a $\rightsquigarrow$ b

Choosing a Flexible List Monad in Idris

Asher Frost

August 13, 2025

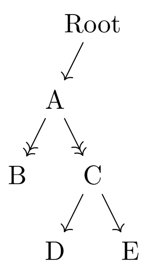
Abstract

1 Introduction

In both Haskell and Idris, Monads play a very important role in abstraction. They are central to `do` notation in both languages, which forms the basis of functional abstraction, without which programming with pure "state" would be difficult and unexpressive.

One of the simplest examples, and also one of the first given, is the list monad. This is because `>>=`, which for a list would have type, `List a -> (a -> List b) -> List b` is exactly the same as an equivalent `flatMap` type Citation. Because of this, it is easier to teach relationship between "restricted iterators", and functors, applicatives, monads, traversables, and the like.

Given this utility, and the fact that lists have a very central role



2 Free Choice

The choice library defines 3 different implementations of a list monad, all of which capture "global state". The first two of these are `LogicT`, based off the LinkCitation`LogicT` Haskell package. the second being `ListT` based off the LinkCitation`ListT` Haskell package.

Both of these packages capture the notion of "free" `Foldable`, with `LogicT` doing so through continutaions, and `ListT` doing so with monoidal structure.

However, here we also create a novel *tree* based approach to a list monad. Firstly, the definition of it is as follows:

```
data TreeT : forall k0, k1. (m : k0 -> k1) -> (a : k0) -> Type where
^^IMLeaf : (r : Lazy (List a)) -> TreeT m a
^^IMBranch : (c0 : (TreeT m a)) -> (c1 : m (TreeT m a)) -> TreeT m a
^^I
ChoiceT1 : forall k. (Type -> k) -> Type -> k
ChoiceT1 m a = m (TreeT {k0 = Type} m a)

data ChoiceT : forall k. (m : Type -> k) -> (a : Type) -> Type where
^^IMkChoiceT : ChoiceT1 m a -> ChoiceT m a
^^I
```

First, `TreeT` defines a binary tree like structure. Note that this is almost the exact same as the `StepT` from `ListT`, with `StepT` being