Implementation and verification of a data structure using dependent types

Anton Danilkin

April 27, 2022

Abstract

1 Introduction

One of the most important data structures in functional programming are singly linked lists, which support prepending an element in the beginning, as well as destructing a non-empty list into its head (the first element) and tail (the list containing the rest of elements). Conversely, arrays are frequently used in imperative programming; they support getting and setting an element by its index, as well as, in case of vectors, dynamic resizing.

The problem of lists is that the operation of retrieving or updating an element at a specific index has linear spacial complexity in term of the size. On the flip side, operations on arrays and vectors mutate the instance they work on, which not only makes reasoning about program behavior harder, but also means that older versions of the data structure become unaccessible for future use. One way to fix that would be making a copy before each operation, but that again would mean that everything would have linear complexity.

For these reasons, the following question arises: is it possible to have a data structure that combines the benefits of singly linked lists and vectors?

2 Similarities between numbers and lists

Here are some standard definitions of Peano natural numbers and lists, as well as operations on them in Coq:

```
Inductive nat :=
  | O : nat
  | S : nat → nat.
```

```
Inductive list A :=
  | nil : list A
  | cons : A → list A → list A.

Arguments nil {A}.
Arguments cons {A} x l.
```

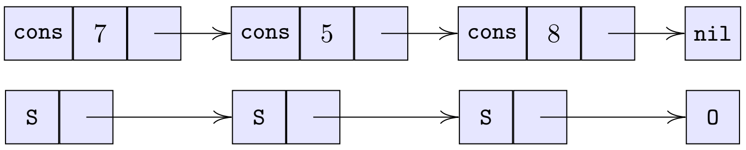
```
Definition pred (n : nat) : nat :=
  match n with
  | O ⇒ O
  | S n′ ⇒ n′
  end.
```

```
Definition tl {A} (l : list A) : list A :=
  match l with
  | nil ⇒ nil
  | cons _ l′ ⇒ l′
  end.
```

```
Fixpoint plus (n1 n2 : nat) : nat :=
  match n1 with
  | O ⇒ n2
  | S n1′ ⇒ S (plus n1′ n2)
  end.
```

```
Fixpoint app {A} (l1 l2 : list A) : list A :=
  match l1 with
  | nil ⇒ l2
  | cons x l1′ ⇒ cons x (app l1′ l2)
  end.
```

As remarked by Chris Okasaki [Oka98], there is a clear resemblance between them: the only real difference is that `list A` holds a datum of type `A`, where as `nat` does not. Here is how the list [7; 5; 8] and the number 3 (which is the length of the list) could be represented:



In fact, we can go from a list to the corresponding natural number by getting its length, and back from a natural number to one of the corresponding lists by repeating an element that many times:

```
Fixpoint length {A} (l : list A) : nat :=
  match l with
  | nil ⇒ O
  | cons _ l′ ⇒ S (length l′)
  end.
```

```
Fixpoint repeat {A} (x : A) (n : nat) : list A :=
  match n with
  | O ⇒ nil
  | S n′ ⇒ cons x (repeat x n′)
  end.
```

3 Binary version

As there are many ways to represent natural numbers (and not only the unary system that was considered above). Depending on which representation we chose, operations on the numbers (such as increment, decrement, sum, converting into other representations) will have different efficiency, and in each case we can find an analogous data structure.

