Experience Report: Formalizing the DFUDS representation in Coq

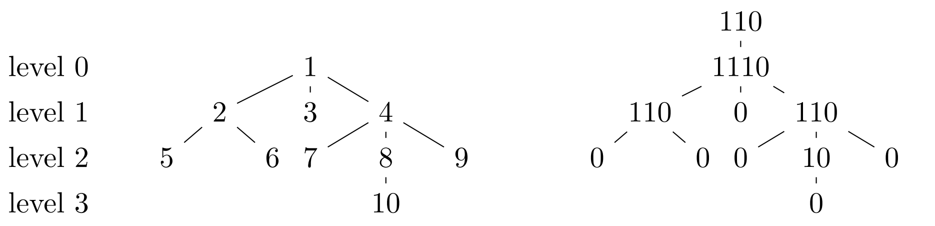
Xuanrui (Ray) Qi

May 11, 2019

1 What is DFUDS?

DFUDS, or *depth-first unary degree sequence*, is a way to represent ordinal trees (i.e., trees where the order of nodes matter) succinctly. The structure of a tree of n nodes can be represented in $2n + 2$, i.e., $O(n)$ bits maximum. This is one of the most compact representations of ordinal trees besides the similarly compact LOUDS (*level-order unary degree sequence*) and BP (*balanced parentheses*) representations, the former of which has been formalized in our previous work [1]).

On a high level, DFUDS is very similar to LOUDS, except that DFUDS uses another traversal order: LOUDS is based on level-order, i.e. breadth-first traversal, while DFUDS is based on pre-order traversal. Each node is represented succinctly as a few bits in the resulting bit array. Our algorithm is again based on [2].



(a) A sample tree (b) Its DFUDS vector encoding

Figure 1: DFUDS encoding of a sample unlabeled tree

One can see that DFUDS and LOUDS work very similarly on a high level: in fact, the node description function used in both encodings are identical, and only the "prefix" is different. The substantial difference between the two representations is thus in the traversal order.

In this brief work, we focused on formalizing just one DFUDS operation, $\textbf{child}(B, v) = \textbf{succ}_{\mathbf{0}}(B, v) - v$, where B is the DFUDS encoding of a tree and v is the position of a node in the encoding.

2 Take 1: direct implementation

LOUDS use level-order traversal, which is tricky in functional programming, but DFUDS is just a structural traversal over the tree, which we like. It is thus very easy to just implement DFUDS directly:

```
Definition describe_node (t : tree A) :=
  match t with
  | Node _ ch => rcons (nseq (size ch) true) false
  end.

Fixpoint DFUDS' (t : tree A) :=
  match t with
  | Node _ ch => describe_node t ++ flatten [seq (DFUDS' t') | t' <- ch]
  end.

Definition DFUDS t := [:: true; true; false] ++ DFUDS' t.
```

This does not look too bad at all! However, it was a nightmare to prove using this directly implementation. We will explain the reason very soon.

Defining position and the **children**(B, v) operation was not too hard, either:

```
Definition pred0 := pred false.
Definition succ0 := succ false.

Definition DFUDS_children B v := succ0 B v - v.

Definition DFUDS_bits (t : tree A) := (number_of_nodes t).*2 - 1.

Fixpoint DFUDS_position (t : tree A) (p : seq nat) := 3 +
  match t, p with
  | Node _ ch, [::] => 1
  | Node _ ch, h :: p' =>
    (size ch).+1 +
    sumn [seq (DFUDS_bits t') | t' <- take h ch] +
    DFUDS_position (nth t ch h) p'
  end.
```

They do not look complicated, but proving anything about paths was a nightmare. The main problem is showing the relationship between valid paths in the tree and the position in the DFUDS array. When we are doing a level-order traversal, we descend one level at each step, corresponding perfectly to one element in the "path", which is a array of indices of nodes at each level, so this was not a problem at all.

However, in DFUDS, this became a huge problem. Traversal does not really respect the path; when we descend into a path, we must first fully traverse all preceding nodes. This makes it very painful. Therefore, even after profuse number of attempts, I could not find a way to prove our main theorem in this direct implementation.

