LibNDT: Towards a formal library on spreadable properties over linked nested datatypes

Mathieu Montin[1], Amélie Ledein, Catherine Dubois

1 Loria, Nancy, France
2 INRIA, Paris-Saclay, France
3 ENSIIE, Samovar, Evry, France

Introduction

Motivation: study a family of nested datatypes from a **practical** point of view

- initially, induction principles
- then, functions and properties than can be automatically obtained

Contribution: a core library, LIBNDT, written both in AGDA and COQ, about *spreadable* properties of *linked nested datatypes* (LNDTs) available on `https://github.com/mmontin/libndt`

Regular vs Nested Parameterized Datatypes

Regular datatype: the type parameter is always the same in the type definition (generic)

```
data List a = Empty | Cons(a, List a)
data Tree a = Tip a | Bin(Tree a, Tree a)
```

Nested datatype: the type parameter changes between the type signature and at least, one of its instances in the datatype constructors

```
data Nest a = Null | Cons (a, Nest (a, a))
data Pow a = Zero a | Succ (Pow (a, a))
data Bush a = BLeaf  | BNode (a, Bush (Bush a))
```

Regular vs Nested Parameterized Datatypes

Regular datatype: the type parameter is always the same in the type definition (generic)

```
data List a = Empty | Cons(a, List a)
data Tree a = Tip a | Bin(Tree a, Tree a)
```

Nested datatype: the type parameter changes between the type signature and at least, one of its instances in the datatype constructors

```
data Nest a = Null | Cons (a, Nest (a, a))
data Pow a = Zero a | Succ (Pow (a, a))
data Bush a = BLeaf  | BNode (a, Bush (Bush a))
```

$\longrightarrow$ Generalize this kind of definition by abstracting the type transformation (TT) performed on the type parameter

Regular vs Nested Parameterized Datatypes

Regular datatype: the type parameter is always the same in the type definition (generic)

```
data List a = Empty | Cons(a, List a)
data Tree a = Tip a | Bin(Tree a, Tree a)
```

Nested datatype: the type parameter changes between the type signature and at least, one of its instances in the datatype constructors

```
data Nest a = Null | Cons (a, Nest (a, a))
data Pow a = Zero a | Succ (Pow (a, a))
data Bush a = BLeaf  | BNode (a, Bush (Bush a))
```

$\longrightarrow$ Generalize this kind of definition by abstracting the type transformation (TT) performed on the type parameter
$\longrightarrow$ Focus on *list like* datatypes

Regular vs Nested Parameterized Datatypes

Regular datatype: the type parameter is always the same in the type definition (generic)

```
data List a = Empty | Cons(a, List a)
data Tree a = Tip a | Bin(Tree a, Tree a)
```

Nested datatype: the type parameter changes between the type signature and at least, one of its instances in the datatype constructors

```
data Nest a = Null | Cons (a, Nest (a, a))
data Pow a = Zero a | Succ (Pow (a, a))
data Bush a = BLeaf  | BNode (a, Bush (Bush a))
```

⟶ Generalize this kind of definition by abstracting the type transformation (TT) performed on the type parameter
⟶ Focus on *list like* datatypes
⟶ Linked Nested DataTypes

Introducing LNDTs

Definition of LNDT

```
Inductive LNDT (F : TT) (A : Type) : Type :=
  | empty : LNDT F A
  | node : A → LNDT F (F A) → LNDT F A.
```

with

```
Definition TT := Type → Type.
```

Lists, Nests, N-perfect trees as LNDTs

Lists as LNDTs

```
Definition List := LNDT Id.
```

Nests as LNDTs

```
Definition Nest := LNDT (fun A ⇒ A * A).
```

Perfect trees as LNDTs

```
Definition N_PT n := LNDT (Tuple n).
```

with Tuple n A defined as A^{n+1}

Lists, Nests as perfect trees

```
Definition List : Type → Type := N_PT 0.

Definition Nest : Type → Type := N_PT 1.
```

A nest example

```
Definition example : Nest nat :=
node _ _ 7
(node _ _ (1, 2)
  (node _ _ ((6, 7), (7, 4))
   (node _ _ (((2, 5), (7, 1)), ((3, 8), (9, 3)))
    (empty _ _ )))).
```

