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

Mathieu Montin
Université de Lorraine, Loria
CNRS, Inria, France
mathieu.montin@loria.fr

Amélie Ledein
Université Paris-Saclay
ENS Paris-Saclay, LMF
CNRS, Inria, France
amelie.ledein@inria.fr

Catherine Dubois
ENSIIE, IP Paris, Samovar, France
catherine.dubois@ensiie.fr

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 datatypes where the type parameter changes throughout the recursion, and have a variety of applications such as modelling memory or modelling constraints over regular datatypes without relying on dependent types. In this work, we focus on a specific subset of nested datatypes which we call *linked nested datatypes* (LNDT). We show that some usual datatypes such has *List* and *Maybe*, as well as some well-known nested datatypes such as *Nest* and even *Bush* can be built as various instances of LNDT. We proceed by presenting LIBNDT, a library, developed both in AGDA and COQ, which focuses on the set of constructs that can be propagated directly from the parameter on which a LNDT is built, to the LNDT itself. These constructs are of two kinds, functions, such as folds and map, and properties, such as the congruence of map or the satisfaction of a given predicate for at least one, or all, elements of the structure. We make use of the dependent type system of both COQ and AGDA to model the latter. We end by discussing various interesting topics that were raised throughout our development such as the issue of termination, the comparison of our tools and the proof effort required to extend LIBNDT with additional elements.

1 Introduction

Data structures are the key to properly handle many programming challenges. From the easiest algorithms to more advanced features or conceptually challenging programs, choosing the right data structure is often mandatory in programming activities. Functional programming, and associated languages, usually provide constructs to model such structures, which can be summed up as datatypes, where a type is defined with a list of constructors, each of which builds an element of the type using a given number of inputs. While imperative and object-oriented languages do have datatypes, they often manifest in a different manner, which will not be considered in this paper. These datatypes are widely used in programming activities and can model various concepts, depending on the type system of the language in which they are defined. They can represent concrete data, in usual functional programming languages such as OCAML and HASKELL, and even properties in dependently typed languages such as AGDA [14] and COQ [5], both of which have been used in this work.

As mentioned before, relying on relevant datatypes to conduct programming activities is essential, even more so when said types are meant to embed some high-level specification over concrete data. To help users to select the right datatype, they are categorized and studied. Our work takes place in that area. The simplest datatypes are the enumerations, where a type is built from a list of constants. More interesting datatypes allow constructors to have parameters. When one or more of these parameters are typed with the type that is being defined, the datatype in question recursive, a family of types which is all the more interesting. This family is particularly used and studied and are bound to the notion of

Submitted to:
MSFP 2022

induction. Among these recursive datatypes, there is a distinction between regular datatypes and non-regular datatypes [2, 3], also called nested datatypes [10].

A datatype is *regular* when its type parameter – if any – is always the same whenever it appears in the definition. In other words, if the type is polymorphic, then its type parameters are the same both in the signature of the type, as well as in the recursive constructors. Famous examples of regular datatypes are the lists and the trees, defined in HASKELL-like language as follows, with `a` being the type parameter:

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

A datatype is *non regular*, or nested, when its type parameter changes between the type signature and at least of its instances in the constructors. Well-known nested datatypes are the nests (sometimes called pow in the literature) and, most of all, the bushes, where the nesting (the way the type parameter changes) is done with the type definition itself, as shown below:

```
data Bush a = BLeaf | BNode (a, Bush (Bush a))
data Nest a = Zero a | Succ (Nest (a × a))
```

Nested datatypes [3] 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 an abstract notion, and try to theorize their use, regardless of their inner structure. Other works are particularly interested in the folds that can be written on these structures [4, 9, 13, 8, 15]. Folding nested values is indeed mandatory because, due to the constrained nature of their type, this is the main way users have to interact with the value itself. Furthermore, these folds have to, and can be, as generic as possible, allowing the definition of powerful iterators, as well as induction principles. In his thesis [2], Bayley is also interested in the genericity of other functions such as zip or membership. Our work shares this advocated approach of genericity around nested datatypes.

