List Church encoding using a class in Haskell

Matteo Bertorotta

June 2024

In this document, I will review my implementation of a Church encoding of lists. I have been able to implement all functions discussed in the paper such that they work for the encoded lists, albeit with limitations. I will cover the difficulties I encountered and the method I used to tackle these. Reading through this explanation alongside the *list.hs* file is advised, as I go over the implemented structures, functions, classes, and instances in the order in which they appear in *list.hs*, but I do not cover their implementation in detail here. The comments in the code file also go hand in hand with the explanations in this document.

The base functor, *List_* and general Church encoding, *ListCh*, of the *List* structure can be defined without any problems, simply by following the paper's guidelines. I also did not encounter any difficulties when I defined the producers and consumers covered in the paper, i.e. *between*, *sum*, and *maximum*.

The difficulties arise when implementing the transformers. The *filter* function is a nice starting example. We need to implement a function that encodes one step of the filtering process, removing the current value x based on the predicate p. This is no problem for the *Tree_* from the paper, as this *Tree_* only stores its values at the leaf nodes. The one-step *filter* function thus does not need to do anything for branches, as these contain no values to filter. It only has to potentially change *Leaf_ x* nodes to *Empty_* nodes.
The *List* structure is different in that its "branches", *Two a (List a)* for *List* and *Two_ a b* for *List_*, do contain values. The one-step *filter* function must be able to remove the value x of type a from the structure, and only keep the remaining *xs* of type b. It must, however, return a *Tree_ a b*, which cannot **generally** be done with just a value of type b, as the only constructor of *Tree_ a b* with a value of type b is *Two_ a b*, which requires a value of type a. We thus need some way to give Haskell the guarantee that this can be done; that *xs* of type b can form a *List_ a b* on its own.

The one-step *append* and *reverse* functions have similar problems. We cannot **generally** create a *List_ a b* from two values of type b, which is required for the encoded *append* function. This can be done for the *Tree_* structure from the paper, as it has a constructor *Fork_ b b*, allowing exactly this. The problem with one-step *reverse* is that it requires putting a value of type a "at the end" of the structure of type b, thereby creating a structure of type *List_ a b*, which again cannot **generally** be done. The *Tree_* structure from the paper does not have this problem, as there reversing simply means swapping the two values of type b in *Fork_ b b*.

We have found that *filter*, *append*, and *reverse* all have similar problems; they require a guarantee that they can create the required structure of type *List_ a b*, but this guarantee generally cannot be given. This brings us to my solution, namely a class. With a class, we can give the guarantee that there are functions that can perform the aforementioned tasks. I created the *Usable a b* class, which requires its instances to implement *oneStepFilter*, *oneStepAppend*, and *oneStepReverse*, thereby giving the desired guarantees.

There is one interesting thing we need to do. If we just make the one-step functions using the functions of the *Usable* class, and use those in the Church encoded functions, we will for example get the following error at the *appendCh* function: *"No instance for (Usable a b) arising from a use of 'oneStepAppend'. Possible fix: add (Usable a b) to the context of a type expected by the context:* $forall\ b. (List\_\ a\ b \to b) \to b$*"*. This means that *oneStepAppend* cannot generally be sure that types a and b are actually *Usable* together; i.e. that there is a relevant instance *Usable a b*. We can give this guarantee in the Church encoding of *List*: *newtype* $ListCh\ a = ListCh\ (forall\ b. Usable\ a\ b\ \Rightarrow (List\_\ a\ b \to b) \to b)$.

Now that we have the class, we need to make the relevant instances. Because we have functions s and mx of type $List\_\ Int\ Int \to Int$ which we use in *sumCh* and *maximumCh*, we are required to make the instance *Usable Int Int*, which can be generalized to *Usable b b*. This is a very straightforward implementation.
We also have the function *in'* of type $List\_\ a\ (List\ a) \to List\ a$ that gets used in the function *fromCh*,

wherefore we require an instance *Usable a (List a)*. We would want its functions to not involve any recursion over the structures, as they should only help to perform one step of the relevant recursive function. However, the implementation of *oneStepReverse* requires knowing how to put the value of type a at the end of the structure of type b. In the case of *Usable a (List a)*, this means knowing how to put x of type a at the end of xs of type *List a*. This is impossible to achieve without recursion, as we need to go through all of xs to reach its end. *oneStepAppend* has a similar problem. However, this is less of a problem than it seems. Due to the functioning of Church encodings, these specific implementations of *oneStepReverse* and *oneStepAppend* will only be used if we use the *fromCh* function. Because I added Haskell rules to replace occurrences of *toCh (fromCh x)* with x, this should only happen if we actually do output a *List*. If our final value is an *Int*, we will for example use the *O(1)* implementations of *Usable Int Int* instead.
Lastly, I made an instance for *Usable a (Seq a)*, using the *Sequence* structure which is basically a Haskell *List* that has *O(1)* complexity to append a value at the end of a *Sequence*, and *O(log(min(n1,n2)))* complexity to append a *Sequence* at the end of another *Sequence*. This therefore makes it possible to implement *oneStepReverse* without any recursion and *oneStepAppend* more efficiently than the *Usable a (List a)* implementation. This allows us to implement a *toSeq* function making it possible to implement a pipeline and print the final *Seq a* more efficiently than would be the case with *List a*. *Sequence* is implemented with Finger Trees. One could potentially directly make a Church encoding over Finger Trees. I did not try this, as Finger Trees were too complex for me to understand within the time period of the project.

Finally, I made an implementation of the transformation function *map*. *map* does not suffer from any of the problems of the other transformation functions, and can be defined without any problems. However, now that we made our implementations with the *Usable* constraints, *map* has a severe limitation. It is unclear to Haskell that the type of the value that will be mapped to will still be Usable with the value of type a. Therefore, we need to give Haskell this guarantee ourselves. The most useful thing I could think of was adding the constraint $forall\ b.\ Usable\ c\ b \Rightarrow Usable\ a\ b$. However, this poses a big limitation to the system. When the mapping changes the type of the values, this constraint is non-trivial and cannot be proven by Haskell alone. I, however, have no clue how to prove this implication for these non-trivial cases myself. This means that *map* can only be used in cases where the types do not change; e.g. with $(+1) :: Int \rightarrow Int$.