Implementation and verification of a data structure using dependent types

Anton Danilkin

April 29, 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 design 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.
```

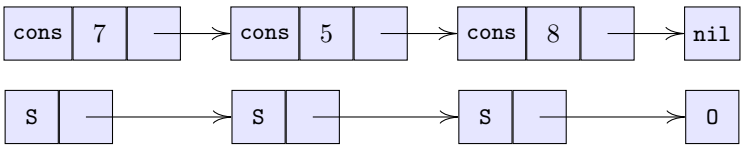
```
Definition pred (n : nat) : nat :=
  match n with
  | O ⇒ O
  | S n′ ⇒ n′
  end.
```

```
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 non-dependent version

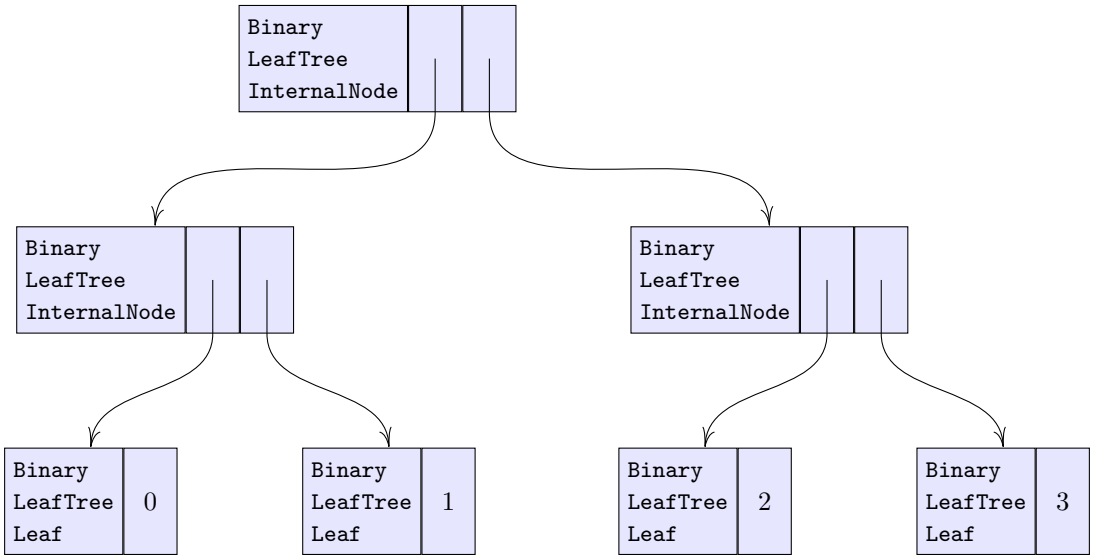
As there are many ways to represent natural numbers (and not only the unary system that was considered above). Depending on which representation we chose, operations on the numbers (such as increment, decrement, sum, converting into other representations) will have different efficiency, and in each case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

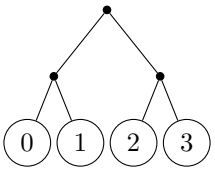
For the following, we will denote by `A` some data type (in examples in will be `nat`). A "complete binary leaf tree" of depth d has 2^d leaves where it stores values of type `A` ("complete binary" meaning that each internal node has exactly 2 children, "leaf" meaning that data is only stores in the leaves).

```
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:

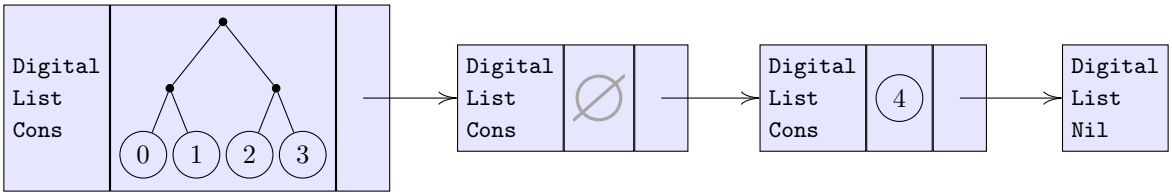


We know that for each natural number n there are $d \in \mathbb{N}$ and $n_{d-1}, n_{d-2}, ..., n_0, n_1 \in \{0, 1\}$ such that $n = n_{d-1}2^{d-1} + n_{d-2}2^{d-2} + ... + n_1 2^1 + n_0 2^0$. The function `to_digits` : $\forall$(`d n` : `nat`), `list nat` gives the list of length `d` consisting of 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 `of_digits` : $\forall$(`d` : `nat`)(`nl` : `list nat`), `nat`: for example, `of_digits` 8 [0; 1; 0; 0; 1; 0; 0; 1] $\rightsquigarrow$ 73.

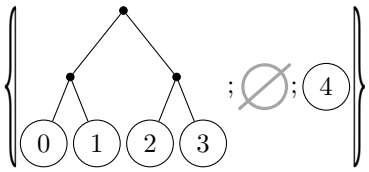
A "digital list" of depth d is a list of length d of which the k'th element is a complete binary leaf tree of depth k if $n_k = 1$ and nothing otherwise.