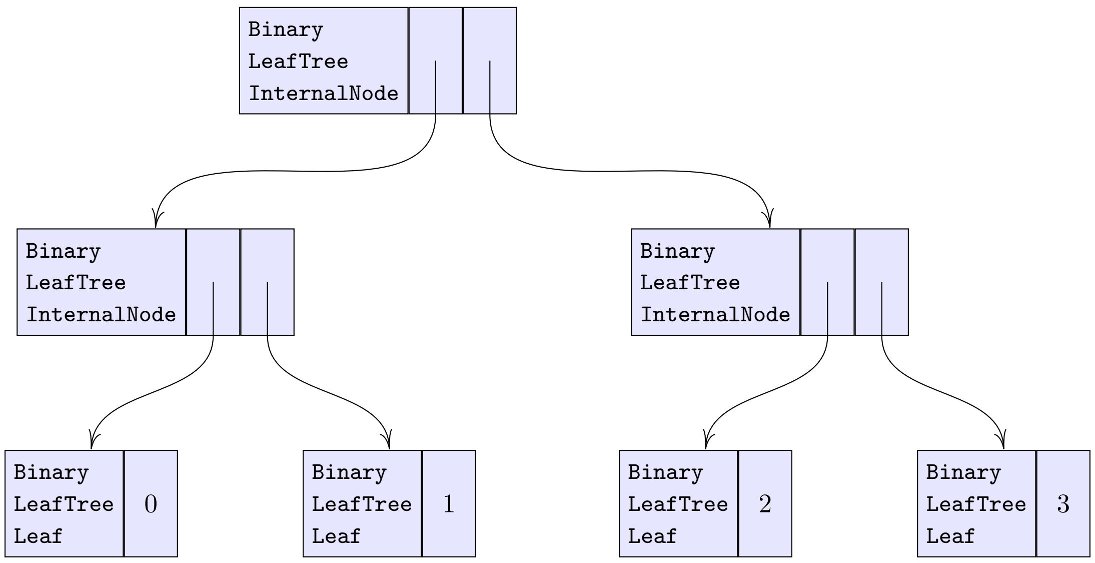
So another simple representation of natural numbers is the binary numeral system. As seen, again, by Chris Okasaki [Oka98], we can augment it to contain pieces of data by complete binary leaf trees in each digit of the number (although here we will use big-endian digit ordering instead of little-endian).

For the following, we will denote by `A` some data type. A "complete binary leaf tree" of depth d has 2^d leaves where it stores values of type `A`.

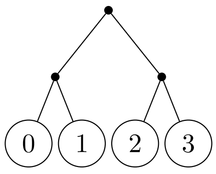
```
Inductive binary_leaf_tree {A} :=
  | BinaryLeafTreeLeaf : A → binary_leaf_tree
  | BinaryLeafTreeInternalNode : binary_leaf_tree → binary_leaf_tree → binary_leaf_tree.

Arguments binary_leaf_tree : clear implicits.
```

Here is an example of a complete binary leaf tree of depth 2 holding values 0, 1, 2, 3:



They will be drawn like that in the future:



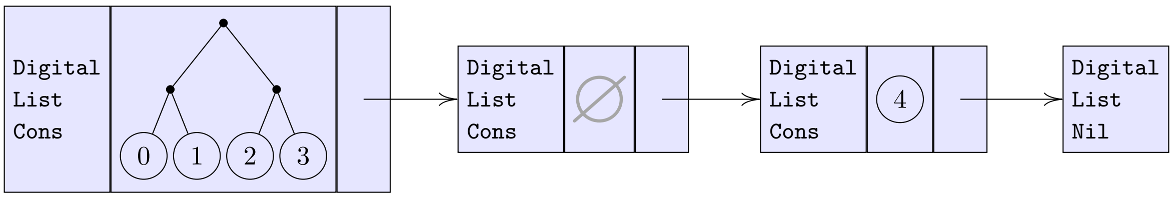
We know that for each natural number n there are $d \in \mathbb{N}$ and $n_{d-1}, n_{d-2}, ..., n_0, n_1 \in \{0,1\}$ such that $n = n_{d-1}2^{d-1} + n_{d-2}2^{d-2} + ... + n_1 2^1 + n_0 2^0$. The function `to_digits` : ∀(`d n` : `nat`), `list nat` gives the list of length `d` consisting of these binary digits of `n` in big-endian ordering: for example, `to_digits` 8 73 $\rightsquigarrow$ [0; 1; 0; 0; 1; 0; 0; 1] (where "$\rightsquigarrow$" means "evaluates to"). The inverse operation is `from_digits` : ∀(`d` : `nat`)(`nl` : `list nat`), `nat`: for example, `from_digits` 8 [0; 1; 0; 0; 1; 0; 0; 1] $\rightsquigarrow$ 73.

A "digital list" of depth d is a list of length d of which the k'th element is a complete binary leaf tree of depth k if $n_k = 1$ and nothing otherwise.

