

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, Samovar
IP Paris, France
catherine.dubois@ensiie.fr

Nested datatypes have been widely studied in the past 25 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. They 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* (LNDDT). We show that some usual datatypes such as *List* and *Maybe*, as well as some well-known nested datatypes such as *Nest* and even *Bush* can be built as various instances of LNDDT. We proceed by presenting LIBNDT, a library, developed both in AGDA and COQ, which focuses on the set of constructs that can be *spread* directly from the parameter on which a LNDDT is built, to the LNDDT itself. These spreadable elements 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. This paper ends with a discussion about 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 key in handling 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 [3], 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 model some high-level specification over concrete data. To help users in selecting 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 set of constant values, modelled by a set of unparametrized constructors. 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 is recursive, a family of types which is all the more interesting. This family is

particularly used and studied and is tied to the notion of induction. Among these recursive datatypes, a distinction exists between regular datatypes and non-regular, also called nested, datatypes [2, 4, 9].

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 lists and trees, defined in HASKELL as follows, where a is the type parameter:

```
data List a = Empty | 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 one 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 Nest a = Zero a | Succ (Nest (a, a))
data Bush a = BLeaf | BNode (a, Bush (Bush a))
```

Nested datatypes have been widely studied since 1998 [4], both theoretically using category theory, and practically in programming languages such as HASKELL. Many approaches deal with nested datatypes as an abstract notion, and try to theorize their use, regardless of their inner structure. Others are particularly interested in the folds that can be written on these structures [5, 10, 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, leading to 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 number of 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* [17], the deriving mechanism of HASKELL [12], or even the use of Finger Trees [11, 1]. Finger Tree is 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.

To that purpose, we have developed a core library, named LIBNDT, available on the first author’s github page¹, 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 *spreadable* components that are elements derivable from the type parameter to the nested datatype itself. These spreadable elements 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. This paper 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 spreadable 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. Section 5 shows a visual summary of these elements. Finally, Section 6 proposes a discussion around some limitations and open questions that remain to be answered and that would benefit future works.

¹<https://github.com/mmontin/libndt>

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.

2 Introducing Linked Nested DataTypes (LNDTs)

2.1 Introductory examples: List_0 , Nest_0 and Bush_0

The most common inductive datatype is the type of lists which consists of an arbitrary number of elements linked one after the other. Written inductively using AGDA, lists can be defined, as usual, using two constructors: the constant empty list `[]` and the cons operator `..::_` written in an infix manner using underscores. We use the subscript $_0$ to mean this definition is not derived from 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 a parameter an element of type $\text{List}_0 A$ where List_0 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 Nest_0 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 for lists – takes as a parameter an element of type $\text{Nest}_0 (A \times A)$ where $A \times A$ is a pair of elements of type A . This makes Nest_0 a nested datatype, where its type parameter evolves throughout the recursion using, in this case, the function $A \rightarrow A \times A$, which we call a *type transformer*. As visible, both List_0 and Nest_0 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 Bush_0 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_0 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 can significantly ease the study of its concrete counterparts, which is especially true for bushes, and motivates our approach.

2.2 Definition of linked nested datatypes

The common denominator between the three aforementioned types is a structure we capture in our concept of LNDDTs. LNDDTs are parametrized by a type transformer, that is an entity which, given a level of universe a and a type living in $\text{Set } a$, provides another type living in $\text{Set } a$. In AGDA, our type transformers are called TT ; they live in $\text{Set}\omega$ (a sort above all others), and are defined as follows:

```

10  TT : Setω
11  TT = ∀ {a} → Set a → Set a

```

This leads to the following LNDDT definition:

```

12  data LNDDT (F : TT) {a} (A : Set a) : Set a where
13    [] : LNDDT F A
14    ... : A → LNDDT F (F A) → LNDDT 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 $\text{LNDDT } F$ also is a TT . Thus, LNDDT is informally of type $\text{TT} \rightarrow \text{TT}^2$. As a consequence LIBNDT, in addition to defining spreadable properties over LNDDTs, also introduces a certain number of type transformers in the process, as shown in Figure 1.

2.3 List, Nest and Bush as instances of LNDDT

Tuples Naturally, the three types considered as examples, List_0 , Nest_0 and Bush_0 can be written as instances of LNDDT, once we provide the right type transformer for each of them. While the type transformer for Bush_0 will be the type itself, we can notice that the type transformer required for List_0 and Nest_0 can be abstracted in a notion that we call **Tuple**:

```

15  Tuple : ℕ → TT
16  Tuple zero = id -- id is the identity function
17  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 $\text{Vec } A \ n$ 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 LNDDTs based on them follows. They are parametrized by the size of the tuple on which they depend:

```

18  N-PT : ℕ → TT
19  N-PT n = LNDDT (Tuple n)

```

²The signature of the related definition had to be tweaked a little to match Agda's distinction between type parameters and the sort of the datatype, a distinction marked by the colon.