```
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:



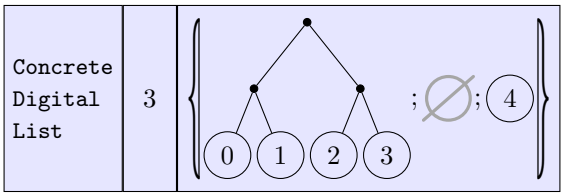
Or, to simplify the drawing, the following notation will be used form now on:



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 corresponds to the digital list given above:

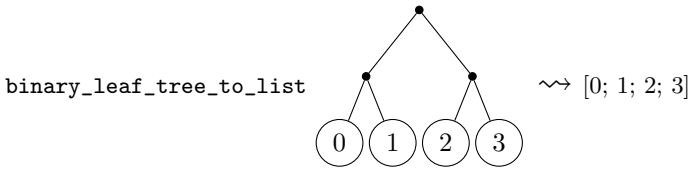


Next we will consider operations that we can do with these types. We will first define each operation on binary leaf trees, then lift it to digital lists (always having a "depth" argument, which corresponds to the depth of the digital list passed, for consistency and for reasons outlined later, even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

3.1 Interpretation

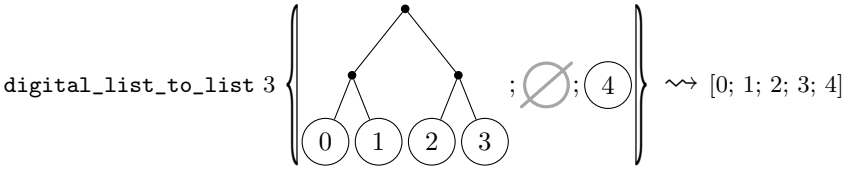
The first operation will be conversion to a simple list, which at the same times gives an "interpretation" of the data structures, allowing us to establish a convection between the operations on them and ones on the regular lists, which means proving the correctness of these operations.

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:



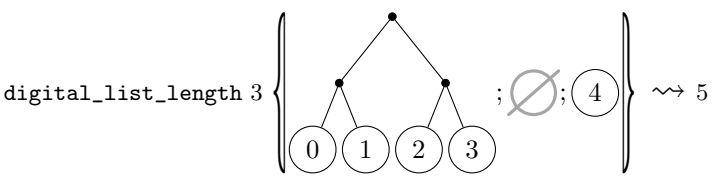

```
Fixpoint digital_list_to_list {A} d (dl : digital_list A) :=
  match dl with
  | DigitalListNil ⇒ []
  | DigitalListCons o dl' ⇒
    match o with
    | None ⇒ []
    | Some blt ⇒ binary_leaf_tree_to_list blt
    end ++ digital_list_to_list (pred d) dl'
  end.
```

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.
```

3.2 Length

The length (the number of data elements stored) of a complete binary leaf tree of depth d is, as remarked earlier, 2^d. So to get the length of a digital list, we just sum the lengths of the binary leaf trees, at positions where they are present. For example:



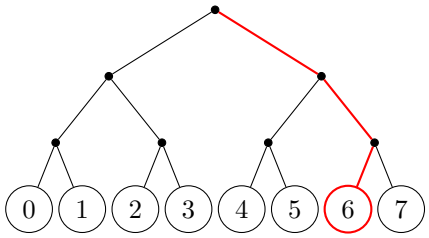
```
Fixpoint digital_list_length {A} d (dl : digital_list A) :=
  match dl with
  | DigitalListNil ⇒ 0
  | DigitalListCons o dl' ⇒ (if o then Nat.pow 2 (pred d) else 0) + digital_list_length (pred d) dl'
  end.

