldE {a} {p} n varBase varZero varSucc app abs (AppE x x') =
  app n (foldE {a} {p} n varBase varZero varSucc app abs x)
        (foldE {a} {p} n varBase varZero varSucc app abs x')
```

Observe that `foldE` is well-founded and the abstract indexed data type `TermE` (`IncrTermE`^{n} a) has five constructors, corresponding to the five cases in the definition of `foldE`. There are two variable cases for `TermE` (`IncrTermE`$^{n+1}$ a), i.e. `VarE Zero` and `VarE (Succ x)`, where `x` is of the type `TermE` (`IncrTermE`^{n} a). So there is a recursive call in the case for `VarE (Succ x)`, which traverses the term `x`.

The following is the map function defined from `foldE`.

```
mapE : {a b : Set} -> (n : Nat) -> (a -> b) ->
       TermE (IncrTermE n a) -> TermE (IncrTermE n b)
mapE {a} {b} n f x = foldE {a} {\ n -> TermE (IncrTermE n b)} n
                      (\ x -> VarE (f x))
                      (\ m -> VarE Zero)
                      (\ m r -> VarE (Succ r))
                      (\ m -> AppE)
                      (\ m -> LamE) x
```

The function call `mapE n f t` keeps every constructor in `t` unchanged and only applies the function `f` in the case of `VarE x`, where `x : a`. The map function for `TermE a` is `mapE Z`.

The dependently typed fold `foldE` can be specialized to the higher-order fold for `TermE`.

```
hfoldE : {a : Set} -> {p : Set -> Set} ->
          ({a : Set} -> a -> p a) ->
          ({a : Set} -> p a -> p a -> p a) ->
          ({a : Set} -> p (Incr (p a)) -> p a) -> TermE a -> p a
hfoldE {a} {p} var app lam =
  foldE {a} {\ n -> p (NTimes (\ y -> Incr (p y)) n a)}
```

```
    Z var (\ m -> var Zero) (\ m r -> var (Succ r)) (\ m -> app) (\ m -> lam)
```

The higher-order fold `hfoldE` replaces the constructors `VarE`, `AppE` and `LamE` to `var`, `app` and `lam` while keeping the constructors `Zero` and `Succ` unchanged.

4.2 Programming with foldE

The following function `redexE` is for reducing a beta-redex. The substitution function `substE` follows the same idea as the `subst` function in Section 3, except that it does not perform action (3) due to the optimization of the `TermE` data type.

```
redexE : {a : Set} -> TermE a -> TermE a
redexE (AppE (LamE t) s) = substE Z s t
redexE t = t

substE : {a : Set} -> (n : Nat) -> TermE a ->
         TermE (IncrTermE n (Incr (TermE a))) -> TermE (IncrTermE n a)
substE {a} n s = foldE {Incr (TermE a)} {\ n -> TermE (IncrTermE n a)} n
                  (base s)
                  (\ m -> VarE Zero)        -- Action (4)
                  (\ m r -> VarE (Succ r))  -- No need for action (3)
                  (\ m -> AppE)
                  (\ m -> LamE)
  where base : TermE a -> Incr (TermE a) -> TermE a
        base s Zero = s                     -- Action (1)
        base s (Succ x) = x                 -- Action (2)
```

The following abstraction function `abstE x t` binds the free variable `x` in `t`. The definition of `abstE` follows the same idea as `abst` (Section 3).

```
abstE : {a : Set} -> a -> TermE a -> TermE a
abstE x t = LamE (mapE Z (matchE x) t)
matchE : {a : Set} -> a -> a -> Incr (TermE a)
matchE {a} a1 a2 = foldBool {Incr (TermE a)} Zero (Succ (VarE a2)) (cmp a1 a2)
```

Let us now consider converting a lambda expression of the type `TermE a` to a lambda expression of the type `Term a`. We will define a generalized conversion function to convert `TermE (IncrTermE n a)` to `Term (NIncr n a)` for any `n`. The main difference between `TermE (IncrTermE n a)` and `Term (NIncr n a)` is that `Succ` can apply to a term `t : TermE (IncrTermE n a)`, so there are terms of the form `VarE (Succ t) : TermE (IncrTermE (S n) a)`. The key idea of the conversion is that when converting a term of the form `VarE (Succ t)`, we first convert the term `t` of type `TermE (IncrTermE n a)` to a term `t'` of type `Term (NIncr n a)`, and then apply the constructor `Succ` to all the free variables in `t'`, i.e. `mapT Z Succ t'`.

