LibNDT: A formal library on spreadable properties over linked nested datatypes.

Mathieu Montin
Université de Lorraine, Loria, CNRS, Inria, France

Amélie Ledein
Université Paris Saclay, ENS Paris Saclay, LMF, CNRS, Inria, France

Catherine Dubois
ENSIIE, IP Paris, Samovar, France

Abstract

Nested datatypes have been widely studied in the past 50 years, both theoretically using category theory, and practically in programming languages such as Haskell. They consist in recursive polymorphic data-types where the type parameter changes throughout the recursion, and have a variety of applications, from memory modelling MM : to substitute to dependent types. In this work, we focus on a specific subset of nested data-types which we call linked nested data-types (LNDT). We explore the relevance of such types and show that well known nested data types such as Nest and even Bush as well as some usual inductive data-types such as List and Maybe can be built as special cases of LNDT, when a unique type parameter changes. We then present LibNDT, a library, developed both in Agda and Coq, which makes use of dependent types to model both LNDT as well as a set of constructs that can be propagated from the type parameter in question to the LNDT built from it. Such constructs include, but are not limited to, functions such as folds and map and properties such as membership and decidability of equality.

2012 ACM Subject Classification Theory of computation → Functional constructs; Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases Nested datatype, Bush, formal development, Agda, Coq

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material Source code avaible at `https://bitbucket.org/MonsieurO/nested/`

Funding Digicosme project IndepTh, ANR project FORMEDICIS

1 Introduction

Datatypes are widely used in programming in order to avoid some kind of bugs and to guarantee some properties, even before having executed anything. MM : voir exemple tas optimisés Among all the datatypes, the recursive ones are particularly used and studied since they are intrinsically related to induction. Usually, there is a distinction between regular datatypes and non-regular datatypes MM : cite yann bayley et bird 98.
A datatype is a regular one when all the occurrences of the declared type take the same parameters as the defining use. Famous examples of regular datatypes are `List a` and `Tree a`, since `data Tree a = Tip a | Bin (Pair (Tree a))` where `type Pair a = (a,a)`. As a point of view of induction, in this case, the recursion is "tail recursive", or Mycroft [17] calls such schemes polymorphic recursions. These datatypes are widely used and mostly sufficient when programming with a functional programming language.
A datatype is a nested one when the declared type can occur with different instances of the type parameters. For example, `Nest a`/`Pow a` and `Bush a` are nested datatypes as you can see: `data Pow a = Zero a | Succ (Pow (Pair a))`. As a point of view of induction, the recursion is here "nested". For this reason, non-regular datatypes are also called nested datatypes, a unique terminology we will use in this article.

Nested datatypes [2] have been widely studied in the past 50 years, both theoretically using category theory, and practically in programming languages such as Haskell.
Many works deal with nested datatypes as a whole, and are particularly interested in the folds that can be written on these structures. Indeed, these folds have to be as general and generic as possible, allowing to have powerful iterators, as well as induction principles. Ian Bayley's thesis [1] is also interested in the genericity of other functions such as zipMM : voir si l'on peut avoir le zip gratos ainsi que show or membership. The work presented in our article has a different scope: we do not consider all nested datatypes, but only a part of them, named **linked nested datatypes** hereafter. Moreover, we are interested in functions and properties that we can obtain for free from their simple definition, in the same spirit MM : voir deriving en haskell, le citer et s'y comparer as Wadley's article [4]. As we will see more precisely in the section 2.2, the linked nested datatypes can be instantiated in different ways to obtain different datatypes like lists MM : ajouter une petite def informelle des lndts. This methodology reminds us of Finger Tree [3], a general nested datatype, parameterized by a monoid. Instantiating this monoid allows to obtain more classical data structures like ordered sequences or interval trees.
MM : paragraphe bizarre, à remonter From a practical point of view, data structures are a central point in computer developments. Programmers must systematically ask themselves which data structures are the most relevant or still efficient for the algorithm we want to develop. Lists, arrays, queues, trees, stacks and so on are commonly used data structures, that can impact the efficiency of the algorithm. The choice is determined in particular by the systematic operations that programmers want to apply to the data structures, as well as by the relevance of whether or not this operation is efficient. There is always a balance to be found between having efficient operations, the complexity of implementation, and the frequency of use. For all these reasons, we have also developed an implementation in Agda and Coq of our library, available here: `https://bitbucket.org/MonsieurO/nested/`. This library allows to manipulate more easily the nested datatypes in order to use them in programs written with Agda or with Coq. An example of development will be presented in the section **??**. For now, you can note that the Coq positivity criterionMM : en dire un peu plus sur les différentes euristiques does not allow to define the Bush datatype, contrary