We can notice that LNDRs based on `Tuple` of a certain index n can actually be seen as $(n + 1)$ -perfect trees [19]. Any value for n gives us a certain type of tree, with two examples being `List` and `Nest`, as defined in the following paragraph. 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 `List`, `Nest` and `Bush`, seen as LNDRs with specific type transformers:

<pre>20 List : TT 21 List = N-PT 0</pre>	<pre>22 Nest : TT 23 Nest = N-PT 1</pre>	<pre>24 Bush : TT 25 Bush = LNDR Bush</pre>
--	--	---

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 `Bush0`. 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 LNDR.

Such a termination checking cannot automatically succeed, which makes the use of `Bush` as well as functions over `Bush` and `Bush0` 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 remains and will be further discussed in Section 6.

Examples Here is an example of `List`:

```
26 list-example : List ℕ
27 list-example = 2 :: 3 :: 7 :: 11 :: 13 :: 17 :: 19 :: 23 :: 29 :: 31 :: 37 :: 41 :: 43 :: 47 :: 53 :: 59 :: []
```

An example of `Nest`:

```
28 nest-example : Nest ℕ
29 nest-example = 4 :: (4, 5) :: ((6, 56), (81, 7)) :: (((5, 4), (1, 7)), ((7, 6), 4, 7)) :: []
```

And an example of `Bush`:

```
30 bush-example : Bush ℕ
31 bush-example = 5 :: (16 :: []) :: ((87 :: [])) :: ((56 :: []) :: []) :: []
```

2.4 Maybe as instance of LNDR

Another possible instance of LNDR, is actually – and surprisingly – the usual `Maybe` type (`Option` in ML-like languages) which is built from the `Null` type transformer, corresponding to the logical negation.

```

32 data ⊥ {a} : Set a where
33   -- Empty type

```

```

34 Null : TT
35 Null _ = ⊥

```

The idea comes from the fact that such a `Null` type transformer always builds empty types, thus only allowing the structure to be empty (using `[]`), or to contain a single element (using `_::_`), since any further nesting will not be inhabited.

In addition, by 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 `Maybe` type as an instance of `LNDT`, by mapping `nothing` to `[]` and `(just x)` to `(x :: [])`. By doing so, we ensure that this definition and the usual one are syntactically equivalent, in addition to behaving the same way.

```

36 Maybe : TT
37 Maybe = LNDT Null

```

```

38 pattern nothing = []
39 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`, as expected. This has been proven in `LIBNDT` although the proof is not presented here. While defining `Maybe` in this alternate manner 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 often enlightening, and always exciting.

2.5 Multi-layered 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`:

```

40 SquaredList : TT
41 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:

```

42 squared-list-example : SquaredList ℕ
43 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. In the case of the `Bush` type, the label of the related box is underlined to indicate that the definition is not safe.

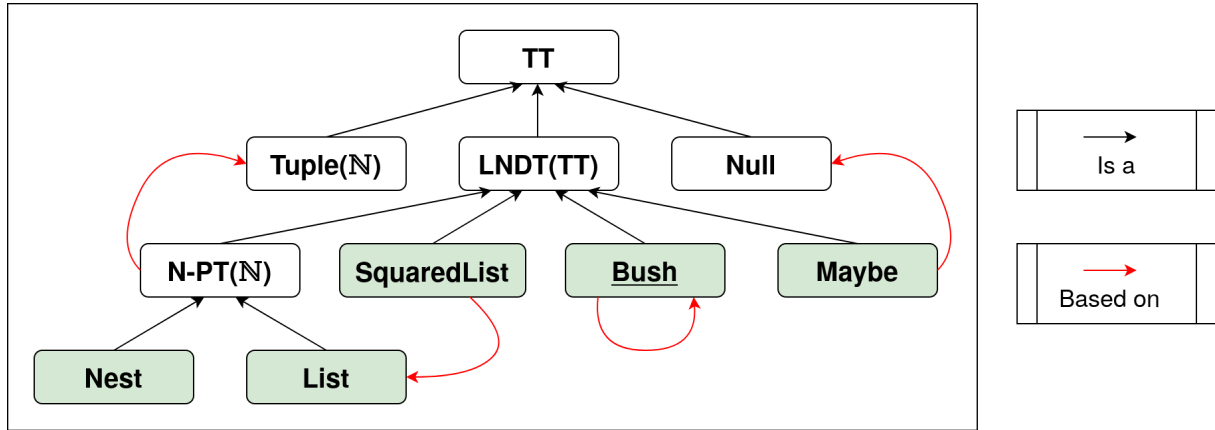


Figure 1: Type transformers in LIBNDT

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 LNDD level which could then be instantiated to any of its instances. The remaining of this paper stands as a list of arguments in favour of this relevance, that is, a list of behaviours that can indeed be expressed for LNDDs. Such behaviours will be regrouped into two classes: *computational* ones, that is concrete functions relying on the content of LNDDs, and *logical* ones, that is properties either directly bound to LNDDs or functions that work on them. In other words, the first class contains functions that process elements from LNDDs while the second class contains everything else. This section depicts our results regarding the first class of common behaviours.

3.1 Mapping LNDDs

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 LNDDs.

Building map functions for LNDDs 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 LNDD, we can start by considering the usual map function written on lists. Below is the common definition of maps for lists (subscripted by 0), which bears no understanding difficulties.

```

44 list-map0 : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → List A → List B
45 list-map0 f [] = []
46 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.

```

47 nest-map0 : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → Nest A → Nest B
48 nest-map0 f [] = []
49 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.

```

50 lndt-map0 : ∀ {a b} {A : Set a} {B : Set b} {F : TT} → (A → B) →
51   (T : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → (F A → F B))
52   → LNDT F A → LNDT F B
53 lndt-map0 _ - [] = []
54 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:

```

55 Map : TT → Set ω
56 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.

```

57 lndt-map : ∀ {F : TT} → Map F → Map (LNDT F)
58 lndt-map F f [] = []
59 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 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 corresponding 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:

```

60 tuple-map : ∀ n → Map (Tuple n)
61 tuple-map zero = id
62 tuple-map (suc n) f (a , ta) = (f a , tuple-map n f ta)

```

The definitions of maps for LNDTs are hence straightforward:

<pre>63 list-map = lndt-map (tuple-map 0) 64 nest-map = lndt-map (tuple-map 1)</pre>	<pre>65 bush-map = lndt-map bush-map 66 maybe-map = lndt-map (λ _ ())</pre>
--	---

We can notice that the map function on bushes is recursively generated using itself, solely relying on `lndt-map` 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 As a first example, here is a list of natural numbers on which the successor function is mapped:

<pre>67 list-map-example : list-map suc (3 :: 4 :: 2 :: 6 :: []) ≡ 4 :: 5 :: 3 :: 7 :: [] 68 list-map-example = refl</pre>
--

As a second example, we define a bush of natural numbers on which we map the function of multiplication by two:

<pre>69 bush-map-example : bush-map (λ x 2) (3 :: (4 :: []) :: []) ≡ (6 :: (8 :: []) :: []) 70 bush-map-example = refl</pre>
--

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 the map functions, but rather a convincing argument in favour of our approach and definitions.

Back to squared lists Squared lists are second degree 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:

<pre>71 squared-list-map : Map SquaredList 72 squared-list-map = lndt-map list-map</pre>
--

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.

<pre>73 squared-list-map-example : squared-list-map (λ x 2) squared-list-example 74 ≡ 16 :: (8 :: 10 :: []) :: ((6 :: 12 :: []) :: (14 :: 2 :: 16 :: []) :: []) :: [] 75 squared-list-map-example = refl</pre>
--

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, right or left, which walk through the structure while combining their content using a given operator and a given seed. 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 $\text{LNDT } F$. Both folds share the same abstract type:

```
76 Fold : TT → Setω
77 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 ways, left or right. Below is the left propagation, while the right one is also present in the library:

```
78 lndt-foldl : ∀ {F : TT} → Fold F → Fold (LNDT F)
79 lndt-foldl _ _ b [] = b
80 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:

```
81 tuple-foldl : ∀ n → Fold (Tuple n)
82 tuple-foldl zero = id
83 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.

<pre>84 nest-foldl : Fold Nest 85 nest-foldl = lndt-foldl (tuple-foldl 1)</pre>	<pre>86 bush-foldl : Fold Bush 87 bush-foldl = lndt-foldl bush-foldl</pre>
---	--

Here is an example of left folding a nest of strings:

```
88 foldl₀ : nest-foldl _+_+ " " ("a" :: ("r" , "t") :: ((("i" , "c") , ("l" , "e")) :: []) ≡ "article"
89 foldl₀ = refl
```

And an example of left folding a bush of strings:

```
90 foldl₁ : bush-foldl _+_+ "m" ("s" :: ("f" :: []) :: ((("p" :: []) :: []) :: []) ≡ "msfp"
91 foldl₁ = refl
```

3.3 Summary

This section exhibited three spreadable elements, that can be built *freely* for $\text{LNDT } F$ when they exist on F , these are the two folds, left and right, and the map function. These elements can be regrouped inside a record which contains all spreadable properties, which we call `Spreadable`, and which will be enriched with logical properties in the next section. From these elements, it is possible to build additional 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 but is 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 is to express the satisfaction of a predicate by all elements, or at least one element inside our LNDTs. In other words, we want to define, given a predicate P , the usual predicates $\text{All } P$ and $\text{Any } P$ over any LNDT.

The first step in that direction is to define the type of predicate transformer, transforming 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 Pred is the AGDA type for unary predicates ($\text{Pred } A \ b = A \rightarrow \text{Set } b$):

```

92  TransPred : TT → Setω
93  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 show Any in details here. Since a LNDT is defined using a certain type transformer as a parameter, 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 $\text{TransPred } F$. This is built as an inductive datatype, similarly as the usual definition of Any over lists, for instance.

```

94  data Indt-any {F : TT} (T : TransPred F) {a b} {A : Set a} (P : Pred A b) :
95    Pred (LNDT F A) b where
96    here : ∀ {a x} → P a → Indt-any T P (a :: x)
97    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.

```

98  tuple-any : ∀ n → TransPred (Tuple n)
99  tuple-any zero = id
100 tuple-any (suc n) P (a , t) = P a ⊔ tuple-any n P t

```

This definition uses \uplus which represents in AGDA 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 than one, in which case either the first element satisfies P , or one of the others satisfies it recursively. From this, we define `Any` for nests and bushes, relying on `Any` on tuples and itself respectively.

<pre>101 nest-any : TransPred Nest 102 nest-any = Indt-any (tuple-any 1)</pre>	<pre>103 bush-any : TransPred Bush 104 bush-any = Indt-any bush-any</pre>
--	---

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` from `Indt-any`.

<pre>105 bush-any-example : bush-any (≡ 10) (3 :: [] :: ((4 :: (7 :: [])) :: [] :: ((10 :: [])) :: [] :: [])) 106 bush-any-example = there (there (here (there (here (here (here (here refl)))))))</pre>
--

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.

<pre>107 TransDec : ∀ {F : TT} → TransPred F → Set ω 108 TransDec TP = ∀ {a b} {A : Set a} {P : Pred A b} → Decidable P → Decidable (TP P)</pre>
--

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 `Indt-any`:

<pre>109 Indt-dec-any : ∀ {F : TT} {T : TransPred F} → TransDec T → TransDec (Indt-any T)</pre>

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 functions 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 functions coincide on every input (aka are extensionally equal). We start by defining this abstract property.

```

110 MapCongruence : ∀ {F : TT} → Map F → Setω
111 MapCongruence map = ∀ {a b} {A : Set a} {B : Set b} (f g : A → B) →
112   (∀ 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.

```

113 Indt-map-cong : ∀ {F : TT} {map : Map F}
114   → MapCongruence map → MapCongruence (Indt-map map)
115 Indt-map-cong _ _ _ _ [] = refl
116 Indt-map-cong cgMap f g p (x :: v) rewrite p x =
117   cong (g x ::_) (Indt-map-cong cgMap _ _ (cgMap f g p) v)

```

4.3 Summary

This section provided examples of logical properties expressed around our LNDTs. We showed that we can build predicates over LNDTs that stand for the membership modulo a predicate, named `Any`, and the satisfaction of a predicate for all members of a LNDT, named `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

Our spreadable properties, most of which have been mentioned throughout this paper, are regrouped into four categories, each corresponding to a concrete data structure in LIBNDT, modelled as a record:

1. `FoldAble` contains the two folds, `left` and `right`, with three additional 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. `MapAble` 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. `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` record which contains all spreadable elements of the library, and thus is characterized by `SpreadAble F → SpreadAble (LNDT F)`.