In order to extend the notions that can be generic over nested datatypes, we do not consider all their possible incarnations. Rather, we focus on a subset of all the possible nested datatypes, which we call *linked nested datatypes*. Moreover, we are interested in functions and properties that we can obtain for free from their definition, in the same spirit as *Theorems for free* [16] or the deriving mechanism of HASKELL [12], or even the use of Finger Trees [11, 1], a general nested datatype, parametrized by a monoid, the instantiation of which can lead to ordered sequences or interval trees. Indeed, our linked nested datatypes are characterized by a common structure with a changing type parameter, responsible for nesting the structure differently.

For that purpose, we have developed a core library, named LIBNDT, available on the first author's github page[1], in both AGDA and COQ, making it accessible to a large number of users. This library provides the users with several nested datatypes, defined as instances of LNDTS, as well as a core set of functions and properties that have been derived from the type parameter to the nested datatype itself. This papers presents the content of this library, with the following outline.

Section 2 presents the thought process behind the definition of our LNDTS, while also giving examples of datatypes that can be built from them. Section 3 and 4 provide examples of constructs that can be derived for our LNDTS from the corresponding definitions of the underlying type parameter. Section 3 focuses on computational such aspects, while Section 4 focuses on logical ones. Finally, Section 6 proposes a discussion around some limits and open questions that remain to be answered and that would benefit future work. Throughout these sections, snippets of code are presented, which come from the AGDA implementation of our work. These are type-checked pieces of code, ensuring their correctness, which is made possible through the use of `lagda`, a tool to combine AGDA code with LaTeX documents.

[1] `https://github.com/mmontin/libndt`

2 Introducing Linked Nested Data Types

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

The most common inductive datatype is the type of lists which consist in an arbitrary number of elements linked one after the other. Written inductively using AGDA, lists can be defined, as shown in the introduction, using two constructors: the constant empty list and the cons operator written in an infix manner (note that the subscript 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 parametrized by 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 parametrized by the same type parameter as its definition, namely `A`. This makes it a regular datatype rather than a nested one, where such a type parameter is assumed to vary throughout the recursion. As a first example of such a nested datatype, let us consider the usual `Nest` datatype:

```
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 datatype, where its type parameter evolves throughout the recursion using, in this case, the *Type Transformer* `A → A × A`. As visible, both `List` and `Nest` are very similar in their structure, 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 famous `Bush` nested datatype, 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 leading to a better picturing and understanding of bushes in the process. This is such a case where providing a relevant abstraction significantly ease the study of its concrete counterparts, which is especially true for bushes.

2.2 Linked Nested Data Types

The common denominator between the three aforementioned types is a structure we capture in our concept of LNDTs. LNDTs are parametrized by a type transformer, that is an entity which, given a level of universe `a` and a type living in `Set a`, provides another type living in `Set a`. In a more formal approach, we have `TT = ∀ b → Set b → Set b` in AGDA, leading to the following LNDT definition:

```
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 datatype applied to a certain type transformer is itself a type transformer, that is, for any `F` of type `TT`, we have that `LNDT F` also is a `TT`. This means that this library, although meant to define spreadable properties over LNDTs, is also defines in the process 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, once we provide the right type transformer for each of them. While the type transformer for bushes will be the type itself, we can notice that the type transformer required for `List` and `Nest` can be abstracted in a notion that we 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`. This is similar to the dependent types of vectors, where `Vec A n` stands for a list of `n` elements of type `A`. However, tuples are more convenient in our case because we never want them to be empty, which vectors can be, and they induce in our development some technical conveniences, on which we will not linger.

N-perfect trees Having defined the tuples, the family of LNDTs based on them follows. They are parametrized by the size of the tuple on which they depend:

```
16  N-PT : ℕ → TT
```

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

We can notice that LNDTs based on `Tuple` of a certain index `n` can actually be seen as $(n+1)$-perfect trees [18]. 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 $\sum_{i=0}^{k} (n+1)^i$, where k is the depth of the tree.

Lists, nests and bushes Thanks to the previous notions, we can finally jump to the definitions of lists, nests and bushes, seen as LNDTs with specific type transformers:

```
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 datatype 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 disallows 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, is actually – and surprisingly – the usual `Maybe` (`Option` in COQ) type which can be built from the `Null` type transformer, corresponding 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 always build empty types, thus only allowing the structure to be empty, or to contain a single element. And indeed, using the `pattern` keyword provided by AGDA, which allows the developer to define aliases usable in pattern-matching situations, we can provide an alternate definition of the type `Maybe` as an instance of LNDT.