to Agda, which will still consider our development as unsafe. Historically, developments on this subject were done in Haskell, but we also wanted to prove some properties on our work, as we will see later in this article.MM : ref toulouse coq irit allemand

In the following, the pieces of code come from the Agda implementation of our library and have been extracted using `lagda`: this process ensures a synchronization between the code and the article, as well as a verification by Agda itself.
AL : Il faudra adapter le plan par la suite
This article proceeds as follows: after presenting and justifying the definition of the linked nested datatypes , we present several datatypes which can be defined in an instance of our linked nested datatypes .

We will then present the computational functions that it is possible to define and preserve. Moreover, we will extend this list to logical functions. Finally, we will focus on the library that we have developed in both Agda and Coq, emphasizing their differences.

Since, by nature, the work presented in this article only applies to some nested datatypes, we will also present some examples of nested datatypes that cannot be instanced with our linked nested datatypes .

AL : Est-ce possible de mettre quelque part que le principe d'induction proposé dans [**?**] est inutilisable en pratique ?

MM : Il faudrait dire qqes mots de lagda et de la disposition du document en fin d'intro

2 Introducing Linked Nested Data Types

2.1 Usual `List`, `Nest` and `Bush`

The most common inductive data-type that is linked are the lists, which consists in an arbitrary number of elements linked with one another. Written inductively using Agda lists can be defined, unsurprisingly, using two constructors: the empty list and the cons operator written in an infix manner (note that the index 0 means it is not yet defined as a LNDT).

```
1  data List₀ {a} (A : Set a) : Set a where
2    [] : List₀ A
3    _::_ : A → List₀ A → List₀ A
```

Lists are parameterized with a given type `A` from a given level of universe `a`. An interesting feature of such a type – that is usually not mentioned, although relevant in our case – is that the recursive constructor takes as parameter an element of type `List A` where `List` is parameterized by the same type parameter as its definition, `A`. This makes it a usual data type rather than a nested one, where such a type parameter supposedly varies throughout the recursion. As a first example of such a nested type, let us consider the usual `Nest` data type – sometimes called `Pow` in the literature:

```
4  data Nest₀ {a} (A : Set a) : Set a where
5    [] : Nest₀ A
6    _::_ : A → Nest₀ (A × A) → Nest₀ A
```

In this case, the recursive constructor – purposely named identically as the one of lists – takes as parameter an element of type `Nest (A × A)` where `A × A` is a pair of elements of type `A`. This makes `Nest` a nested data type, where its type parameter evolves throughout the recursion. As visible, both `List` and `Nest` are very similar in their form, and their only difference is the type parameter on which the newly defined type is recursively called. As a last example, let us consider the well-known `Bush` nested type, nested with itself:

```
7  data Bush₀ {a} (A : Set a) : Set a where
8    [] : Bush₀ A
9    _::_ : A → Bush₀ (Bush₀ A) → Bush₀ A
```

In this case, not only does the type parameter changes in the recursive call, but it changes with a dependence to the type that is being defined. While picturing lists and nests is fairly simple, picturing a bush is challenging. Thankfully, while the parameter change depends on `Bush` itself, the form of the type is fairly similar to lists and nests, which calls out for a common denominator between the three – and possibly more – types, thus allowing us a better picturing and understanding of bushes in the process.