Definition concrete_digital_list_length {A} (cdl : concrete_digital_list A) :=
  let '(ConcreteDigitalList d dl) := cdl in digital_list_length d dl.
```

3.3 Getting an element by its index

The operation of getting the `i`th element is a bit more involved. In all cases, we first convert `i` to list of its binary digits, called `il` (it has the same length as the depth of the data structure we work with).

Then all we need to do for a complete binary leaf tree is to descent it, starting from the root, taking the left branch if the digit in current position is 0 and the right one if the digit in current position is 1. For example, suppose we want to get the 6'th element of the following tree; for that we first convert it to digits (`to_digits` 3 6 ⇝ [1; 1; 0], where 3 is the depth of the tree), and then visit the tree as described (the path is highlighted in red):

`complete_binary_leaf_tree_nth` 3 [1; 1; 0]

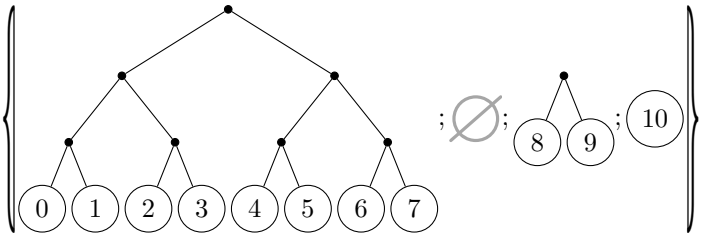


⇝ 6

In this case we suppose that all arguments are valid, but we still have to return option, as we need to return something in all cases.

```
Fixpoint complete_binary_leaf_tree_nth {A} d il (blt : binary_leaf_tree A) : option A :=
  match il, blt with
  | [], BinaryLeafTreeLeaf x ⇒ Some x
  | 0 :: il', BinaryLeafTreeInternalNode blt'1 _ ⇒ complete_binary_leaf_tree_nth (pred d) il' blt'1
  | 1 :: il', BinaryLeafTreeInternalNode _ blt'2 ⇒ complete_binary_leaf_tree_nth (pred d) il' blt'2
  | _, _ ⇒ None
  end.
```

Now, consider the following digital list:



To see how to extract the `i`th element, we need to know in which it is stored at which index. Let us see on this example what we get:

Element index	Tree index	Index in tree
0000_2	0	000_2
0001_2	0	001_2
0010_2	0	010_2
0011_2	0	011_2
0100_2	0	100_2
0101_2	0	101_2
0110_2	0	110_2
0111_2	0	111_2
1000_2	2	0_2
1001_2	2	1_2
1010_2	3	ε_2
1011_2	← **Count of elements in whole digital list**	

The crucial observation here is that the tree index is the index of the first digit in `i` that is less than the corresponding digit of `n`, and the index in the tree is just the rest of digits of `i`.

So in the case $i_{d-1} = n_{d-1} = 0$ (that is, `il` starts with 0 and there is no tree in this position) and in the case $i_{d-1} = n_{d-1} = 1$ (that is, `il` starts with 1 and there is a tree in this position), we recurse to our tail. Otherwise, in the case $i_{d-1} = 0$ and $n_{d-1} = 1$ (that is, `il` starts with 0 and there is a tree in this position), we ask this tree for its element at position defined by `il'`. The case $i_{d-1} = 1$ and $n_{d-1} = 0$ is impossible on valid input.