```
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 LIBNDT 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. Moreover, discovering hidden patterns between types is always interesting.

2.5 More LNDTs

As noticed in Section 2.2, for any `F` of type `TT`, `LNDT F` has type `TT` too, 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 below 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 an 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 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 boxes represent concrete types while the others are abstract types which were used to build them. For instance, the box labelled "Maybe" and the arrows that depart from it, depict that the type `Maybe` is the result of the instantiation of `LNDT` with `Null` as a parameter.

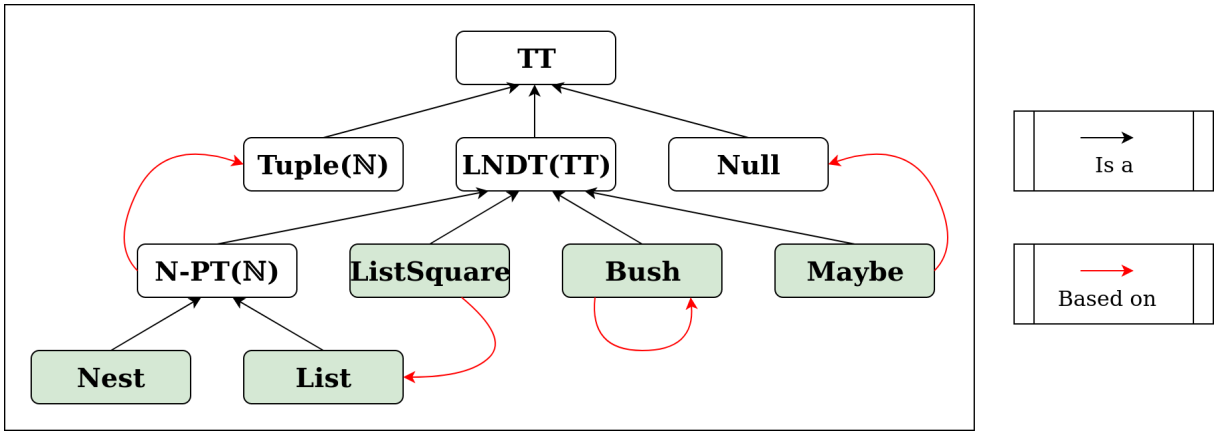

Figure 1: type transformers in LIBNDT

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 arguments in favor of this relevance, that is, a list of behaviours that can indeed be expressed for LNDTs. Such behaviours will be regrouped into two 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 is the *map* primitive. The reasoning behind *map* is well-known and natural: when possible, using some function `f`, to transform elements of type `A` to elements of type `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 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. Below is the common definition of maps for lists, which bears no understanding difficulties.

```
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
```

Writing a similar function for nests is 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
```

The latter definition gives us an important insight as to what `lndt-map` stands for: it is a procedure that ensures 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 that is needed is to define a map function for the type transformer 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 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.

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

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

In both cases, `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 in favour of our approach and definitions. They also show that working with `Bush` is completely possible in AGDA, as long as 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 disallows 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 `Bush` with indices. That is, a definition of `Bush` that keeps tracks of the current number of encapsulations. However, as mentioned earlier, 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 alone 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 two successive nestings. However, regardless of the number of successive nestings, we can provide a map function as long as the seed – the original type transformer– provides a map function itself. The map function for squared lists is obtained as follows:

```
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 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 type transformer 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 in the rest of the paper.