2.2 Linked Nested Data Types

The common denominator between the three aforementioned types is notion we call Linked Nested Data Types (LNDTs). LNDTs are parameterized by a type transformer, that is an entity which, given a level of universe `a` an a type living in `Set a` provides another type living in `Set a`. The type of type transformers is shortened `TT`, that is `TT = ∀ b → Set b → Set b`, and we will also shorten "type transformer" to TT in the remaining of this paper.

```
10  data LNDT (F : TT) {a} (A : Set a) : Set a where
11    [] : LNDT F A
12    _::_ : A → LNDT F (F A) → LNDT F A
```

It is interesting to note that this data type applied to a certain TT is itself a TT, that is, for any `F : TT`, we have `LNDT F : TT`. This means that this library, although meant to define spreadable properties over LNDTs, is also fully about a certain number of type transformers, as shown later on in Figure 1.

2.3 `List`, `Nest` and `Bush` as instances of LNDT

Tuples – Naturally, the three types considered as examples, `List`, `Nest` and `Bush` can be seen as instances of LNDT, should we provide the right TT for each of them. In that purpose, we can notice that the TT required for `List` and `Nest` can be abstracted in a notion that we indeed call `Tuple`:

```
13  Tuple : ℕ → TT
14  Tuple zero = id - id is the identity function
15  Tuple (suc n) A = A × (Tuple n A)
```

A tuple indexed by `n` and parametrized by a type `A` is a collection of `n + 1` elements of type `A`. Note that the reader may wonder why the usual vectors, expressed an a family of types indexed by natural number, were not used instead of tuples. The reason is twofold: first, vectors can be empty, while tuples cannot which is required, and second, using the pair notation instead of the cons operator on vectors turned out to be more convenient.

N-perfect trees – Having defined the tuples, the family of LNDTs based on them follows, and is called N-perfect trees (N-PT)[**?**]:

```
16  N-PT : ℕ → TT
```

```
17  N-PT n = LNDT (Tuple n)
```

We can indeed notice that LNDTs based on `Tuple` of a certain rank `n` can actually be seen as (n+1)-perfect trees. Any value for `n` gives us a certain type of tree, with two examples being `List` and `Nest`. They are perfect in the sense that all the nodes at a given depth either have no child or (n+1) children, which means that the overall number of nodes is of the form $\sum_{i=0}^{k}(n+1)^i$, where k is the depth of the tree.

Lists, nests and bushes – The previous notions lead naturally to the definitions of lists, nests and bushes, seen as LNDTs with specific TTs:

```
18  List : TT
19  List = N-PT 0
```

```
20  Nest : TT
21  Nest = N-PT 1
```

```
22  Bush : TT
23  Bush = LNDT Bush
```

This definition of `Bush` requires Agda to ignore termination checking – although it is not shown here, it is required nevertheless –, which was not the case when defining $\texttt{Bush}_0$. This comes from the fact that positivity checking differs from termination checking. While the two definitions are equivalent, the first one relies on positivity checking because it is directly defined as a type data while the second one relies on termination checking since it is defined as an instance of LNDT. Such a termination checking cannot automatically succeed, which

makes the use of `Bush` as well as functions over `Bush` and $\texttt{Bush}_0$ unsafe. Coq rejects both definitions as unsafe and disallow their use, while Agda allows these definitions but considers them unsafe. This has the upside that working with bushes is possible in Agda, although the problem of termination checking to make their use safe will not be tackled in this paper.

2.4 `Maybe` / `Option`

Another possible instance of LNDT, and the last one tackled directly, in this work is actually the usual `Maybe` (`Option` in `ml`) type, which can be built from a specific type transformer, which we call `Null`, and which corresponds to the logical negation.

```
24  data ⊥ {a} : Set a where
25    - Empty type
```

```
26  Null : TT
27  Null _ = ⊥
```

The idea comes from the fact that such a type transformer will never be inhabited, thus only allowing the structure to be empty, or to contain a single element. And indeed, using the `pattern` keyword provided by Agda, we can provide an alternate definition of the `Maybe`.

```
28  Maybe : TT
29  Maybe = LNDT Null
```

```
30  pattern nothing = []
31  pattern just x = x :: []
```

This type has the right semantics which means that any element of type `Maybe` is either of the form `nothing` or `just x`. This has been proven in the framework although the proof is not presented here. While this alternate definition of `Maybe` is not necessarily relevant per se, it is interesting to see another way of building such a type, as well as to notice that Agda, thanks to the `pattern` mechanism, allows the developer to use this type exactly as one would use the usual `Maybe` type from the current Agda standard library