3 Take 2: traversal with respect to a path

We learn from the lessons of our previous work on LOUDS [1], and first formalize tree traversal in the abstract. Moreover, we describe an operation "traversal up to a path", which collects all nodes traversed when we follow a path. The standard pre-order traversal is easy, and so is the proof about its size:

```
Fixpoint po_traversal_st (t : tree A) :=
  match t with
  | Node _ ch => (f t) :: flatten [seq (po_traversal_st t') | t' <- ch]
  end.

Lemma po_traversal_st_size t : size (po_traversal_st t) = number_of_nodes t.
```

It is easy to note that DFUDS is just specializing this function, flattening, and adding a prefix:

```
Definition DFUDS_st (t : tree A) := po_traversal_st describe_node t.

Definition DFUDS (t : tree A) :=
  [:: true; true; false] ++ flatten (DFUDS_st t).
```

That, however, is just generalizing our DFUDS operation a bit. What should we do about paths? The answer is that we can define another function that corresponds to "traversal up to a path".

```
Fixpoint po_traversal_lt (t : tree A) (p : seq nat) :=
  let cl := children_of_node t in
  match p with
  | [::] => [::]
  | n :: p' =>
    f t ::
      flatten [seq (po_traversal_st f t') | t' <- take n cl] ++
      po_traversal_lt (nth t cl n) p'
  end.
```

We collect all preceding nodes at each level and traverse them in full, and then traverse along the path. We can then show that it is indeed correct by showing it collects all preceding nodes in pre-order. But first, we need a helper function defining the position of a node in the pre-order traversal:

```
Fixpoint po_position (t : tree A) p :=
  let cl := children_of_node t in
  match p with
  | [::] => 0
  | n :: p' =>
    1 + sumn [ seq (number_of_nodes t') | t' <- take n cl ] +
    po_position (nth t cl n) p'
  end.

Lemma po_traversal_lt_sizeE {B} (f : tree A -> B) (t : tree A) p :
  valid_position t p -> size (po_traversal_lt f t p) = po_position t p.
```

Then, we can express our desired correctness lemma. However, in the short time leading towards this project delivery, I was not able to prove the lemma (the original LOUDS formalization took about a year!). It seems that we need to find a new proof technique for this, rather than just induction on paths and trees:

```
Lemma po_traversal_ltE {B} (f : tree A -> B) (t : tree A) p :
  valid_position t p -> po_traversal_lt f t p = take (po_position t p)
                                                       (po_traversal_st f t).
```

We also prove the “mapping” properties of pre-order traversal, as they will turn out to be useful since we often combine traversal with maps:

```
Lemma po_traversal_st_map {B : Type} (f : tree A -> B) (t : tree A) :
  po_traversal_st f t = map f (po_node_list t).

Lemma map_po_traversal_st {B T} (f : tree A -> B) (g : B -> T) :
  map g \o [eta po_traversal_st f] =1 po_traversal_st (g \o f).
```

We also have similar lemmas for `po_traversal_st`.

Then, the position of a nodes in DFUDS becomes just an instance of traversal up to a path (and then summing). For convenience we also define DFUDS traversal up to a path, and prove that this traversal corresponds exactly to `DFUDS_position`:

```
Definition DFUDS_position t p :=
  let desc_l := po_traversal_lt description_length t p in
  4 + sumn desc_l.

Definition DFUDS_lt (t : tree A) p := po_traversal_lt describe_node t p.

Lemma DFUDS_positionE t p : valid_position t p ->
  DFUDS_position t p = 4 + size (flatten (DFUDS_lt t p)).
```

The high-level definition of `DFUDS_children` and `DFUDS_childrenE` are the same as before, but the underlying internal implementation is very different. The proof must proceed differently. In this short period of time, I was not able to complete the proof of `DFUDS_childrenE` either, but I believe using the correspondences between traversal and traversal up-to a path, and more trial and error, we can complete the proof. As far as I understand, most difficulties are purely engineering — the expressions are very complicated. However, we were able to prove some interesting small lemmas about DFUDS, such as the size of DFUDS:

```
Lemma DFUDS_sizeE t : size (DFUDS t) = (number_of_nodes t).*2.+2.
```

I expect that if given a few months time `DFUDS_children` will not be a difficulty, but in such a short span of time, we simply do not have enough time for enough trial-and-error.

References

[1] Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, and Kazunari Tanaka. Proving tree algorithms for succinct data structures, 2019. arXiv:1608.02792 [cs.IT].

[2] Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, New York, NY, USA, 1st edition, 2016.