```
cvtE : {a : Set} -> (n : Nat) -> TermE (IncrTermE n a) -> Term (NIncr n a)
cvtE {a} n t = foldE {a} {\ n -> Term (NIncr n a)} n
                (\ a -> Var a)
                (\ m -> Var Zero)
                (\ m t' -> mapT Z Succ t')
                (\ m -> App)
                (\ m -> Lam) t
```

The function `cvtE` converts `VarE x`, `VarE Zero`, `AppE` and `LamE` to `Var x`, `Var Zero`, `App` and `Lam` accordingly. The only subtle case is how to convert `VarE (Succ t)`, which we have explained.

4.3 Reasoning with the induction principle indE

The following is the induction principle for `TermE (IncrTermE n a)`, we elide the definition as it is the same as `foldE`.

```
indE : {a : Set} -> {p : (n : Nat) -> TermE (IncrTermE n a) -> Set} -> (n : Nat) ->
       ((x : a) -> p Z (VarE x)) ->
       ((m : Nat) -> p (S m) (VarE Zero)) ->
       ((m : Nat) -> {r : TermE (IncrTermE m a)} ->
                 p m r -> p (S m) (VarE (Succ r))) ->
       ((m : Nat) -> {x1 x2 : TermE (IncrTermE m a)} ->
                 p m x1 -> p m x2 -> p m (AppE x1 x2)) ->
       ((m : Nat) -> {x : TermE (IncrTermE (S m) a)} ->
                 p (S m) x -> p m (LamE x)) ->
       (v : TermE (IncrTermE n a)) -> p n v
```

The induction principle `indE` gives us a way to prove a property `p : (n : Nat) -> TermE (IncrTermE n a) -> Set` holds for any `v : TermE (IncrTermE n a)`. To obtain such proof, we first have to prove `p` holds for `VarE x : TermE (IncrTermE Z a)` and `VarE Zero : TermE (IncrTermE (S m) a)`. They correspond to the two arguments `(x : a) -> p Z (VarE x)` and `(m : Nat) -> p (S m) (VarE Zero)` for `indE`. Then we will need to prove three inductive cases, they correspond to the other three arguments for `indE`. For example, one inductive case requires us to prove `p m (LamE x)`, using the inductive hypothesis `p (S m) x`.

We now use `indE` to prove that $(\lambda x.t)\ x =_{\beta} t$ holds for our definition of `redexE`. The proof is similar to the proof of theorem `thm1` for `redex` in Section 3.

```
thm1 : {a : Set} -> (x : a) -> (t : TermE a) ->
       redexE (AppE (abstE x t) (VarE x)) == t
thm1 {a} x t =
  indE {a} {\ n v -> substE n (VarE x) (mapE n (matchE x) v) == v} Z
      (lem3 x) (\ m -> refl) (\ m {r} ih -> cong (\ y -> VarE (Succ y)) ih)
      (\ m ih1 ih2 -> cong2 AppE ih1 ih2) (\ m ih -> cong LamE ih) t
lem3 : {a : Set} -> (x y : a) ->
         substE Z (VarE x) (mapE Z (matchE x) (VarE y)) == VarE y
```

To convince ourself that the conversion function `cvtE` is well-behaved, we use `indE` prove that the conversion function commutes with the substitution, i.e. $\texttt{cvtE}([s/x]t) = [(\texttt{cvtE}\ s)/x](\texttt{cvtE}\ t)$.

```
cvtThm : {a : Set} -> (n : Nat) -> (s : TermE a) ->
         (t : TermE (IncrTermE (S n) a)) ->
         cvtE {a} n (substE n s t) == subst n (cvtE {a} Z s) (cvtE {a} (S n) t)
cvtThm {a} n s t =
  indE {Incr (TermE a)}
       {\ n v -> cvtE {a} n (substE n s v) ==
                 subst n (cvtE {a} Z s) (cvtE {a} (S n) v)} n
       (\ x -> lemmVar x s)   -- case VarE x
       (\ m -> refl)          -- case VarE Zero
       (\ m {r} ih ->         -- case VarE (Succ r)
        equational cvtE {a} (S m) (substE (S m) s (VarE (Succ r)))
          equals cvtE {a} (S m) (VarE (Succ (substE m s r))) by refl
          equals mapT {NIncr m a} {Incr (NIncr m a)} Z Succ
                      (cvtE {a} m (substE m s r))
```

```
          by refl
       equals mapT {NIncr m a} {Incr (NIncr m a)} Z Succ
                   (subst m (cvtE {a} Z s) (cvtE {a} (S m) r))
          by cong (mapT {NIncr m a} {Incr (NIncr m a)} Z Succ) ih
       equals subst (S m) (cvtE {a} Z s)
               (mapT {NIncr (S m) a} {Incr (Incr (NIncr m a))} Z Succ
                    (cvtE {a} (S m) r))
          by mapSubst {a} m (cvtE {a} Z s) (cvtE {a} (S m) r)
       equals subst (S m) (cvtE {a} Z s)
                   (cvtE {a} (S (S m)) (VarE (Succ r)))
          by refl)
    (\ m {x1} {x2} ih1 ih2 -> cong2 App ih1 ih2) -- case AppE x1 x2
    (\ m {x} ih -> cong Lam ih)                  -- case LamE x
    t
```

The cases for `LamE x`, `AppE x1 x2` and `VarE Zero` are straightforward. For the case of `VarE x`, we need the following `lemmVar` and `lemmM`.