3.2 Folding LNDTs

Another common behaviour which can be defined over LNDTs directly are the fold functions, whether right or left. While we described the whole thought process behind the map function, we will, from now on, only give the property that is spreadable and the associated definitions. Both folds are spreadable and can thus be transported 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 manners, left or right. Below is the left propagation (the right one is omitted here but present in the library):

```
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 left fold on an inner type transformer, we define 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" :: ("r" , "t") :: (("i" , "c") , "l" , "e") :: [])
83    ≡ "article" ; foldl₀ = refl
84  --
85  foldl₁ : bush-foldl _++_ "m" ("s" :: ("f" :: []) :: (("p" :: []) :: []) :: [])
86    ≡ "msfp" ; 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 in 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 common behaviours

Until now, we considered functions on LNDTs which provide a concrete value from such types, without any type dependence to values of any kind. In other words, most of what has been shown earlier could have equally been developed in a classical functional programming language with polymorphic types. Yet, we work with dependent types, which enlarges the boundaries of what can be expressed around our LNDTs. This section shows examples of what we call logical properties, which means any definition about LNDTs whose type is dependent. They include primitive predicates over LNDTs, predicates around the computational aspects of LIBNDT alongside decidability properties.

4.1 Primitive predicates for LNDTs

4.1.1 Predicate transformers

Our first idea was to express the satisfaction of a predicate by all elements, or at least one element inside of our LNDTs, that is, given a predicate `P`, defining `All P` and `Any P` over any LNDT. The first step in that direction is to define the abstract type of such a transformation, from a predicate over a certain type `A` to a predicate over `F A` where `F` is a certain type transformer. The definition is as follows, where the predicate type is denoted by `Pred`.

```
87  TransPred : TT → Setω
88  TransPred F = ∀ {a b} {A : Set a} → Pred A b → Pred (F A) b
```

This specification now defined, we need to find inhabitants for them on our LNDTs, the semantics of which should be respectively `Any` and `All`. Both have been defined and can be found in the library, however, we only explain and present `Any` here. A LNDT is defined using a certain type transformer as a parameter, which means this type transformer needs to be associated with a predicate transformer itself, so that our LNDT can provide its extended version. In other words, we look from an inhabitant of `TransPred (LNDT F)` provided we have an inhabitant of `TransPred F`. This is built as an inductive datatype, similarly as the usual definition of `Any` over lists, for instance.

```
89  data Indt-any {F : TT} (T : TransPred F) {a b} {A : Set a} (P : Pred A b) :
90    Pred (LNDT F A) b where
91      here : ∀ {a x} → P a → Indt-any T P (a :: x)
92      there : ∀ {a x} → Indt-any T (T P) x → Indt-any T P (a :: x)
```

There are two cases, either the first element of the structure satisfies `P`, or one of the elements of the tail of the structure satisfies `P` nested with `T`, which is the predicate transformer associated with the underlying type transformer. In order to give examples, we define the predicate transformer `Any` over tuples, so that it can be propagated to nests and lists. This definition uses ⊎ which represents the logical union and consists of two cases: either the tuple contains a single element, in which case `P` itself is returned, or it contains more that one, in which case either the first element satisfies `P`, or one of the others satisfies it recursively.

```
93  tuple-any : ∀ n → TransPred (Tuple n)
94  tuple-any zero = id
95  tuple-any (suc n) P (a , t) = P a ⊎ tuple-any n P t
```

From this, we define `Any` for nests and bushes, relying on `Any` on tuples and itself respectively.

```
nest-any : TransPred Nest
nest-any = lndt-any (tuple-any 1)
```

```
bush-any : TransPred Bush
bush-any = lndt-any bush-any
```

Here is an example of `Any` over bushes, with the proof that the number 10 is a member of a bush. It can be found somewhere in the third bush, following the chain of `here` and `there`.

```
bush-any-example : bush-any (_≡ 10) (3 :: [] :: ((4 :: (7 :: []) :: []) :: ((10 :: []) :: []) :: []) :: [])
bush-any-example = there (there (here (there (here (here (here refl))))))
```

4.1.2 Decidability transformers

Since we are able to propagate a predicate transformer, we would like to tackle the propagation of the decidability of predicates. To do so, we express what it means for a predicate transformer to preserve decidability.

```
TransDec : ∀ {F : TT} → TransPred F → Setω
TransDec TP = ∀ {a b} {A : Set a} {P : Pred A b} → Decidable P → Decidable (TP P)
```

From this definition, we can prove that our predicate transformers `Any` and `All` over LNDTs do preserve decidability on the condition that the underlying predicate transformer does so. Proofs are omitted, although here is the signature relative to `lndt-any`:

```
lndt-dec-any : ∀ {F : TT} {T : TransPred F} → TransDec T → TransDec (lndt-any T)
```

More concretely, this means that in our library, the decidability of a predicate `P` can be propagated to the decidability of `Any` and `All` for all our LNDTs. In other words, we are able to decide if at least one, or all elements of a LNDT do satisfy `P`, even for bushes.

4.2 Predicates over computational aspects

A second logical family over LNDTs revolves around the satisfaction of predicates for the functions we have defined. This is a wide area that consists in giving specification for our functions and proving that they do satisfy their specification. Although our function are low-level and not composite, which makes the specification process all the harder, we have several such examples in our library, despite only one being presented here, the congruence of a mapping. In other words, we present the proof that mapping with a function `f` is the same as mapping with a function `g` on any LNDTs provided these function coincide on every input (aka are extensionally equal). We start by defining this abstract property.