LNDTs as TT - Multi-layered LNDTs

List, Nest, PerfectTree are also type transformers
More generally, if F : TT, then LNDT F : TT

$\longrightarrow$ Multi-layered LNDTs

```
Definition SquaredList : Type → Type := LNDT List.


Definition exampleSqList : SquaredList nat:=
node _ _ 1
  (node _ _ (node _ _ 1 (empty _ _))
    (node _ _ (node _ _ (node _ _ 1 (empty _ _)) (empty _ _))
     (empty _ _))).
(* or in a more friendly notation  [ 1 ; [1] ; [[1]] ]  *)
```

What about Bush as a LNDT?

```
data Bush a = BLeaf  | BNode (a, Bush (Bush a))
```

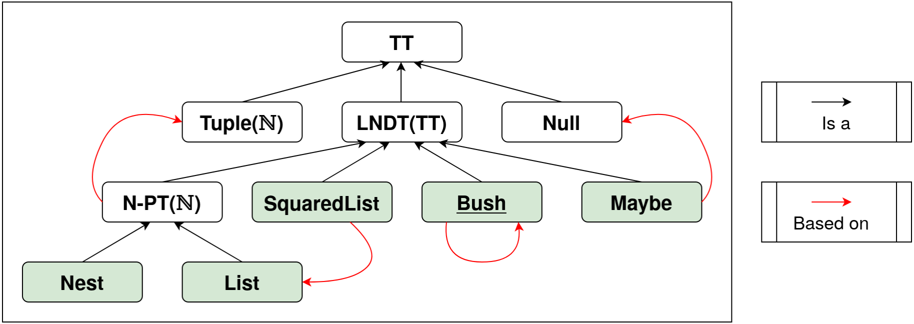
- Cannot be defined in Coq as an instance of LNDT (nor directly)
- Can be defined in Agda if termination checking is turned off

  Bush : TT
  Bush = LNDT Bush
  bush-example : Bush ℕ
  bush-example = 5 :: (16 :: []) :: ((87 :: []) :: ((56 :: []) :: []) :: []) :: []

- A safe definition accepted both in Agda and Coq

```
Inductive BushN (A : Type) : nat → Type :=
 | Base: A → BushN A 0
 | NilBN (n: nat): BushN A (S n)
 | ConsBN (n: nat):
     BushN A n → BushN A (S (S n)) → BushN A (S n).
```

Overview of LNDTs in LibNDT



Spreadable Functions

Issues addressed here:

1. What kind of functions can be expressed for LNDTs seen as collections?
   map, fold, size . . .
2. Can they be obtained, *at a low cost*, for LNDT F if they are available for F?
   If so, we call them spreadable functions.

Map for LNDTs

Apply a function $f : A \to B$ on each element of a value of type LNDT F A and obtain a value of type LNDT F B

- Specification of type transformers "mappable"

```
Definition Map (F : TT) : Type :=
  ∀ A B  : Type, (A → B) → (F A → F B).
```

- Map is spreadable!

```
Fixpoint lndt_map {F : TT} (map : Map F)
 (A B : Type) (f : A → B) (t : LNDT F A) : LNDT F B :=
 match t with
 |empty _ _    ⇒ empty _ _
 |node _ _ x e ⇒ node F B (f x)
                          (lndt_map map (F A) (F B) (map A B f) e)
 end.
```

lndt_map F : Map F →Map (LNDT F)

In order to derive a map function for LNDTs*, all is needed is to define a map function for the type transformer on which they are based.*

⟶ *In order to derive a map function for n perfect trees, all is needed is to define a map function for Tuple n*

```
Fixpoint tuple_map (n : nat) : Map (Tuple n) :=
 match n with
  | 0   ⇒ fun A B f t ⇒ f t
  | S p ⇒ fun A B f t ⇒
          (f (fst t), tuple_map p _ _ f (snd t))
 end.

Definition N_PT_map n := Indt_map (tuple_map n).
Definition list_map := N_PT_map 0.
Definition nest_map := N_PT_map 1.

Eval compute in
  nest_map _ _ (fun x ⇒ x + 3) example.
(* node _ _ 10
   (node _ _ (4, 5)
    (node _ _ ((9, 10), (10, 7))
     (node _ _ (((5, 8), (10, 4)), ((6, 11), (12, 6)))
      (empty _ _ )))). *)
```

