Towards an induction principle for nested data types

Frank Fu

Joint work with P. Selinger

WOLLIC 2023

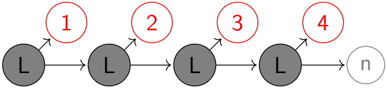
Inductive data type: Nat

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat

foldNat : ∀{a : Set} → a → (a → a) → Nat → a
foldNat z s zero = z
foldNat z s (succ n) = s (foldNat z s n)

indNat : ∀{p : Nat → Set} →
           p zero →
           (∀(x : Nat) → p x → p (succ x)) →
           ∀(n : Nat) → p n
indNat base step zero = base
indNat base step (succ n) = step n (indNat base step n)
```

Inductive data type: List




```
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)

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

indList : ∀{a : Set}{p : List a → Set} →
            p nil →
            (∀(x : a)(xs : List a) → p xs → p (cons x xs)) →
            ∀(ℓ : List a) → p ℓ
```

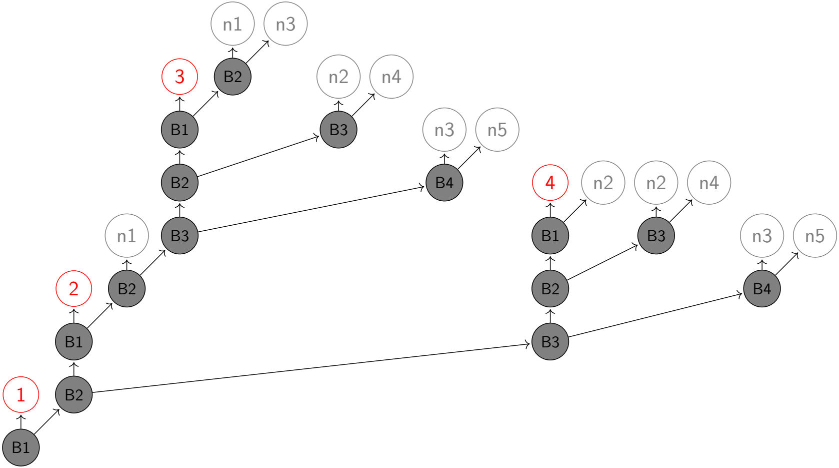
Nested data type: Bush

```
data Bush (a : Set) : Set where
  leaf : Bush a
  cons : a → Bush (Bush a) → Bush a
```

Nested data type: Bush

$$Bn := \underbrace{\text{Bush}\,(\ldots(\text{Bush}}_{n}\ \text{Nat})\ldots)$$

```
data Bush (a : Set) : Set where
  leaf : Bush a
  cons : a → Bush (Bush a) → Bush a
```



Higher-order fold for Bush

```
{-# TERMINATING #-}
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)

{-# TERMINATING #-}
hfold : ∀(b : Set → Set) →
        (∀(a : Set) → b a) →
        (∀(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))
```

Problems of higher-order fold

- Not defined by structure recursion.
- `hfold` depends on `hmap`.
- No corresponding induction principle.
- Hard to use [Bird and Meertens 1998].

Dependently typed fold

NTimes n Bush a $=$ Bush (Bush $\ldots$ (Bush a)).

```
nfold : ∀(p : Nat → Set) →
        (∀(n : Nat) → p (succ n)) →
        (∀(n : Nat) → p n → p (succ (succ n)) → p (succ n)) →
        ∀(a : Set) → (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)

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
```

Induction principle

```
nfold : ∀(p : Nat → Set) →
        (∀(n : Nat) → p (succ n)) →
        (∀(n : Nat) → p n → p (succ (succ n)) → p (succ n)) →
        ∀(a : Set) → (a → p zero) →
          ∀(n : Nat) → NTimes n Bush a → p n

ind : ∀(p : ∀(n : Nat) → NTimes n Bush a → Set) →
        (∀(n : Nat) → p (succ n) leaf) →
        (∀(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)) →
        ∀(a : Set) → (∀(x : a) → p zero x) →
          ∀(n : Nat) → ∀(xs : NTimes n Bush a) → p n xs
```

Summary of the paper

- We showed `nfold` can be used to define map and many other functions.
- We showed how to reason about `Bush` using induction principle.
- We showed `nfold` and `hfold` are *mutually* definable in Agda.
- We gave an example to illustrate our approach also works for arbitrary nested data types (new addition!).

A quick glance

```
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
```

Thank you!

Dependently typed fold for Bob Dylan

```
data BobDylanIndex : Set where
  varA : BobDylanIndex
  varB : BobDylanIndex
  BobC : BobDylanIndex → BobDylanIndex
  DylanC : BobDylanIndex → BobDylanIndex → BobDylanIndex

I : (Set → Set) → (Set → Set → Set) → Set → Set →
    BobDylanIndex → Set

I Bob Dylan a b (DylanC (BobC varA) (BobC varB)) == Dylan (Bob a) (Bob b)
```

Dependently typed fold for Bob Dylan

```
nfold : ∀(p : BobDylanIndex → Set) →
        (∀ a → p a → p (BobC a)) →
        (∀ a → p (DylanC (BobC (DylanC a (BobC a))) (BobC a)) →
               p (BobC (DylanC a a)) → p (BobC a)) →
        (∀ a b → p (BobC a) → p (BobC b) → p (DylanC a b)) →
        (∀ a b → p (DylanC (BobC a) (BobC b)) → p (DylanC a b)) →
        ∀(a b : Set) → (a → p varA) → (b → p varB) →
          (∀ i → I Bob Dylan a b i → p i)
```