Choosing a Flexible List Monad in Idris

Asher Frost

July 20, 2025

Abstract

To do (??)

1 Introduction

Monads (combined with functors and applicatives) form the backbone of functional programming. While confusing to beginners, the ability to do generalized "iterator style programming" allows for abstractions and better code design. [1] They allow for the usage of distinct ideas, for instance, side effects and potential failure, in the same way.

This allows us to write code that is generic and understandable. Because of this, the natural extension to this was to allow *multiple* of these "wrappers" per value. This would allow for things such as "output with pure state", where, instead of having `State (IO a)`, where we need to manually deal with both levels of nesting (which breaks do notation), we can instead have `StateT IO a`, which neatly wraps the "inner" monad (`IO` in this case) with state.

This led to the creation of Monad Transformers, Monads that add "capibilites" to other Monads. Almost ever single monad has a transformer variant, `State : Type -> Type` becomes `StateT : (Type -> Type) -> Type -> Type`, for instance. Then, when we wish to get the "base" or "regular" version of a monad, we merely pass in the "identity" type, something that just returns the inner type unchanged.

Despite this, there is one very important type that is, at its face, a monad, but not a monad transformer, that being the list monad. The list monad (and applicative) should conceptually model the idea of a "line of computation". That is, when the code in **To do** (??) is executed, it takes each possible pair and computes the sum:

```
addM : List Nat -> List Nat -> List Nat
addM xs ys = do
        x : Nat <- xs
        y : Nat <- ys
        pure (x + y)
```

However, if we wanted to add to this the ability to print out the product of each pair as we were going through, we would be stuck, as we might try one of **To do** (??)

```
addA : List (IO Nat) -> List (IO Nat) -> List (IO Nat)
addA xs ys = do
x : IO Nat <- xs
y : IO Nat <- ys
let z : IO Nat = x * y
pure (x + y)
```

There are two main ways we might make a list transformer, those being below. **To do** (??)

2 Free Choice

The solution to all the above problems was to create a Choice Monad Transformer, closely based off the design of the `logict` package in Haskell and it's `LogicT` [2]. However, while this is based off of that package, there are some major differences. First, however, let us compare the regular List and the `LogicT` **To do** (??)

```
data List : Type -> Type where
        Nil : List a
        (::) : a -> List a -> List a

data LogicT : (Type -> Type) -> Type -> Type where
        MkLogicT :
                (forall r.
                        (a -> m r -> m r)
                        -> m r
                        -> m r
                )
                -> LogicT m a

unLogicT :
        LogicT m a
        -> (forall r.
                (a -> m r -> m r)
                -> m r
                -> m r
                )
```

[1]

While these two definitions might look unrelated at first, if we take the seemingly strange step of doing `foldr`, on a `List`, we get something of the form

[1] Ported to Idris

```
foldr {t = List} : (a -> r -> r) -> r -> List ea -> r
```

and flipping around the order of parameters we get

```
foldr {t = List} : List a -> (a -> r -> r) -> r -> r
```

which looks shockingly similar to

```
unLogicT :
        LogicT m a
        -> (forall r.
        (a -> m r -> m r)
        -> m r
        -> m r
        )
```

Indeed, the only difference are the presence of the type `m`. This works because we can then pass in, if `m = Identity`, `::` and `Nil` to recover a list:

```
unLogicT :
LogicT Identity a
-> (forall r.
(a -> Identity r -> Identity r)
-> Identity r
-> Identity r
)
```

Then, simplifying this by Identityt $\Rightarrow$ t and `Logic = LogicT Identity`, we get

```
unLogic :
Logic a -> (forall r. (a -> r -> r) -> r -> r)
```

Which, given universal introduction and elimination (that $\forall x.(A \to B) \rhd (\forall x.A) \to (\forall x.B)$ and that $\forall x.A \rhd A$ given that x does not occur in A), is equal to `foldr`. That is, `LogicT` provides a way to fold together computations.

2.1 The Cons of Folding

While this implementation is quite nice, it suffers from a couple problems

1. Its use of Continuation Passing Style (CPS) means that its much more complex to compile**To do (??)**
2. CPS makes the code harder to understand**To do (??)**
3. CPS makes the code harder to debug
4. It can lead to code that is hard to make monomorphic (r only appears on the left hand side)

Because of this, here, we propose `ChoiceT`, a List Monad Transformer that does not use CPS, is fully monomorphic, is intutive to understand, and an extension, rather than proxy, for regular Lists.

2.2 Choice as a List

The `ChoiceT` monad transformer is defined in terms of one other type, `MStep`, which is devised as follows:

```
data MStep : (Type -> Type) -> Type -> Type where
        MNil : MStep m a
        MCons : a -> m (MStep m a) -> MStep m a

data ChoiceT : (Type -> Type) -> Type -> Type where
        MkChoiceT : m (MStep m a) -> ChoiceT m a
```

[2]

In short, `ChoiceT` works by allowing for "delimited continuations in lists". That is, while in a regular list you can retrive values until you get to the end of the list (which allows you to have arbitrarly long list pattern matches) `MStep` wraps each succesive action in a monadic context, `m`.

Note also that because of how they are defined, `MStep` does *not* have any context on its first value. This is where `MkChoiceT` comes in, allowing us to "wrap" `MStep` itself in the monadic context, thereby creating a true list monad.

A Mathematical Symbols

Name	Symbol
`Functor`	פ
`Applicative`	ע
`Monad`	מ
`MonadTrans`	ט
`Foldable`[3]	כ
`Traversable`	ת
`Semigroup`	ס
`MStep`	נ
`ChoiceT`[4]	ב

B Omitted Proofs

Lemma 1. $\mathtt{פ}\alpha \Rightarrow \mathtt{פ}(\mathtt{נ}\alpha)$

Proof. By elimination we transform to $\vdash (f : \beta \to \gamma) \to (x : \mathtt{נ}\beta) \to (\mathtt{נ}\gamma)$. By introduction we have $(f : \beta \to \gamma), (x : \mathtt{נ}\beta) \vdash (\mathtt{נ}\gamma)$ We preform induction on x to

[2] The package itself creates a helper type alias, MList, and has MStep be kind polymorhpic. For simplicites sake, these are both changed to improve readability.

yield two cases, $\texttt{map}_0 : (f : \beta \to \gamma) \to ([] : \texttt{J}\alpha\beta) \to (\texttt{J}\alpha\gamma)$ and $\texttt{map}_0 : (f : \beta \to \gamma) \to ([] : \texttt{J}\alpha\beta) \to (\texttt{J}\alpha\gamma)$ □

Theorem 1.

Proof. □

References

[1] Mark P. Jones. "Functional Programming with Overloading and Higher-Order Polymorphism". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 97–136. ISBN: 3540594515.

[2] Oleg Kiselyov et al. "Backtracking, interleaving, and terminating monad transformers: (functional pearl)". In: *SIGPLAN Not.* 40.9 (Sept. 2005), pp. 192–203. ISSN: 0362-1340. DOI: 10.1145/1090189.1086390. URL: https://doi.org/10.1145/1090189.1086390.