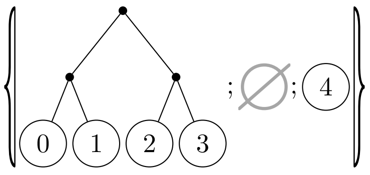
```
Inductive digital_list {A} :=
  | DigitalListNil : digital_list
  | DigitalListCons : option (binary_leaf_tree A) → digital_list → digital_list.

Arguments digital_list : clear implicits.
```

Here is an example of a digit list for $n = 5$, so $d = 3$, $n_2 = 1$, $n_1 = 0$, $n_2 = 1$, which stores values 0, 1, 2, 3, 4:



Or, to simplify the drawing, the following notation will be used form now on:

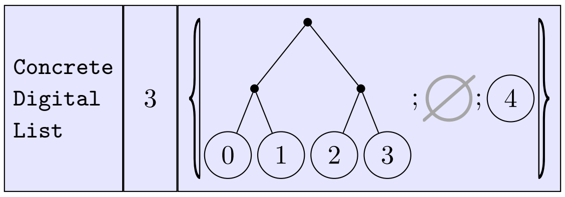


A "concrete digital list" is a $d \in \mathbb{N}$ and a digital list of depth d (one of the reasons to store the depth explicitly is so that it can be accessed immediately).

```
Inductive concrete_digital_list {A} :=
  | ConcreteDigitalList : ∀(d : nat), digital_list A → concrete_digital_list.

Arguments concrete_digital_list : clear implicits.
```

The following concrete digital list corresponding to the digital list given above:

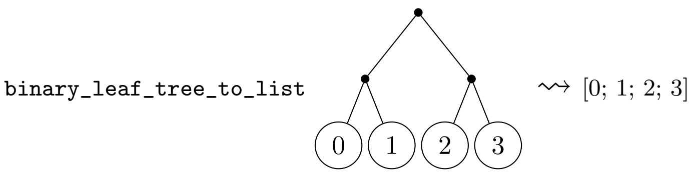


Next we will consider operations that we can do with these types. We will define each operation on binary leaf trees, then lift it to digital lists (always having a "depth" argument, that corresponds to the depth of the digital list passed, for consistency and for reasons outlined later, even when it is not actually used), and finally to concrete digital lists.

3.1 Interpretation

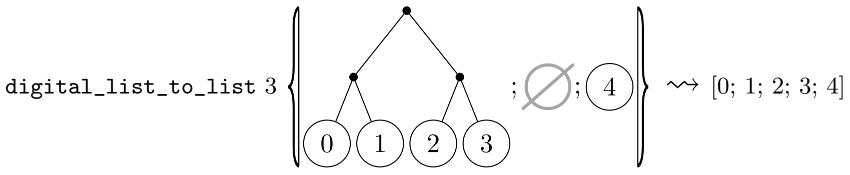
The first operation will be conversion to a simple list, which at the same times gives an "interpretation" of the data structures, allowing us to establish a convection between the operations on them and ones on the regular lists, which means proving the correctness of these operations.

To convert a binary leaf tree to a list, we just traverse its leaves in their natural order. For example:



```
Fixpoint binary_leaf_tree_to_list {A} (blt : binary_leaf_tree A) :=
  match blt with
  | BinaryLeafTreeLeaf x ⇒ [x]
  | BinaryLeafTreeInternalNode blt′1 blt′2 ⇒
    binary_leaf_tree_to_list blt′1 ++ binary_leaf_tree_to_list blt′2
  end.
```

For a digital list, we convert each binary leaf tree (if it exists) to a list and then concatenate the results. For example:



```
Fixpoint digital_list_to_list {A} d (dl : digital_list A) :=
  match dl with
  | DigitalListNil ⇒ []
  | DigitalListCons o dl′ ⇒
    match o with
    | None ⇒ []
    | Some blt ⇒ binary_leaf_tree_to_list blt
    end ++ digital_list_to_list (pred d) dl′
  end.
```

The version for concrete digital lists is straight-forward:

```
Definition concrete_digital_list_to_list {A} (cdl : concrete_digital_list A) :=
  let ′(ConcreteDigitalList d dl) := cdl in digital_list_to_list d dl.
```

References

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. DOI: 10.1017/CBO9780511530104.