Fold for LNDTs

Walk through a value of type LNDT F A while combining its elements using a given operator f and a seed b

- Specification of type transformers "foldable"

```
Definition Fold (F : TT) : Type :=
  ∀ (A B : Type), (B → A → B) → B → F A → B.
```

- Fold is spreadable (in two ways)!

```
Fixpoint lndt_foldr (F : TT) (foldr : Fold F)
 A B (f : B → A → B) (b : B) (t : LNDT F A) :=
 match t with
 |empty _ _ ⇒ b
 |node _ _ x v  ⇒ f (lndt_foldr _ foldr _ _ (foldr _ _ f) b v) x
 end.
```

lndt_foldr F : Fold F →Fold (LNDT F)

Fold for LNDTs

- Examples

```
Fixpoint tuple_foldr (n : nat) : Fold (Tuple n) :=
 match n with
  | 0   ⇒ fun A B f b t ⇒ f b t
  | S p ⇒ fun A B f b t ⇒ f (tuple_foldr p _ _ f b (snd t)) (fst t)
 end.


Definition N_PT_foldr n := lndt_foldr _ (tuple_foldr n).
Definition list_foldr := N_PT_foldr 0.
Definition nest_foldr := N_PT_foldr 1.

Definition nest_foldr := N_PT_foldr 1.

Eval compute in nest_foldr _ _ (fun x y ⇒ x + y) 0 example.
(*  = 72 : nat   *)
```

Back to Bush in Agda

Nothing to do!

```
bush-map : Map Bush
bush-map = Indt-map bush-map
```

```
bush-foldl : Fold Bush
bush-foldr = Indt-foldr bush-foldr
```

Logical Spreadable Elements

Issues addressed here:

1. What kind of logical elements can be expressed for LNDTs?
   3 categories: primitive predicates, decidability properties, properties on spreadable functions
2. Can they be obtained freely for LNDT F if they are available for F?
   If it is the case, we call them logical spreadable elements.

All for LNDTs

Check if a predicate defined on the type A is satisfied by all the elements of a value of type LNDT F A

- Specification of a predicate transformer (from A to $F\ A$)

```
Definition TransPred (F : TT) : Type :=
  ∀ (A : Type), (A → Prop) → ((F A) → Prop).
```

- All is spreadable (according to a predicate transformer)

```
Fixpoint lndt_all {F: TT} (T: TransPred F) A (P: A → Prop)
                  (t: LNDT F A) :=
match t with
 |empty _ _ ⇒ True
 |node _ _ x e  ⇒ (P x) ∧ (lndt_all T _ (T _ P) e)
end.
```

lndt_all F : (TransPred F) →(TransPred (LNDT F))

All for LNDTs

- Examples

```
Fixpoint tuple_all (n : nat) : TransPred (Tuple n) :=
 match n with
  | 0   ⇒ fun A P t ⇒ P t
  | S p ⇒ fun A P t ⇒ (P (fst t)) ∧ (tuple_all _ _ P (snd t))
   end.
```

```
Definition N_PT_all n := Indt_all (tuple_all n).
```

N_PT_all : ∀ (n : nat) (A : Type), (A →Prop) →(LNDT ((Tuple n) A) →Prop)

```
Definition nest_all :=  N_PT_all 1.
```

```
Lemma exampleAll : nest_all _ (fun x ⇒ x >= 1) example.
```

Decidability Transformers

If a predicate transformer preserves decidability then this property is propagated to LNDTs

- Specification of decidability properties transformers

```
Definition TransDec {F : TT} (TransP : TransPred F) : Type :=
  ∀ (A : Type) (P : A → Prop), Dec A P → Dec (F A) (TransP _ P).
```

- Proof of the decidability *preservation* of lndt_all

```
Lemma lndt_dec_all : ∀ {F : TT} (T : TransPred F),
    TransDec T → TransDec (lndt_all T).
```

by induction on the LNDT structure

- Applications on perfect trees, nest and bushes

Properties about spreadable functions and predicates

Same mechanism to transfer properties about spreadable functions and predicates, e.g. composition map and map congruence