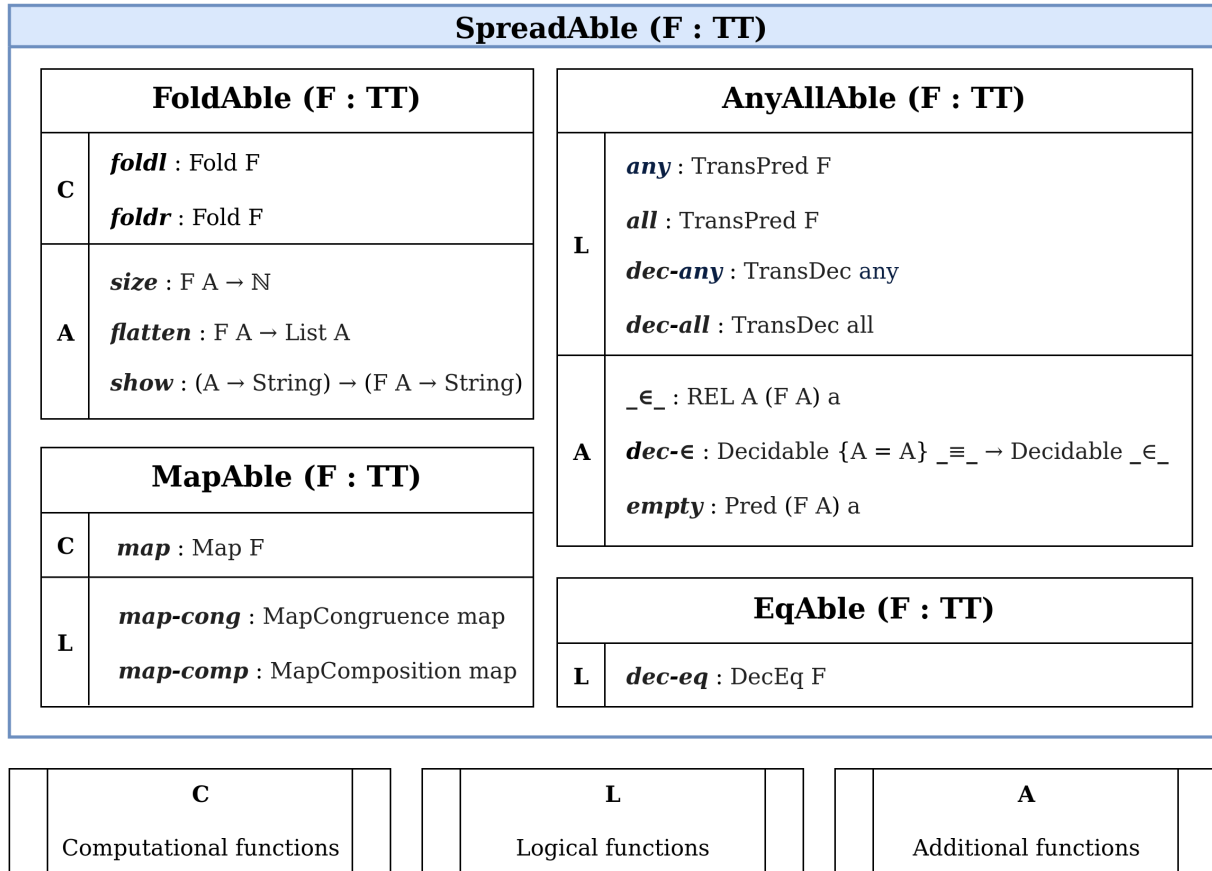


Figure 2: Overview of the library LIBNDT

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. Most of these elements have been mentioned throughout the paper and need additional explanation. 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 (*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 – *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 extent, and at which cost.

