

Doubly Linked List in a Purely Functional Programming Language

Asked 15 years ago Modified 13 years, 9 months ago Viewed 10k times



52



How does one go about doing doubly linked lists in a pure functional language? That is, something like Haskell where you're not in a Monad so you don't have mutation. Is it possible? (Singly linked list is obviously pretty easy).

data-structures

haskell

functional-programming

linked-list

Share

Improve this question

Follow

edited Mar 17, 2011 at 16:55



Aryabhata

asked Dec 4, 2009 at 0:59



Claudiu

229k ● 173 ● 503 ● 697

- 3 Out of curiosity, what exactly do you need this for?
– [Sasha Chedygov](#) Dec 4, 2009 at 3:18

4 Answers

Sorted by:

Highest score (default)





67



In a pure functional language, a doubly-linked list is not that interesting. The idea of a doubly linked list is to be able to grab a node and go in either direction, or to be able to splice into the middle of a list. In a pure functional language you probably are better off with one of these two data structures:

- A singly linked list with a pointer in the middle, from which you can go either left or right (a variant of Huet's "Zipper")
- A finger tree, which is a mind-blowing data structure invented by Ralf Hinze and Ross Paterson.

I'm a big fan of the zipper; it's useful in a lot of situations.

Share Improve this answer

answered Dec 4, 2009 at 3:15

Follow



[Norman Ramsey](#)

202k ● 62 ● 371 ● 541

I definitely agree that those are far better options. =)

– [Edward Kmett](#) Dec 7, 2009 at 21:52

Finger trees... interesting... :) – [sholsapp](#) Aug 22, 2010 at 4:51

2 Data.Sequence is the impl. of the finger tree, easy to use :)
– [wuxb](#) Jul 18, 2011 at 17:43

Can you elaborate on the variant of "Zipper". Any examples?
Thanks! – [DavidY](#) Jan 6, 2022 at 3:02

For those who are wondering like me, this article can really help: blog.ezyang.com/2010/04/you-could-have-invented-zippers – [DavidY](#) Jan 7, 2022 at 3:44



There are a number of approaches.

23



If you don't want to mutate the doubly-linked list once you have constructed it you can just 'tie the knot' by relying on laziness.



http://www.haskell.org/haskellwiki/Tying_the_Knot



If you want a mutable doubly-linked list you need to fake references somehow -- or use real ones -- a la the trick proposed by Oleg Kiselyov and implemented here:

<http://hackage.haskell.org/packages/archive/liboleg/2009.9.1/doc/html/Data-FDList.html>

Interestingly, note that the former relies fundamentally upon laziness to succeed. You ultimately need mutation or laziness to tie the knot.

answered Dec 4, 2009 at 1:11

[Edward Kmett](#)

30k ● 7 ● 87 ● 107



13



I would reiterate musicfan's question: "what exactly do you need this for?" As Norman Ramsey notes: if you need multi-directional traversal, then zippers are easier; if you need fast splicing, finger trees work well.

But, just to see how it looks...



```
import Control.Arrow
import Data.List
```

```
data LNode a = LNode { here :: a, prev :: LList a, next :: LList a }
type LList a = Maybe (LNode a)
```

```
toList :: LList a -> [a]
toList = unfoldr $ fmap $ here &&& next
```

```
fromList :: [a] -> LList a
fromList l = head nodes where
    nodes = scanr ((.) Just . uncurry LNode) Nothing $
```

```
append :: LList a -> LList a -> LList a
append = join Nothing where
    join k (Just a) b = a' where
        a' = Just $ a { prev = k, next = join a' (next a) }
    join k _ (Just b) = b' where
        b' = Just $ b { prev = k, next = join b' (next b) }
    join _ _ _ = Nothing
```



2



In OCaml, for circular simply linked list you can always do something like that:

```
type t = { a : t Lazy.t }

let cycle n =
  let rec start = {a = lazy (aux n) }
  and aux = function
    | 0 -> start
    | n -> { a = lazy (aux (n-1))}
  in start
```

For doubly linked lists, I imagine it's possible to do something similar. But you have to rely on laziness and on records being friendly structures when it comes to typing. Quick and dirty cyclic doubly linked list:

```
type 'a t = { data : 'a; before : 'a t Lazy.t; after : 'a t Lazy.t }

let of_list l =
  match l with [] -> assert false | hd::tl ->
  let rec start = { data = hd; before = last; after =
  and couple = lazy (aux (lazy start) hd)
  and next = lazy (Lazy.force (fst (Lazy.force couple
  and last = lazy (Lazy.force (snd (Lazy.force couple
  and aux before = function
    | [] -> (lazy start), before
    | hd::tl -> let rec current = lazy { data = hd; be
  after }
  and couple = lazy (aux current tl)
  and after = lazy (Lazy.force (fst (
  and last = lazy (Lazy.force (snd (L
```

[in](#) start

current, last

Share Improve this answer

[edited Sep 9, 2010 at 21:15](#)

Follow

answered Sep 9, 2010 at 17:51



[Guillaume Yziquel](#)

21 ● 2