```
MapCongruence : ∀ {F : TT} → Map F → Setω
MapCongruence map = ∀ {a b} {A : Set a} {B : Set b} (f g : A → B) →
  (∀ x → f x ≡ g x) → (∀ x → map f x ≡ map g x)
```

We provide the proof that our LNDTs preserve the congruence of a mapping. It means that, if `map` is congruent over a type transformer `F` then `lndt-map map` is congruent over `LNDT F`. The proof is done inductively, and is shown here as an example of such a proof in AGDA, although hardly understandable for non AGDA users, admittedly.

```
lndt-map-cong : ∀ {F : TT} {map : Map F}
  → MapCongruence map → MapCongruence (lndt-map map)
lndt-map-cong _ _ _ _ [] = refl
lndt-map-cong cgMap f g p (x :: v) rewrite p x =
  cong (g x ::_) (lndt-map-cong cgMap _ _ (cgMap f g p) v)
```

4.3 Summary

This section provided examples as to how dependent types can be used to build logical properties around our LNDTs. We showed that we can build predicates over LNDTs that stand for the membership moduloa predicate, `Any`, and the satisfaction of a predicate for all members of a LNDT, `All`. We showed that these were decidable provided the unary predicate over their elements is decidable. We also showed an example of specification for one of our computational behaviour: mapping with a given function, which stays congruent. All these elements were built with the same underlying idea used throughout this paper: the propagation of properties from `F` to `LNDT F` where `F` is a type transformer. Next section summarizes all our definitions in a single picture.

5 Picturing the spreadable elements of LIBNDT

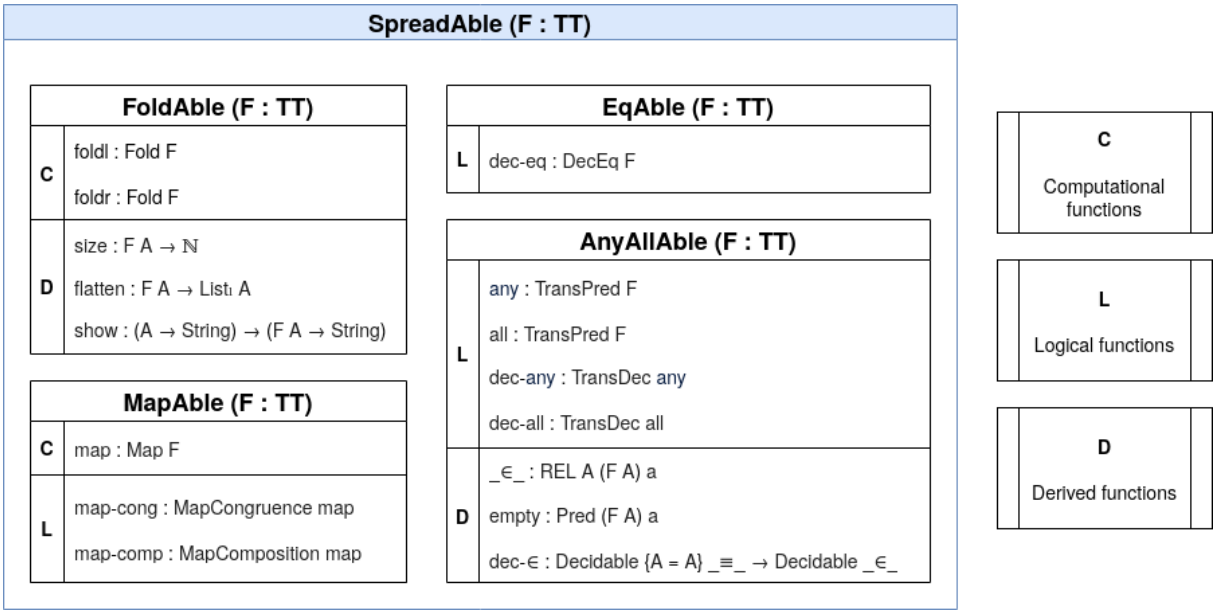

Figure 2: Overview of the library LIBNDT

Our spreadable properties, most of which have been mentioned throughout this paper, are regrouped into four categories: (1) `FoldAble` contains the two folds, left and right, with three derived functions, `size`, `flatten` and `show` which, respectively, counts the number of elements in our structure, flattens the structure into a list, and produces a string representation of the structure; (2) `MappAble` contains a map function, with two related properties, the congruence and the composition associated with it ; (3) `EqAble`

contains the proof of decidability of equality between LNDTs; (4) finally, `AnyAllAble` contains two predicate extensions over LNDTs, the "at least one element", and the "all elements" satisfying a given underlying predicate, alongside their decidability and additional derived constructs: the membership in a LNDT, its decidability, and the emptiness of a LNDT. All these elements are regrouped into a `SpreadAble` structure which is characterized by `SpreadAble F → SpreadAble (LNDT F)`.

6 Discussion

This last section brings together a set of observations and open questions that remain to be discussed and that we would like to underline. They are displayed in no particular order.

COQ and AGDA Our development was presented in AGDA, but is also available in COQ, as mentioned earlier. Having these two implementations of our ideas was fruitful in terms of providing users with our contribution, as well as in noticing differences between the tools. The main one lies in the definition of `Bush` which is accepted by AGDA and refused by COQ, regardless of the use of LNDT to define it (`Bush`) or not ($\texttt{Bush}_0$). This means that COQ is, in that regard, more restrictive than AGDA in the sense that it refuses any definition from which subsequent functions would be troublesome in terms of termination proofs. AGDA accepts these definitions – $\texttt{Bush}_0$ directly, `Bush` when termination checking is disabled – although subsequent definitions do indeed induce termination proofs issues in both cases. The good thing about being able to define `Bush` is that we can work with them by forcing the termination checker to trust us, although this leads to an unsafe development. Working in this manner is of course discouraged when dealing with safety issues, but it can prove to be interesting to handle this hard-to-grasp type. In our work, all functions that work on `Bush` do so in such a way that they are recursively defined using themselves, with a base case hard to picture, although they produce relevant results. This would be interesting to see how and if COQ could be extended to accept similar definitions, to what extend, and at which cost.