2.5 More LNDTs

As noticed in Section 2.2, for any `F : TT`, we have `LNDT F : TT` which means that we can keep building new interesting LNDTs by chaining multiple calls to `LNDT`. While we say "interesting", it is fair to say that not all such attempts indeed bear that characteristic. However, some do, and here is an example of a second degree LNDT that can be built, and from which multiple features will be retrieved for free, thanks to the spreadable properties depicted later on. We call this type `SquaredList`:

```
32  SquaredList : TT
```

```
33  SquaredList = LNDT List
```

Here is an example of inhabitant of `SquaredList`, starting with a natural number, then a list of natural numbers, then a list of lists of natural numbers, and so on:

```
34  squared-list-example : SquaredList ℕ
35  squared-list-example = 8 :: (4 :: 5 :: []) :: ((3 :: 6 :: []) :: (7 :: 1 :: 8 :: []) :: []) :: []
```

2.6 An overview of our LNDTs

Figure 1 shows an overview of the type transformers that were defined in LibNDT, and how they relate with one another. The light green squares represent concrete types while the others are abstract types which were used to build them, either by instantiation or by reference. For instance, considering the square labelled "Maybe" and the arrows that departs from it, it can be read this way "The type `Maybe` is the result of the instantiation of `LNDT` with `Null` as parameter".

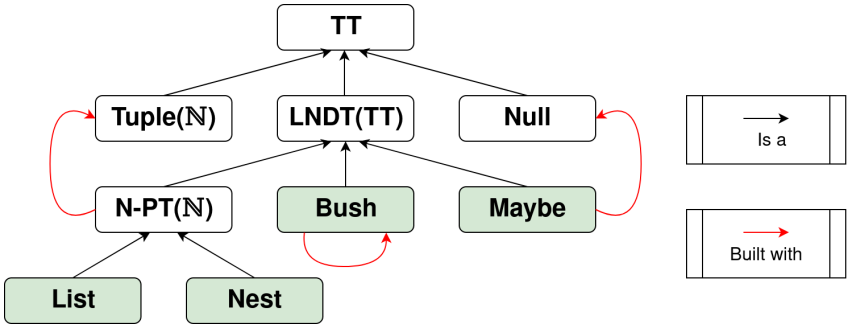

Figure 1 Type transformers in LibNDT

3 Computational common behaviours

However tempting regrouping these types under a common denominator is, the relevance of this process depends on the possibility to express behaviours at the LNDT level which could then be instantiated to any of its instances. The remaining of this paper stands as a list of argument in favour of this relevance, that is, a list of behaviours that can indeed be expressed for LNDTs. Such behaviours will be regrouped into 2 classes: computational ones, that is functions that affect the content of LNDTs, and logical ones, that is properties either directly bound to LNDTs or functions that work on them. In other words, the first class contains functions that process elements from LNDTs while the second class contains everything else. This section depicts our results regarding the first class of common behaviours.

When considering functions that work on collections of elements, a few examples come to mind, the first of which often is the *map* primitive. The reasoning behind *map* is well-known and natural: provided it is possible, using a certain function `f`, to transform elements of type `A` to elements of types `B`, it should be possible to transform a collection of elements of type `A` to a collection of elements of type `B` through a procedure, parametrized by `f`, called *map*. We begin by studying the relevance of this assumption for LNDTs.