```
lemmVar : {a : Set} -> (x : Incr (TermE a)) -> (s : TermE a) ->
          cvtE Z (substE Z s (VarE x)) == subst Z (cvtE Z s) (cvtE (S Z) (VarE x))
lemmVar {a} Zero s = refl
lemmVar {a} (Succ x) s = lemmM {a} Z (cvtE Z s) (cvtE Z x)

lemmM : {a : Set} -> (m : Nat) -> (s : Term a) -> (t : Term (NIncr m a)) ->
        t == subst m s (mapT {a} {Incr a} m Succ t)
```

The lemma `lemmVar` is just a special case of `cvtThm` for `VarE x`, it needs the lemma `lemmM`, which can be proved by straightforward induction.

For the case of `VarE (Succ r)`, we need the induction hypothesis `ih` and the `mapSubst` theorem we proved in Section 3. We also use a custom equational reasoning tactic in the following form [5].

```
equational t1
  equals t2 by p1
  equals t3 by p2 ...
  equals tn by p(n-1)
```

This means we prove the following $t_1 \overset{p_1}{=} t_2 \overset{p_2}{=} t_3 \ldots \overset{p_{n-1}}{=} t_n$.

4.4 Discussion

We have shown that not only we are able to program all the functions for manipulating de Bruijn lambda terms like Bird and Paterson did [Bird and Paterson 1999a], but also we are able to reason about the programs formally using induction principles. We are working in a terminating dependently typed language, whereas Bird and Paterson worked in Haskell, a language with general recursion. Benefiting from the flexibility of dependently typed folds, we also make some minor improvements. The following are Bird and Patterson's implementation of `cvtE` and `cvtBodyE` functions in Haskell.

```
cvtE :: TermE a -> Term a
cvtE = gfoldE Var App (Lam . joinT . mapT distT) id

cvtBodyE :: TermE (Incr (TermE a)) -> Term (Incr a)
```

[5] Please see the file `Equality.agda` in the supplementary material for details.

```
cvtBodyE = joinT . mapT distT . cvtE . mapE (mapI cvtE)
```

The functions `gfoldE`, `joinT`, `mapE` and `mapT` all require traversal over a term structure. Bird and Paterson's implementation of `cvtE` requires three traversal functions and `cvtBodyE` requires five traversal functions. While our implementation of `cvtE` only requires two traversal functions, i.e. `foldE` and `mapT`. Moreover, the function `cvtE` corresponds to `cvtE Z` in our implementation and `cvtBodyE` corresponds to `cvtE (S Z)`. So our implementation of `cvtE` is more flexible.

5 OBTAINING THE DEPENDENTLY TYPED FOLDS FOR ANY NESTED DATA TYPES

In this section, we hint at how the construction of dependently typed fold in Section 2 can be generalized to arbitrary nested data types. We do not provide a general formulation of the construction because of the technical overhead in formulating the most general form of nested data type declarations. One can find a formulation of nested data types as the fix points of the polynomial higher-order functors in [Bird and Paterson 1999b]. Instead, we give an example of nested data type that is hopefully general enough to make it clear what one would do in the general case. We leave the general formulation and the meta-theoretical study of dependently typed folds as future work.

In the previous sections, we use a natural number `n` as the index to obtain the dependently typed folds for `NBush n a`, `Term (NIncr n a)` and `TermE (IncrTermE n a)`. While using the natural number index has the benefit of being intuitive, it raises the question of whether it is always possible to obtain dependently typed folds for nested data types. Fortunately, in a dependently typed language, types can depend not only on the natural numbers but also on any inductive data types. In a general setting, given an arbitrarily complicated nested data type, we can use a customized regular data type as the index to define the dependently typed fold.

5.1 A method to obtain dependently typed folds

Consider the following nested data type `I` and `D`.