The issue of termination As mentioned several times throughout this paper, as well as in the previous observation, `Bush` is problematic in terms of termination. This is not surprising, and languages that allow us to work with them are not usually equipped with a termination checker. An open question is to what extend such a termination could be proven. Considering the definition of `Bush` provided in Section 2.3, `Bush = LNDT Bush`, it is highly unlikely that any termination checker would accept this definition, and that any of the usual termination techniques (exhibiting a well-founded order, using sized types...) would allow to extend it in a way that they do. To tackle this issue, it is possible to give an alternate, more verbose definition of `Bush` using an indexation of the number of times `Bush` is encapsulated. This was tested in our library, but it remains unclear to which extend this solves the termination issue while providing the same expressiveness. Furthermore, such a definition does not comply with our overall pattern captured by the notion of LNDT. Overall, the termination issue over `Bush` as well as similar datatypes remains open.

Automated term generation AGDA comes with an automated term generator, named AGSY, which, when called, attempts to build a term in a certain context with respect to a certain goal. AGSY works fine with LIBNDT, except for `Bush` where termination issues appear in the process of term generation. More precisely, while functions on `Bush`, although we are not able to prove it, do terminate, automated term generation over `Bush` do not. As far as we could observe, this behaviour might come from AGSY's will to explore possible terms with the maximum number of elements, rather than the opposite. This would

however be interesting to investigate deeper why this happens, and if this would be relevant to implement different heuristics inside AGSY and similar automated theorem provers, if possible, to fix this issue.

Extending LIBNDT with additional spreadable elements LIBNDT provides a representation of several datatypes using our notion of LNDTS, all of them consisting in a collection of elements structured in a head-tail manner. Each of the types that we model as instances of LNDT comes with a set of functions which are candidates for abstraction over LNDT. Most of the usual functions coming from `List`, for instance, have been considered as possible functions to define for LNDTS as a whole. Currently, LIBNDT consists of several LNDTS, on which 16 functions (either computational, logical or derived) can be derived automatically from the underlying type transformer to the corresponding LNDT. Possibly, many more functions could be added to this set, although a lot of them have been considered and ultimately proved impossible to abstract. An example of such a case is the idea of defining `map` using `fold` which seems both appealing, because it would reduce the minimal bricks of our LIBNDT, and promising, since `map` can indeed be written using `fold` when considering `List`. However, in practice, this is impossible due to type contradictions. This shows that common sense is not the best advisor in the matter. The common ancestor of both `map` and `fold`, when considering LNDTS, is the induction principle derived from the definition, and one cannot be written using the other. As a side note, this induction principle is successfully generated by COQ, and can be written in AGDA.