```
Fixpoint digital_list_nth_inner {A} d il (dl : digital_list A) : option A :=
  match il, dl with
  | 0 :: il', DigitalListCons None dl' ⇒ digital_list_nth_inner (pred d) il' dl'
  | 1 :: il', DigitalListCons (Some _) dl' ⇒ digital_list_nth_inner (pred d) il' dl'
  | 0 :: il', DigitalListCons (Some blt) dl' ⇒ complete_binary_leaf_tree_nth (pred d) il' blt
  | _, _ ⇒ None
  end.
```

All that is left now is to check whether `i` is in bounds (to get a functions that works on and `i`) and to convert it into digits.

```
Definition digital_list_nth {A} d i (dl : digital_list A) : option A :=
  if Nat.ltb i (digital_list_length d dl)
  then digital_list_nth_inner d (to_digits d i) dl
  else None.

Definition concrete_digital_list_nth {A} i (cdl : concrete_digital_list A) : option A :=
  let '(ConcreteDigitalList d dl) := cdl in digital_list_nth d i dl.
```

3.4 Updating an element by its index

The operation of replacing the `i`th element by another `x : A` is completely analogous to getting the `i`th element, the only difference is that we reconstruct back the data structures with the element updated.

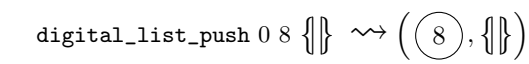
```
Fixpoint complete_binary_leaf_tree_update {A} d il x (blt : binary_leaf_tree A) :
  option (binary_leaf_tree A) :=
  match il, blt with
  | [], _ ⇒ Some (BinaryLeafTreeLeaf x)
  | 0 :: il', BinaryLeafTreeInternalNode blt'1 blt'2 ⇒
    option_map
      (fun blt'1_0 ⇒ BinaryLeafTreeInternalNode blt'1_0 blt'2)
      (complete_binary_leaf_tree_update (pred d) il' x blt'1)
  | 1 :: il', BinaryLeafTreeInternalNode blt'1 blt'2 ⇒
    option_map
      (fun blt'2_0 ⇒ BinaryLeafTreeInternalNode blt'1 blt'2_0)
      (complete_binary_leaf_tree_update (pred d) il' x blt'2)
  | _, _ ⇒ None
  end.

Fixpoint digital_list_update_inner {A} d il x (dl : digital_list A) : option (digital_list A) :=
  match il, dl with
  | 0 :: il', DigitalListCons (None as o) dl'
  | 1 :: il', DigitalListCons (Some _ as o) dl' ⇒
    option_map
      (DigitalListCons o)
      (digital_list_update_inner (pred d) il' x dl')
  | 0 :: il', DigitalListCons (Some blt) dl' ⇒
    Some (DigitalListCons (complete_binary_leaf_tree_update (pred d) il' x blt) dl')
  | _, _ ⇒ None
  end.

Definition digital_list_update {A} d i x (dl : digital_list A) : option (digital_list A) :=
  if Nat.ltb i (digital_list_length d dl)
  then digital_list_update_inner d (to_digits d i) x dl
  else None.
```