The issue of termination As mentioned earlier, Bush is troublesome in terms of termination, which is unsurprising considering the nature of this datatype. In that regard, languages in which bushes can be used do not usually embed a termination checker and instead rely on trusting the developers ; and when they do like in AGDA, it needs to be turned off for the time being. An open question is to what extent such a termination could be proven. Considering the definition of Bush provided in Section 2.3, $\text{Bush} = \text{LNNT Bush}$, it is highly unlikely that any termination checker would ever accept this definition, and that any of the usual termination techniques (exhibiting a well-founded order, using sized types, etc.) 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 definition is accepted by AGDA but also by COQ, as shown by the following snippet:

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

This was tested in our library, but it remains unclear to which extent 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 LNNT. Overall, the termination issue over Bush 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 LIBNNT, 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 does not. As far as we could observe, this behaviour might come from AGSY's heuristic 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, to fix this issue.

Extending LIBNNT with more spreadable elements LIBNNT provides a representation of several datatypes using our notion of LNNTs, all of them consisting in a collection of elements structured in a head-tail manner. Each of the original types that we model as instances of LNNT comes in the literature with a set of candidate functions for abstraction over LNNT. Most of the usual functions coming from List, for instance, have been considered as possible functions to define for LNNTs as a whole. Currently, LIBNNT consists of several LNNTs, on which 16 functions (either computational, logical or additional) can be derived automatically from the underlying type transformer to the corresponding LNNT. 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 LIBNNT, and promising, since map can indeed be written using fold when considering List. However,

in practice, this is impossible due to type requirements. Indeed, the intuitive way of building `map` using `fold`, which works for lists, is the following:

$$\text{map } f = \text{foldr } (\lambda a l \rightarrow f a :: l) []$$

This does not transpose into any LNDT because, `l` is of type $\text{LNDT } (F \ B)$ although `f a :: l` expects an element of type $\text{LNDT } (F \ (F \ B))$, where `f` is of type $A \rightarrow B$. 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 $F \circ F = F$. 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 because the structures need to be similar to be zipped together, and for instance any two bushes have little chance to be similar enough. 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 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. These possibilities are as follows:

1. A first possibility is to rely on even higher abstractions, which means studying nested datatypes as an abstract notion rather than a concrete family of inductive 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.
2. A second possibility is to resort to meta-programming, which exists in COQ with frameworks such as “Coq à la carte” [7] or more recently “MetaCoq” [16] 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.
3. A third promising possibility would be to use ornaments [18, 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

Assessments In this paper, we have defined a restricted class of nested datatypes with a similar structure, the LNDDTs. All different instances of LNDDT are built using a different parameter called a type transformer. We studied these types over two axes: first, we studied a family of types that can successfully be seen as a LNDDT, and second, we built a set of functions, whether computation or logical, which can be derived automatically from the underlying type transformer to the corresponding LNDDT. Throughout this investigation, we developed a library regrouping these elements, called LIBNDDT. This library was 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. LIBNDDT is a small library, which contains the most fundamental notions to working with LNDDTs, and which will be extended with new elements in the future.

Limitations 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 LNDDTs, although we do not provide a way to export the notions it contains to other kinds of nested datatypes.

Perspectives These limitations bring perspectives to our work. We would like to extend LIBNDDT to cover a wider ground in terms of possible instances of LNDDT, as well as in terms of possible spreadable properties. We would also like to extend LIBNDDT 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] doi:10.1017/S0956796805005769. 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] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions*. doi:10.1007/978-3-662-07964-5.
- [4] Richard Bird & Lambert Meertens (1998): *Nested datatypes*. In Johan Jeuring, editor: *Mathematics of Program Construction*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–67, doi:10.1007/BFb0054285.
- [5] Richard Bird & Ross Paterson (1999): *Generalised Folds for Nested Datatypes*. *Formal Aspects of Computing* 11(2), p. 200–222, doi:10.1007/s001650050047.
- [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.
- [8] Peng Fu & Peter Selinger: *Dependently Typed Folds for Nested Data Types*. arXiv:1806.05230v1.
- [9] Ralf Hinze (1998): *Numerical Representations as Higher-Order Nested Datatypes*. Technical Report.
- [10] Ralf Hinze (1999): *Efficient Generalized Folds*. Technical Report.
- [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 Löb (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.
- [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] Matthieu Sozeau, Abhishek Anand, Simon Boulrier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project*. *J. Autom. Reason.* 64(5), pp. 947–999, doi:10.1007/s10817-019-09540-0.
- [17] 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.
- [18] 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.
- [19] Yuming Zou & Paul E. Black (2019): *perfect binary tree*. Available at <https://www.nist.gov/dads/HTML/perfectBinaryTree.html>.