Another example of failed attempt at abstraction is the `zip` function, which cannot be extended to LNDTS for the same reason. We are confident, however, that more functions could be abstracted, especially in the logical area. For instance, we were interested in specifying contracts on our computation functions and were wondering if we could for instance prove, using our notion of membership and our notion of mapping that, given a LNDT `l` and an element `x` such that `x` is a member of `l`, the fact that `f x` is a member of `map f x`. This should hold, however, it proved to be hard both to express and to prove. We are confident that this can be proven in our library, however, this requires a strict discipline on how to reliably find invariants in recursive function over LNDTS, when the signature of the function contains elements that are not concerned by this recursivity. This remains an open question and will most likely be the subject of a future work.

Extending LIBNDT with additional datatypes Since, by nature, the work presented in this article only applies to a specific subset of nested datatypes, there are some other nested datatypes which are not handled in our work. An example are the well-known Finger Trees [1] which require an additional constructor to be defined – although some tricks using `Null` and `Maybe` could be considered. This observation leads to interesting questions as to how our work could be extended to nested datatypes that do not directly satisfy the structure we propose in LNDT. Such questions have been tackled over regular datatypes in different ways, which would be interesting to consider when relying on our work to use nested datatypes.

A first possibility is to rely on even higher abstractions, which means studying nested datatypes as a concept rather than as a concrete category of types. This is possible and has been done in other works such as [2]. However, this suffers from the usual drawback of high level abstraction: they are very far from concrete preoccupations and thus can hardly be used to obtain free code for their instances. In our case, the structure depicted in LNDT is essential to any of the concrete elements we have, which enforces a strong confidence in the level of abstraction we chose.

A second possibility is to resort to meta-programming, which exists in COQ with frameworks such as "Coq à la carte" [7] and which is currently under development in AGDA. Such a meta-programming

would allow us to define new datatypes at runtime, but it is not yet clear to us if this would allow us to reuse some of our result with little effort, if any.

A third promising possibility would be to use ornaments [17, 6]. Ornaments are a way to build a hierarchy between datatypes from which functions can be derived with a certain degree of automation. Using ornaments would possibly allow LIBNDT to handle many more nested datatypes.

7 Conclusion

In this paper, we have defined a restricted class of nested datatypes with a similar structure, the LNDTS. All different instances of LNDT are built using a different parameter called a type transformer. We have studied these types over two axes: first, we have assessed which types can successfully be seen as a LNDT, and second, we have built a set of functions, whether computation or logical, which can be derived automatically from the underlying type transformer to the corresponding LNDT. Throughout this investigation, we have developed a library regrouping these elements, called LIBNDT. This library has been developed both in COQ and in AGDA. The main difference between the two implementations is that AGDA allowed us to define the type `Bush`, on which all our functions can be applied. LIBNDT is a small library, which contains the most fundamental notions to working with LNDTS, and which will be extended with new elements in the future.

In that regard, our work has several limitations, most of which have already been discussed in Section 6, and can be summarized as follows. Our work is focused on a specific subset of nested datatypes, which is thus less generic than other higher level approaches, although it provides a bigger operational part. Our library could be used to model even more datatypes, and could be extended with additional spreadable properties. Moreover, our library does allow the user access to the `Bush` type when using AGDA, but this use is unsafe by nature, since termination issues are yet to be tackled. Finally, our approach provides a generic framework over LNDTS, although we do not provide a way to export the notions it contains to other kinds of nested datatypes.