```
Definition concrete_digital_list_update {A} i x (cdl : concrete_digital_list A) :
  option (concrete_digital_list A) :=
  let '(ConcreteDigitalList d dl) := cdl in
    option_map (ConcreteDigitalList d) (digital_list_update d i x dl).
```

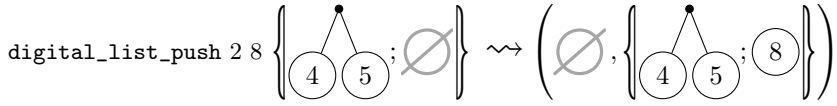
3.5 Adding an element to the end

Adding an element `x : A` to the end of a digital list of depth `d` can potentially produce a carry (a complete binary leaf tree of depth `d`) and always gives a digital list of the same depth `d`.

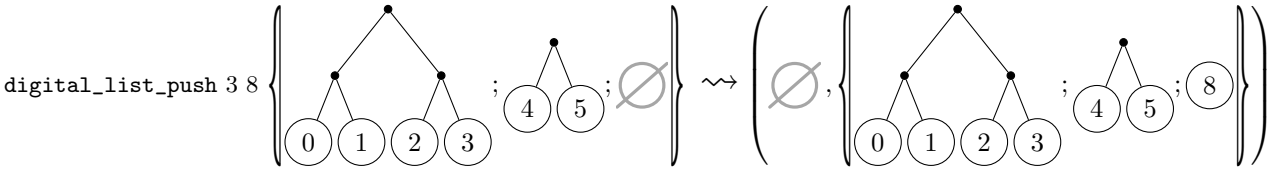
If the initial digital list is of depth 0, we output a carry with the element x. For example:



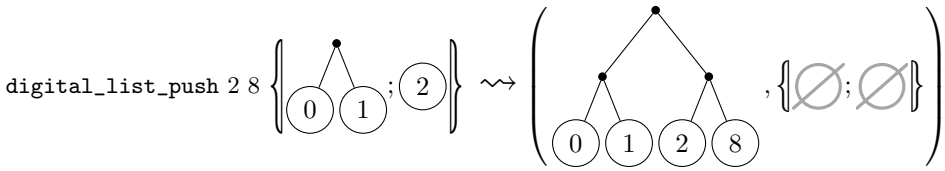
Else, we recurse. Suppose that we did not get a carry. In this case, we just add our digit (if it exists) to the result of the recursive call. For example, if the recursive call looked like this:



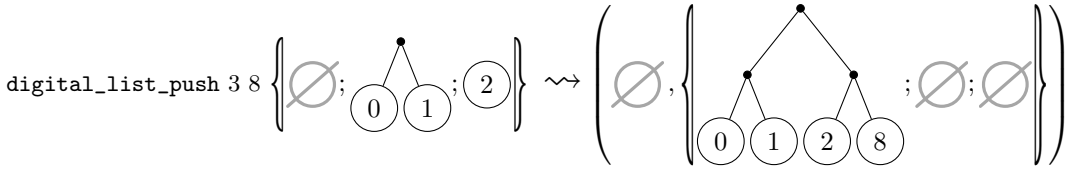
Then the parent execution can be the following:



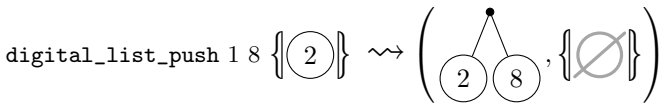
Else there was a carry. And again, the are two cases, depending on whether or not the head of the digital list contains a tree. If it does not, we just add the carry that we got in this place. For example, if the recursive call looked like this:



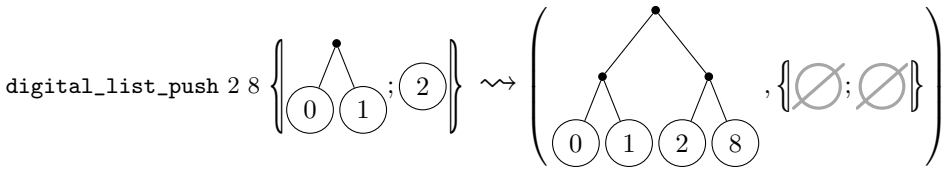
Then the parent execution can be the following:



But if the head of the list contains a tree, then we create a tree from it and the carry and return it again as a carry. For example, if the recursive call looked like this:



Then the parent execution can be the following:



We can remark an analogy with binary adders:

First operand	Second operand	Carry	Result
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

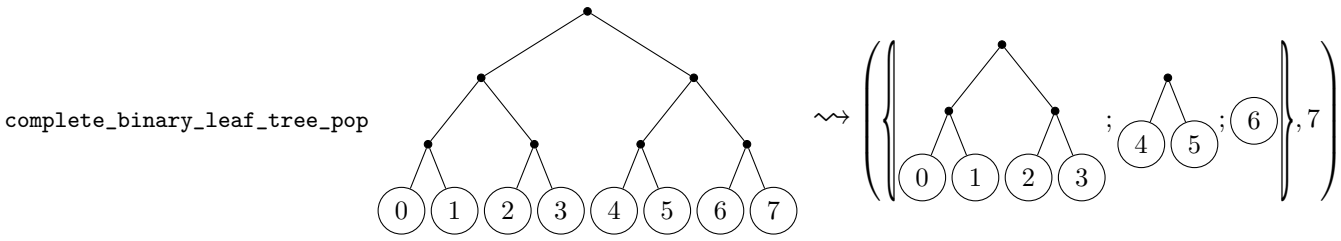
```
Fixpoint digital_list_push {A} d x (dl : digital_list A) : option (binary_leaf_tree A) * (digital_list A) :=
  match dl with
  | DigitalListNil ⇒ (Some (BinaryLeafTreeLeaf x), DigitalListNil)
  | DigitalListCons o dl′ ⇒
    match digital_list_push (pred d) x dl′ with
    | (None, dl′0) ⇒ (None, DigitalListCons o dl′0)
    | (Some blt0, dl′0) ⇒
      match o with
      | None ⇒ (None, DigitalListCons (Some blt0) dl′0)
      | Some blt ⇒ (Some (BinaryLeafTreeInternalNode blt blt0), DigitalListCons None dl′0)
      end
    end
  end.
```

To get a concrete digital list from an optional carry and the resulting digital list, we add the carry to the head of the digital list if it exists (and get a digital list of depth one more than the original), if not, then just return the digital list from the operation.