```
data I (a : Set) : Set where
  Zero : I a
  Succ : a -> I (I a) -> I a

data D (a b : Set) : Set where
  DNil : D a b
  DCons : a -> b -> D (I a) b -> D (D (I b) (I b)) (I a) -> D a b
  ACons : I b -> D (I (I (D b a))) (D (D b a) (D a b)) -> D a b
```

Here `D` is a type-constructor of arity 2 and `I` is a type-constructor of arity 1. The nested data type `D` is reasonably arbitrary and general. It refers to another nested data type `I`, and its constructors `DCons` and `ACons` are also nested. We will consider how to obtain a dependently typed fold for `D`.

We first define the following regular data type `IndexD` to describe all the types arising from `D`, i.e. the types constructed from `D`, `I` and the variables `a`, `b`.

```
data IndexD : Set where
  VarA : IndexD
  VarB : IndexD
  IsD : IndexD -> IndexD -> IndexD
  IsI : IndexD -> IndexD
```

The constructors `VarA` and `VarB` describe the two variables for `D`, the constructor `IsD` of arity 2 describes the type-constructor `D` and the constructor `IsI` of arity 1 describes `I`.

We then use structural recursion to define the following type-level function that translates a value of `IndexD` to its corresponding type.

```
H : IndexD -> Set -> Set -> Set
H VarA a b = a
H VarB a b = b
H (IsD x y) a b = D (H x a b) (H y a b)
H (IsI x) a b = I (H x a b)
```

For example, `H (IsD (IsI (IsD (IsI VarA) (IsI VarB))) (IsI VarA)) Nat Char` will be evaluated to the type `D (I (D (I Nat) (I Char))) (I Nat)`.

Similar to `NBush n a`, we view `H i a b` as a kind of abstract indexed data type. We now define the dependently typed fold for `H i a b` interactively in Agda. We begin with the following.

```
foldD : {a b : Set} {p : IndexD -> Set} -> (i : IndexD) -> H i a b -> p i
foldD {a} {b} {p} i l = ?
```

We will extend the type of `foldD` with arguments that correspond to the constructors of the abstract data type `H i a b`. By dependent pattern-matching on `i` and `H i a b`, we have the following seven cases.

```
foldD : {a b : Set} {p : IndexD -> Set} -> (i : IndexD) -> H i a b -> p i
foldD {a} {b} {p} VarA l = ?
foldD {a} {b} {p} VarB l = ?
foldD {a} {b} {p} (IsD i j) DNil = ?
foldD {a} {b} {p} (IsD i j) (DCons x y l v) = ?
foldD {a} {b} {p} (IsD i j) (ACons x l) = ?
foldD {a} {b} {p} (IsI i) Zero = ?
foldD {a} {b} {p} (IsI i) (Succ x y) = ?
```

The cases for `VarA`, `VarB` and `DNil` suggest that we extend `foldD` with the arguments `varA`, `varB` and `bnil`.

```
foldD : {a b : Set} {p : IndexD -> Set} ->
        (i : IndexD) ->
        (varA : a -> p VarA) ->
        (varB : b -> p VarB) ->
        (bnil : {i j : IndexD} -> p (IsD i j)) ->
         H i a b -> p i
foldD {a} {b} {p} VarA varA varB bnil l = varA l
foldD {a} {b} {p} VarB varA varB bnil l = varB l
foldD {a} {b} {p} (IsD i j) varA varB bnil DNil = bnil
foldD {a} {b} {p} (IsD i j) varA varB bnil (DCons x y l v) = ?
...
```

We will now focus on the case for `DCons x y l v`, as the other cases follow similarly. We want to make a recursive call of `foldD` on each of the components in `DCons x y l v`. The following is the environment provided by Agda.

```
Goal: p (IsD i j)
-------------------------------------------------------
v     : D (D (I (H j a b)) (I (H j a b))) (I (H i a b))
l     : D (I (H i a b)) (H j a b)
y     : H j a b
x     : H i a b
```

The key to make these recursive calls is to provide `foldD` with the correct indexes and these indexes can be structurally larger than `IsD i j`. For example, the index we provide to `foldD` when calling it on `v` is `IsD (IsD (IsI j) (IsI j)) (IsI i)` and the result of the recursive call is of the type `p (IsD (IsD (IsI j) (IsI j)) (IsI i))`. Thus we add the argument `bcons` for `foldD` to combine the results of the recursive calls on `x, y, l, v`.