- reminder of the specification Map

```
Definition Map (F : TT) : Type :=
  ∀ A B  : Type, (A → B) → (F A → F B).
```

- Specification of the map congruence property

```
Definition MapCongruence {F : TT} (map : Map F) : Type :=
  ∀ (A B : Type) (f : A → B) (g : A → B) (x : F A),
    (∀ x, f x = g x) → map A B f x = map A B g x.
```

- Proof of the *preservation* of LNDT map congruence

```
Lemma lndt_cng_map : ∀ {F : TT} {map : Map F}
  (cgMap : MapCongruence map), MapCongruence (lndt_map map).
```

by functional induction on lndt_map

Overview of the library LIBNDT

SpreadAble (F : TT)

FoldAble (F : TT)	
C	***foldl*** : Fold F ***foldr*** : Fold F
A	***size*** : F A → ℕ ***flatten*** : F A → List A ***show*** : (A → String) → (F A → String)

MapAble (F : TT)	
C	***map*** : Map F
L	***map-cong*** : MapCongruence map ***map-comp*** : MapComposition map

AnyAllAble (F : TT)	
L	***any*** : TransPred F ***all*** : TransPred F ***dec-any*** : TransDec any ***dec-all*** : TransDec all
A	_∈_ : REL A (F A) a ***dec-∈*** : Decidable {A = A} _≡_ → Decidable _∈_ ***empty*** : Pred (F A) a

EqAble (F : TT)	
L	***dec-eq*** : DecEq F

C	L	A
Computational functions	Logical functions	Additional functions

Future work

Extending LibNDT with additional spreadable elements

- More specifications for the previous spreadable functions and predicates (see Set/Collection interface)
- More spreadable functions and predicates, e.g. random generators (using Coq QuickChick), induction principles

Future work

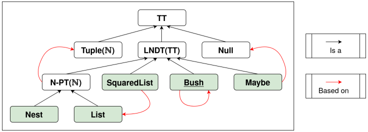
Extending LibNDT with additional datatypes

- Limitation of the work to a specific family of nested datatypes, with an *head-tail* view
- What about these ones?

  ```
  data Pow a = Zero a | Succ (Pow (a, a))
  data FingerTree a = Empty |
                      Single a |
                      Deep (D a) (FingerTree (N a)) (D a)
  ```

  They do not enter the family :-(
- What solutions to extend without re-doing all the work?
  - nested datatypes as an abstract notion instead of a concrete family of inductives types
    $\longrightarrow$ far from concrete preoccupations and often difficult to use, in particular in the context of proofs
  - meta-programming (e.g. Coq à la carte, MetaCoq)
  - ornaments

Agda



SpreadAble (F : TT)

FoldAble (F : TT)

C	***foldl*** : Fold F ***foldr*** : Fold F
A	***size*** : F A → ℕ ***flatten*** : F A → List A ***show*** : (A → String) → (F A → String)

MapAble (F : TT)

C	***map*** : Map F
L	***map-cong*** : MapCongruence map ***map-comp*** : MapComposition map

AnyAllAble (F : TT)

L	***any*** : TransPred F ***all*** : TransPred F ***dec-any*** : TransDec any ***dec-all*** : TransDec all
A	_∈_ : REL A (F A) a ***dec-∈*** : Decidable {A = A} _≡_ → Decidable _∈_ ***empty*** : Pred (F A) a

EqAble (F : TT)

L	***dec-eq*** : DecEq F

C	**L**	**A**
Computational functions	Logical functions	Additional functions

Thank you for your attention! Questions?

Backup

Express lndt_map from lndt_foldr

On lists:

```
map f = foldr (fun a l => f a :: l) []
```

This does not transpose into any LNDT: `l` is of type `LNDT (F B)` although `f a :: _` expects an element of type `LNDT (F (F B))`, where `f` is of type `A → B`.

Since `F` is the identity for `List`, it works, but it is a special case and can only be generalized for any `F` such as `F ∘ F = F`.