3.1 Mapping LNDTs

Building map functions for LNDTs – As this is our first example of common behaviour, let us consider, in details, the steps in our reasoning. Since lists are the simplest example of LNDT we can start by considering the *map* function written on lists.

```
36  list-map₀ : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → List A → List B
37  list-map₀ f [] = []
38  list-map₀ f (x :: l) = f x :: list-map₀ f l
```

This is the common definition of maps for lists, which bears no noticeable difficulties. Writing a similar function for nests is somewhat more challenging because the tail of the nest is not of the same type of the nest itself, which means the same function `f` cannot be passed as it is in the recursive call.

```
39  nest-map₀ : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → Nest A → Nest B
40  nest-map₀ f [] = []
41  nest-map₀ f (x :: n) = f x :: nest-map₀ (λ {(a , b) → f a , f b}) n
```

However different, these two have a lot in common, which can be captured using LNDTs, as long as the way the function f must be transformed throughout the recursion is provided. By taking as parameter this transformation of function, and calling it `T`, we propose the following implementation for LNDTs.

```
42  lndt-map₀ : ∀ {a b} {A : Set a} {B : Set b} {F : TT} → (A → B) →
43    (T : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → (F A → F B))
44    → LNDT F A → LNDT F B
45  lndt-map₀ _ _ [] = []
46  lndt-map₀ f T (x :: e) = f x :: lndt-map₀ (T f) T e
```

While the signature of this function is not straightforward, it is possible to make it clearer by noticing a certain regularity in its core, which can be made visible as follows:

```
47  Map : TT → Setω
48  Map F = ∀ {a b} {A : Set a} {B : Set b} → (A → B) → F A → F B
```

This definition gives an abstract signature to type transformers `F` for which it is possible, given a function from `A` to `B`, to build a function from `F A` to `F B` that is, a map function. Using this new notation, the map function over LNDTs has a far better looking – and much more explicit – signature.

```
49  lndt-map : ∀ {F : TT} → Map F → Map (LNDT F)
50  lndt-map F f [] = []
51  lndt-map F f (x :: e) = f x :: lndt-map F (F f) e
```

This definition, through this new signature, gives us an important insight as to what `lndtmap` stands for: it is a procedure that ensure it is possible to map on LNDTs provided it is possible to map over the type transformer on which they are build. This is the first property that justifies the spreadable aspect of this library: we study which elements can be transported from `F` to `LNDT F`, and `Map` as defined earlier is one of them.

Instantiating map functions – In order to deduce map functions for LNDTs, all is needed is to define a map function for the TT on which they are based. The first function, which can be spread to the annotated LNDT, is called the seed for the associated behaviour. In this case, the behaviour in question is the ability to map over a given type. Here is the map seed for tuples:

```
52  tuple-map : ∀ n → Map (Tuple n)
53  tuple-map zero = id
54  tuple-map (suc n) f (a , ta) = f a , tuple-map n f ta
```

The definitions of maps for LNDTs are hence straightforward:

```
55  list-map = lndt-map (tuple-map 0)
56  nest-map = lndt-map (tuple-map 1)
```

```
57  bush-map = lndt-map bush-map
58  maybe-map = lndt-map λ _ ()
```

What is particularly interesting here is that the map function on bushes is recursively generated using itself, solely relying on `lndtmap` as a way of computing a result. In other words, the seeds for bushes will never have to be defined and any behaviour on bushes is always solely generated with the associated spreadable property.

Examples of usage of map functions – Here are examples where we first define a list of natural numbers on which we map the successor function, after which we define a bush of natural numbers on which we apply the multiplication by two function.

```
59  map₀ : list-map suc (3 :: 4 :: 2 :: 6 :: [])
60    ≡ 4 :: 5 :: 3 :: 7 :: []
61  map₀ = refl
```

```
62  map₁ : bush-map (_* 2) (3 :: (4 :: []) :: [])
63    ≡ (6 :: (8 :: []) :: [])
64  map₁ = refl
```

In both case, `refl` is a correctly typed term regarding the signature of the function, which means both sides of the equality are indeed the same. These examples are by no means a proof of correctness of `map` but rather a convincing argument if favour of our approach and definitions. They also show that working with bushes is completely possible in Agda, so long the termination checker is turned off for their definition. This can prove to be an issue, though, when trying to work in a safe manner – note that Agda itself has a safe option which disallow a certain number of options such as turning off the termination checker. This means that the problem of termination will need to be addressed at some point, and has already been tackled in a few manners using for instance other definitions of bushes with indices. However, this is not the goal of this work to handle such termination issues. We propose a global notion from which different concrete types can be defined, including bushes, and the fact itself that it is possible to work with them is interesting per se.