```
foldD : {a b : Set} {p : IndexD -> Set} ->
        (i : IndexD) ->
        (varA : a -> p VarA) ->
        (varB : b -> p VarB) ->
        (bnil : {i j : IndexD} -> p (IsD i j)) ->
        (bcons : {i j : IndexD} -> p i -> p j -> p (IsD (IsI i) j) ->
                p (IsD (IsD (IsI j) (IsI j)) (IsI i)) -> p (IsD i j)) ->
         H i a b -> p i
foldD {a} {b} {p} VarA varA varB bnil bcons l = varA l
foldD {a} {b} {p} VarB varA varB bnil bcons l = varB l
foldD {a} {b} {p} (IsD i j) varA varB bnil bcons DNil = bnil
foldD {a} {b} {p} (IsD i j) varA varB bnil bcons (DCons x y l v) =
  bcons
   (foldD {a} {b} {p} i varA varB bnil bcons x)
   (foldD {a} {b} {p} j varA varB bnil bcons y)
   (foldD {a} {b} {p} (IsD (IsI i) j) varA varB bnil bcons l)
   (foldD {a} {b} {p} (IsD (IsD (IsI j) (IsI j)) (IsI i))
      varA varB bnil bcons v)
...
```

Note that the above recursive definition of `foldD` is well-founded, Agda is able to confirm its termination. The type of the final definition of `foldD` is the following, its full definition is in the supplementary material.

```
foldD : {a b : Set} {p : IndexD -> Set} ->
       (i : IndexD) ->
       (varA : a -> p VarA) ->
       (varB : b -> p VarB) ->
       (bnil : {i j : IndexD} -> p (IsD i j)) ->
       (bcons : {i j : IndexD} -> p i -> p j -> p (IsD (IsI i) j) ->
                 p (IsD (IsD (IsI j) (IsI j)) (IsI i)) -> p (IsD i j)) ->
       (acons : {i j : IndexD} -> p (IsI j) ->
                 p (IsD (IsI (IsI (IsD j i))) (IsD (IsD j i) (IsD i j))) ->
                 p (IsD i j)) ->
       (zero : {i : IndexD} -> p (IsI i)) ->
       (succ : {i : IndexD} -> p i -> p (IsI (IsI i)) -> p (IsI i)) ->
       H i a b -> p i
```

We can use the dependently typed fold `foldD` to define the following map and sum functions.

```
mapD : {a b c d : Set} -> (i : IndexD) -> (a -> c) -> (b -> d) -> H i a b -> H i c d
mapD {a} {b} {c} {d} i f g l =
  foldD {a} {b} {\ i -> H i c d} i f g DNil DCons ACons Zero Succ l

mapD' : {a b c d : Set} -> (a -> c) -> (b -> d) -> D a b -> D c d
mapD' = mapD (IsD VarA VarB)
```

```
sumD : D Nat Nat -> Nat
sumD x =  foldD {Nat} {Nat} {\ i -> Nat} (IsD VarA VarB) (\ y -> y) (\ y -> y)
  Z (\ x1 x2 x3 x4 -> add x1 (add x2 (add x3 x4))) add Z add x
```

The map function `mapD` traverses over the abstract data type `H i a b`, leaving all the constructors unchanged, while applying `f` and `g` at the leaves.

Similar to what we have described in the previous sections, we can obtain an induction principle from `foldD` by generalizing its type, i.e., generalizing the kind of `p` to `(i : IndexD) -> H i a b -> Set`. Moreover, since `H i a b` can be viewed as an indexed data type with seven constructors, we can obtain the indexed representation (Section 2.1) and the Church-encoding of `H` (Section 2.2). Finally, it should be clear that we can apply the method we just described to obtain dependently typed folds for any nested data types.

5.2 Specializing dependently typed folds to higher-order folds

Let us consider how to specialize dependently typed folds to the higher-order folds, using the data type `D` as example. The following is the type of the higher-order fold for `D`.

```
hfoldD : {a b : Set} {p : Set -> Set -> Set} ->
         (dnil : {a b : Set} -> p a b) ->
         (dcons : {a b : Set} -> a -> b -> p (I a) b ->
                    p (p (I b) (I b)) (I a) -> p a b) ->
         (acons : {a b : Set} -> I b ->
                    p (I (I (p b a))) (p (p b a) (p a b)) -> p a b) ->
          D a b -> p a b
```

To obtain this type, we first obtain `dnil`, `dcons` and `acons`, their types are the same as the types for the constructors `DNil`, `DCons` and `ACons`, except the type-constructor `D` is replaced by the type variable `p` of kind `Set -> Set -> Set`. We then use `bnil`, `bcons` and `acons` as the additional arguments for the function `D a b -> p a b`. Note that for any nested data type, we can obtain the type of its higher-order fold this way.

We now define the following type-level function `Hp` to replace the constructor `IsD` by the binary type variable `p : Set -> Set -> Set`, while keeping the other type-constructors unchanged.

```
Hp : IndexD -> (Set -> Set -> Set) -> Set -> Set -> Set
Hp VarA p a b = a
Hp VarB p a b = b
Hp (IsD i j) p a b = p (Hp i p a b) (Hp j p a b)
Hp (IsI i) p a b = I (Hp i p a b)
```

The higher-order fold `hfoldD` is defined by instantiating `foldD` with `\ i -> Hp i p a b`.