```
Definition concrete_digital_list_push {A} x (cdl : concrete_digital_list A) : concrete_digital_list A :=
  let ′(ConcreteDigitalList d dl) := cdl in
    match digital_list_push d x dl with
    | (None, dl0) ⇒ ConcreteDigitalList d dl0
    | (Some blt0, dl0) ⇒ ConcreteDigitalList (S d) (DigitalListCons (Some blt0) dl0)
    end.
```

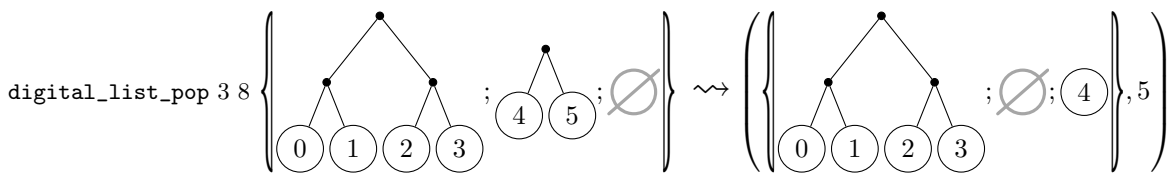
3.6 Removing an element from the end

To remove the last element from a complete binary leaf tree, we split it into a digital list and this element. For example:

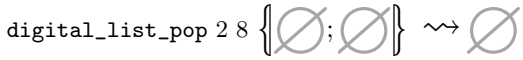



```
Fixpoint complete_binary_leaf_tree_pop {A} d (blt : binary_leaf_tree A) : digital_list A * A :=
  match blt with
  | BinaryLeafTreeLeaf x ⇒ (DigitalListNil, x)
  | BinaryLeafTreeInternalNode blt′1 blt′2 ⇒
    let ′(dl′, x) := complete_binary_leaf_tree_pop (pred d) blt′2 in
      (DigitalListCons (Some blt′1) dl′, x)
  end.
```

To remove the last element of a digital list, we just recurse until we are at the last existing tree, from which we remove the element from the end, and then reconstruct back the digital list, leaving unchanged all of the previous trees. For example:



The only case when we return ∅ is when the original digital list does not contain trees. For example:



We do not try to maintain the minimum possible depth, so the head of the digital list can become empty.

```
Fixpoint digital_list_pop {A} d (dl : digital_list A) : option (digital_list A * A) :=
  match dl with
  | DigitalListNil ⇒ None
  | DigitalListCons o dl′ ⇒
    match digital_list_pop (pred d) dl′ with
    | None ⇒
      option_map
        (fun blt ⇒
          let ′(dl′0, x) := complete_binary_leaf_tree_pop (pred d) blt in
            (DigitalListCons None dl′0, x)
        )
        o
    | Some (dl′0, x) ⇒ Some (DigitalListCons o dl′0, x)
    end
  end.

Definition concrete_digital_list_pop {A} (cdl : concrete_digital_list A) :
  option (concrete_digital_list A * A) :=
  let ′(ConcreteDigitalList d dl) := cdl in
    option_map
      (fun ′(dl0, x) ⇒ (ConcreteDigitalList d dl0, x))
      (digital_list_pop d dl).
```

References

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