Back to squared lists – Squared lists are second degrees LNDTs, in the sense that they are built by a succession of two nestings. For such types. However, regardless of the number of successive nestings, we can provide a map function as long as the seed – the original TT – provides a map function itself. Here is the function in question for squared lists:

```
65  squared-list-map : Map SquaredList
```

```
66  squared-list-map = lndt-map list-map
```

As an example of usage of this newly created map function, we can apply a multiplication by two on all elements of the example squared lists defined in Section 2.5

```
67  squared-list-map-example : squared-list-map (_* 2) squared-list-example
68    ≡ 16 :: (8 :: 10 :: []) :: ((6 :: 12 :: []) :: (14 :: 2 :: 16 :: []) :: []) :: []
69  squared-list-map-example = refl
```

As shown in this example, nesting several times over a given TT does not alter our ability to provide free functions from the seed of the chain. Throughout this paper, more examples of spreadable behaviours will be given, all of which can be spread several times similarly. As a consequence, we will not explicitly go back to squared lists.

3.2 Folding LNDTs

Another common behaviour which can be defined over LNDTs directly are the fold operators, whether right or left. While we described the whole thought process behind the map operator, we will, from now on, only give the property that is spreadable and the associated definitions. Both folds are spreadable and can thus be transferred from `F` to `LNDT F`. Both folds share the same abstract type:

```
70  Fold : TT → Setω
71  Fold F = ∀ {a b} {A : Set a} {B : Set b} → (B → A → B) → B → F A → B
```

Then, assuming we can fold over the type parameter, we can propagate this fold to LNDT in two different manner, left or right. Here is the left propagation:

```
72  lndt-foldl : ∀ {F : TT} → Fold F → Fold (LNDT F)
73  lndt-foldl _ _ b [] = b
74  lndt-foldl foldl f b (x :: e) = lndt-foldl foldl (foldl f) (f b x) e
```

As an example of fold on an inner TT, here is the left fold on tuples:

```
75  tuple-foldl : ∀ n → Fold (Tuple n)
76  tuple-foldl zero = id
77  tuple-foldl (suc n) f b₀ (a , ta) = tuple-foldl n f (f b₀ a) ta
```

This leads to the definition of folds for our LNDTs. Here are the left folds for bushes and nests, with the respective seeds the fold on tuples, and itself.

```
78  nest-foldl : Fold Nest
79  nest-foldl = lndt-foldl (tuple-foldl 1)
```

```
80  bush-foldl : Fold Bush
81  bush-foldl = lndt-foldl bush-foldl
```

Here are examples of usage of these folds. In the first case, we concatenate the string from left to right in a nest, and in the second case, from right to left in a bush:

```
82  foldl₀ : nest-foldl _++_ "" ("a" :: ("u" , "r") :: (("e" , "l") , "i" , "a") :: [])
83    ≡ "aurelia" ; foldl₀ = refl
84  -
85  foldl₁ : bush-foldl _++_ "l" ("o" :: ("l" :: []) :: (("a" :: []) :: []) :: [])
86    ≡ "lola" ; foldl₁ = refl
```

3.3 Summary

This section exhibited three spreadable elements, that can be built freely for `LNDT F` when they exist on `F`, these are the two folds, left and right, and the map function. These elements can be regrouped inside a structure which contains all spreadable properties, which we call `SpreadAble`, and which will be enriched with logical properties from the next section. From the current element, it is possible to build other functions directly, such as size (the number of elements contained in a specific structure) and flatten, returning a list of these elements. This has been done in the library, however not shown here.

4 Logical one

5 Agda, Coq, Induction

Parler des principes d'induction, de la positivité, etc, Agsy, terminaison à mettre ici

6 Conclusion

MM : parler de coq à la carte

MM : parler de comment étendre notre approche, exemple de types non directement implémentables éventuellement, mais le tourner positivement

References

1 Ian Bayley. *Generic Operations on Nested Datatypes*. PhD thesis, University of Oxford, 2001.

2 Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction*, pages 52–67, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

3 RALF HINZE and ROSS PATERSON. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006. doi:10.1017/S0956796805005769.

4 Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/99370.99404.