```
hfoldD {a} {b} {p} dnil dcons acons x =
  foldD {a} {b} {\ i -> Hp i p a b} (IsD VarA VarB)
    (\ y -> y) (\ y -> y) dnil dcons acons Zero Succ x
```

The higher-order fold `hfoldD` traverses over `D`, replacing the constructors `DNil`, `DCons` and `ACons` by `dnil`, `dcons` and `acons`, while leaving the constructors `Zero` and `Succ` unchanged.

5.3 Discussion

Looking back at the definition of `foldD`, it can be criticized for being too general. For example, although `foldD` is defined for folding `D`, we can also use `foldD` to define a summation for `I Nat`.

```
sumI : I Nat -> Nat
```

```
sumI l = foldD {Nat} {Nat} {\ y -> Nat} (IsI VarA) (\ y -> y) (\ y -> y)
           Z (\ x x1 x2 x3 -> Z) (\ x x1 -> Z) Z add l
```

In this definition of `sumI`, we need to supply additional arguments such as `\ x x1 x2 x3 -> Z` and `\ x x1 -> Z` even though we know these arguments will not be used when evaluating `sumI`.

One way to understand why `foldD` may also work for `I` is that since the definition of `D` involves a nested use of `I`, so a fold for `I` is needed when folding `D`. As a result, the function `foldD` may be used to operate on the values that only involves data type `I`. If one wants to program with the data type `I`, then it is more natural to program with the designated dependently typed fold for `I`.

We call the dependently typed folds obtained from the method in Section 5.1 *the direct dependently typed folds*. The example of obtaining `foldD` shows that the direct dependently typed folds always exist and we know how to systematically construct them.

In practice, we often have more intuition on how we intend to use the nested data types. For example, consider the `Term` and `TermE` data type in the previous sections, where we are particularly interested in types of the forms `Term` ($\texttt{Incr}^n a$) and `TermE` ($\texttt{IncrTermE}^n a$). So in these cases we use a natural number as the index to define folds for `Term` ($\texttt{Incr}^n a$) and `TermE` ($\texttt{IncrTermE}^n a$). Let us call the folds for `Term` ($\texttt{Incr}^n a$) and `TermE` ($\texttt{IncrTermE}^n a$) *customized dependently typed folds*.

Given a nested data type, its customized dependently typed folds can be different from its direct dependently typed fold. For example, for the data types `Term` and `TermE`, the customized dependently typed folds `foldT` and `foldE` are not the same as the direct dependently typed folds. On the other hand, for the data type `Bush`, the customized dependently typed fold `foldB` coincides with its direct dependently typed fold.

We say a dependently typed fold is *proper* if it can be specialized to the corresponding higher-order fold. Because the higher-order fold is considered the *defining property* of a nested data type, as the higher-order fold can be understood as the unique morphism from the initial nested data type object in the higher-order functor category [Bird and Meertens 1998, §4]. We show that the direct dependently typed folds are proper in Section 5.2. For the customized dependently typed folds, the properness has to be shown in a case by case basis (e.g. `hfoldT` and `hfoldE` for `Term` and `TermE`).

6 RELATED WORK

Generalized folds This paper is inspired by the works of Bird, Paterson and Meertens. The higher-order folds such as `hfoldB` in Section 1 were thought not expressive enough to define functions such as summation, which leads to the consideration of *generalized folds* ([Bird and Paterson 1999a], [Bird and Paterson 1999b]). The generalized folds further generalize the existing higher-order folds with extra higher-order type variables and arguments. For example, the following is a version of generalized fold for the `Bush` data type. We have to use the unsafe flag `--no-termination` to make Agda accept the following code.

```
gfoldB : {a : Set} -> {p q : Set -> Set} ->
        ({b : Set} -> p b) ->
        ({b : Set} -> q b -> p (p b) -> p b) ->
        ({b : Set} -> p b -> q (p b)) ->
        Bush (q a) -> p a
gfoldB base step k NilB = base
gfoldB {a} {p} {q} base step k (ConsB x xs) =
  step x (gfoldB {p a} {p} {q} base step k
           (hmapB (\ y -> k (gfoldB {a} {p} {q} base step k y)) xs))

hfoldB : {a : Set} -> {p : Set -> Set} ->
```

```
        ({b : Set} -> p b) -> ({b : Set} -> b -> p (p b) -> p b) -> Bush a -> p a
hfoldB {a} {p} base step = gfoldB {a} {p} {\ y -> y} base step (\ y -> y)

sumB : Bush Nat -> Nat
sumB = gfoldB {Nat} {\ y -> Nat} {\ y -> Nat} Z add (\ x -> x)
```

We can see the higher-order fold `hfoldB` is indeed an instance of the generalized fold `gfoldB`. Moreover, `gfoldB` can be used to define functions such as `sumB`.

Higher-order folds Johann and Ghani show that the higher-order folds such as `hfoldB` can be used to define functions such as `sumB` ([Johann and Ghani 2007], [Johann and Ghani 2009]). The intuitive idea is instead of defining the summation function directly, one uses `hfoldB` to define the following auxiliary function `sumAux` first, and define `sumB'` based on `sumAux`.