These limitations bring perspectives to our work. We would like to extend LIBNDT to cover a wider ground in terms of possible instances of LNDT, as well as in terms of possible spreadable properties. We would also like to extend LIBNDT to handle more kinds of datatypes. This could be done using meta-programming or ornaments. We would also like to investigate some termination issues, based on `Bush` but not limited to. Indeed, our work contributes to raising the question of proofs of termination regarding recursive function when the defined function is passed as parameter inside its body to another function. Finally, we would like to investigate to which extend proof assistants should accept the type `Bush` and similar types, and the cost of such an acceptance.

References

[1] Available at http://www.staff.city.ac.uk/~ross/papers/FingerTree.html.

[2] Ian Bayley (2001): *Generic Operations on Nested Datatypes*. Ph.D. thesis, University of Oxford.

[3] Richard Bird & Lambert Meertens (1998): *Nested datatypes*. In Johan Jeuring, editor: *Mathematics of Program Construction*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–67.

[4] Richard Bird & Ross Paterson (1999): *Generalised Folds for Nested Datatypes*. *Formal Aspects of Computing* 11(2), pp. 200–222. Available at http://www.soi.city.ac.uk/~ross/papers/gfold.html.

[5] Pierre Castéran & Yves Bertot (2004): *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science, Springer Verlag. Available at https://hal.archives-ouvertes.fr/hal-00344237. Traduction en chinois parue en 2010. Tsinghua University Press. ISBN 9787302208136.

[6] Pierre-Evariste Dagand & Conor McBride (2014): *Transporting functions across ornaments*. *Journal of Functional Programming* 24(2-3), p. 316–383, doi:10.1017/S0956796814000069.

[7] Benjamin Delaware, Bruno C. d. S. Oliveira & Tom Schrijvers (2013): *Meta-Theory à La Carte*. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, Association for Computing Machinery, New York, NY, USA, p. 207–218, doi:10.1145/2429069.2429094. Available at https://doi.org/10.1145/2429069.2429094.

[8] Peng Fu & Peter Selinger: *Dependently Typed Folds for Nested Data Types*. arXiv:1806.05230v1.

[9] Ralf Hinze (1999): *Efficient Generalized Folds*.

[10] Ralf Hinze (1999): *Numerical Representations as Higher-Order Nested Datatypes*.

[11] Ralf Hinze & Ross Paterson (2006): *Finger trees: a simple general-purpose data structure*. *Journal of Functional Programming* 16(2), p. 197–217, doi:10.1017/S0956796805005769.

[12] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring & Andres Löh (2010): *A Generic Deriving Mechanism for Haskell*. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, Association for Computing Machinery, New York, NY, USA, p. 37–48, doi:10.1145/1863523.1863529. Available at https://doi.org/10.1145/1863523.1863529.

[13] Clare Martin, Jeremy Gibbons & Ian Bayley (2004): *Disciplined, efficient, generalised folds for nested datatypes*. *Formal Asp. Comput.* 16, pp. 19–35, doi:10.1007/s00165-003-0013-6.

[14] Ulf Norell (2009): *Dependently Typed Programming in Agda*. pp. 1–2, doi:10.1007/978-3-642-04652-0_5.

[15] Chris Okasaki (1998): *Purely Functional Data Structures*. Cambridge University Press, doi:10.1017/CBO9780511530104.

[16] Philip Wadler (1989): *Theorems for Free!* In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, Association for Computing Machinery, New York, NY, USA, p. 347–359, doi:10.1145/99370.99404. Available at https://doi.org/10.1145/99370.99404.

[17] Thomas Williams, Pierre-Évariste Dagand & Didier Rémy (2014): *Ornaments in Practice*. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, Association for Computing Machinery, New York, NY, USA, p. 15–24, doi:10.1145/2633628.2633631. Available at https://doi.org/10.1145/2633628.2633631.

[18] Yuming Zou & Paul E. Black (2019): *perfect binary tree*. Available at https://www.nist.gov/dads/HTML/perfectBinaryTree.html.