```
sumAux : {a : Set} -> Bush a -> (a -> Nat) -> Nat
sumAux {a} = hfoldB {a} {\ a -> (a -> Nat) -> Nat}
               (\ x -> Z) (\ x k f -> add (f x) (k (\ r -> r f)))

sumB' : Bush Nat -> Nat
sumB' l = sumAux l (\ y -> y)
```

When defining `sumAux`, we instantiate the type variable `p` in `hfoldB` with `\ a -> (a -> Nat) -> Nat`. Johann and Ghani generalize the pattern of `sumAux` and show how it is related to an advanced concept called Kan extensions from category theory [Johann and Ghani 2007].

Comparison of dependently typed folds, generalized folds and higher-order folds. We already mentioned that dependently typed folds can be specialized to higher-order folds, and that dependently typed folds does not requires map functions, and that map functions can be defined by dependently typed folds, and that dependently typed folds are defined using well-founded recursion, and that dependently typed folds correspond to the induction principles.

Mendler-style iterators Abel, Matthes and Uustalu propose to use a generalized version of Mendler-style iteration [Mendler 1987] to program with nested data types ([Abel et al. 2003], [Abel et al. 2005]). The intuitive idea is that one first defines nested data types as recursive types, i.e. the fix points of higher-order functors. For example, the `Bush` data type is encoded as the following `Bush'` with the constructors `BNil` and `BCons`.

```
data Mu (F : (Set -> Set) -> (Set -> Set)) (a : Set) : Set where
  In : F (Mu F) a -> Mu F a
BushF : (Set -> Set) -> (Set -> Set)
BushF B a = Unit + a * (B (B a))
Bush' : Set -> Set
Bush' = Mu BushF
BNil : {a : Set} -> Bush' a
BNil = In (Inl unit)
BCons : {a : Set} -> a -> Bush' (Bush' a) -> Bush' a
BCons x xs = In (Inr (Times x xs))
```

Note that the data type `Mu` is not strictly-positive in Agda, we have to use the unsafe flag `--no-positivity` to make Agda accept the above code.

One then defines the following generalized Mendler-style iterator `gIt`. The iterator `gIt` uses a generalized impredicative type abstraction `mon`.

```
mon : (Set -> Set) -> (Set -> Set) -> (Set -> Set) -> Set
mon F H G = {a b : Set} -> (a -> H b) -> F a -> G b

gIt : {F : (Set -> Set) -> (Set -> Set)} {H G : Set -> Set} ->
      ({X : Set -> Set} -> mon X H G -> mon (F X) H G) ->
```

```
      mon (Mu F) H G
gIt s f (In t) = s (gIt s) f t
```

We have to use the unsafe flag `--type-in-type` to make Agda accept the definition of `mon`. Moreover, another unsafe flag `--no-termination` is needed because the termination of `gIt` is not obvious for Agda.

The `sumAux'` function and the map function `mapBush` can be defined from `gIt`.

```
sumAux' : {a : Set} -> (a -> Nat) -> Bush' a -> Nat
sumAux' {a} = gIt {BushF} {\ x -> Nat} {\ x -> Nat}
              (\ {X} r {a} {b} f t ->
                  match t (\ x -> Z)
                          (\ p -> add (f (p1 p))
                                  (r {X a} {b} (r {a} {b} f) (p2 p)))) {a} {a}
mapBush : {X Y : Set} -> (X -> Y) -> (Bush' X -> Bush' Y)
mapBush = mfold bushF In
```

Although in Agda, the `Mu` data type is not strictly positive, the definition of `mon` requires impredicative polymorphism and the definition of `gIt` is not obviously terminating, Abel et. al. [Abel et al. 2003] show that the recursive types together with the generalized Mendler-style iterators can be encoded in Girard's $\mathbf{F}_\omega$ [Girard 1972] using a syntactic version of Kan extensions, hence the programs defined by the generalized Mendler-style iterators are still terminating.

Induction principles for Mendler-style iterators Matthes proposes to use a system called *LNMIt* (logic for natural Mendler-style iteration of rank 2) [Matthes 2009, Fig. 1.] to reason about programs defined by Mendler-style iterators. The *LNMIt* consists of the usual constants such as `In` and `gIt`, it also contains the special constants for map and induction. Although the map constant and the iterator `gIt` come with reduction rules, there is no such rule for the induction constant. The type of the induction constant contains the map constant, and the induction constant provides a mean to show a property holds for the data types defined from `Mu`. It is unclear to us how the induction constant in *LNMIt* is related to the iterator `gIt`. Matthes [Matthes 2009, §5] proves in Coq that *LNMIt* can be defined within the Calculus of Inductive Construction [Coquand and Paulin 1990] with the additional axioms of impredicative `Set` and proof irrelevance.

Comparison of dependently typed folds, generalized Mendler-style iterators and LNMIt. Both dependently typed folds and generalized Mendler-style iterators enable total programming with iterators, and allow maps functions to be defined from the iterators. The approach of generalized Mendler-style iterators requires working with recursive types instead of the usual inductive definitions, it has the advantage of not imposing any positivity constraint for the data types. The dependently typed folds approach requires working with the inductive definitions of data types, it is limited to a subclass of strictly positive data types. As for the verification of programs involving nested data types, the justification of System *LNMIt* requires impredicative `Set` and proof irrelevance, while the induction principles obtained from dependently typed folds work directly in total Agda and does not requires the axioms of impredicative `Set` and proof irrelevance.

Other related work from type theory The main technique we use to realize dependently typed folds is called *large elimination* in the dependent types literature ([Altenkirch 1993], [Werner 1992]), i.e. computing types by pattern-matching on values. Werner shows that the inductive reasoning in the Calculus of Inductive Construction is consistent [Werner 1994]. Modern dependently typed languages such as Agda, Coq allow user-defined regular/nested data types and well-founded recursive function definitions [Coquand 1992], this enables us to define the dependently typed folds, their corresponding induction principles and various type-level functions.

7 CONCLUSION AND FUTURE WORK

We show how to define dependently typed folds for nested data types and how to specialize them to the corresponding higher-order folds. Dependently typed folds can be used to define maps, and other terminating functions. They give rise to the induction principles, similar to the folds for regular data types. We show how to use induction principles to reason about the programs involving nested data types. We also discuss how dependently typed folds give rise to the indexed representations and how to obtain the Church encodings of the indexed representations.

For future work, we would like to formalize a general procedure to obtain the direct dependently typed folds. We would also like to consider the meta-theoretic properties of dependently typed folds. For example, it would be nice to be able to prove a meta-theorem stating that any higher-order fold obtained from the direct dependently typed fold will have the same computational behavior as the higher-order fold obtained from general recursion.

REFERENCES

Andreas Abel, Ralph Matthes, and Tarmo Uustalu. 2003. Generalized iteration and coiteration for higher-order nested datatypes. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 54–69.

Andreas Abel, Ralph Matthes, and Tarmo Uustalu. 2005. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science* 333, 1-2 (2005), 3–66.

Thorsten Altenkirch. 1993. *Constructions, Inductive Types and Strong Normalization*. Ph.D. Dissertation. University of Edinburgh.

Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *Mathematics of program construction*. Springer, 52–67.

Richard Bird and Ross Paterson. 1999a. De Bruijn notation as a nested datatype. *Journal of functional programming* 9, 1 (1999), 77–91.

Richard Bird and Ross Paterson. 1999b. Generalised folds for nested datatypes. *Formal Aspects of Computing* 11, 2 (1999), 200–222.

Thierry Coquand. 1989. *Metamathematical investigations of a calculus of constructions*. Technical Report RR-1088. INRIA. https://hal.inria.fr/inria-00075471

Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. 66–79.

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and computation* 76, 2-3 (1988), 95–120.

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*. Springer, 50–66.

Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.

Herman Geuvers. 2001. Induction is not derivable in second order dependent type theory. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 166–181.

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris VII.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and types*. Vol. 7. Cambridge University Press Cambridge.

Patricia Johann and Neil Ghani. 2007. Initial algebra semantics is enough!. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 207–222.

Patricia Johann and Neil Ghani. 2009. A principled approach to programming with nested types in Haskell. *Higher-Order and Symbolic Computation* 22, 2 (2009), 155–189.

Ralph Matthes. 2009. An induction principle for nested datatypes in intensional type theory. *Journal of Functional Programming* 19, 3-4 (2009), 439–468.

Paul Francis Mendler. 1987. *Inductive definition in type theory*. Ph.D. Dissertation.

Benjamin Werner. 1992. A normalization proof for an impredicative type system with large elimination over integers. In *International Workshop on Types for Proofs and Programs (TYPES)*. 341–357.

Benjamin Werner. 1994. *Une Théorie des Constructions Inductives*. Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00196524