Monad Factory: Type-Indexed Monads

Mark Snyder, Perry Alexander

The University of Kansas
Information and Telecommunication Technology Center
2335 Irving Hill Rd, Lawrence, KS 66045
{marks,alex}@ittc.ku.edu

Abstract. Monads provide a greatly useful capability to pure languages in simulating side-effects, but implementations such as the Monad Transformer Library [1] in Haskell prohibit reuse of those side-effects such as threading through two different states without some explicit work-around. Monad Factory provides a straightforward solution for *opening* the non-proper morphisms by indexing monads at both the type-level and term-level, allowing 'copies' of the monads to be created and simultaneously used within even the same monadic transformer stack. This expands monads' applicability and mitigates the amount of boilerplate code we need for monads to work together, and yet we use them nearly identically to non-indexed monads.

Key words: monads, Haskell, type-level programming

1 Introduction

Programming with monads in Haskell provides a rich set of tools to the pure functional programmer, but their homogeneous nature sometimes proves unsatisfactory. Due to functional dependencies in the Monad Transformer Library (`mtl`), an individual monad can only be used in one way in a fragment of code, such as using the `State` monad to store a particular state type. When a programmer wants to use a monad in multiple ways at once, some hack or work-around is necessary, such as using a record structure or carefully `lift`-ing through a specific monad transformer stack. While monad transformers allow us to combine the effects of different monads, we cannot directly use transformers to combine the effects of a particular monad multiple times. The problem grows as code from various sources begins to interact–when using existing code, a monad's usage might be "reserved" for some orthogonal use; should you edit that code, even if you can? Even when we give in and re-implement a common monad to differentiate between the uses, we must provide instances for a host of other common monads in defining the transformer, including providing an instance relating to the the copied monad–these details require more knowledge about monads than simply using them. We seek a more adaptable process for capturing monadic behavior, and expect some streamlining, re-usability, and a less error-prone process than some ad-hoc options that are commonly used. In particular, we expect to write fewer instances for transformer inter-operability.

This paper introduces *type-indexed monads*, which allow for multiple distinct instances of a monad to coexist, via explicit annotations. A simple type class links the term-level index and the type-level index together, allowing both type inference and evaluation to differentiate between instances of monads.

An arbitrary number of these indexed monads can coexist, making monadic programming more modular and more flexible. This 'factory' approach solves some of the problems associated with non-indexed monads, and avoids the need for early design decisions on monad usage.

This approach tries to work with the existing implementation and existing code based on `mtl`, rather than propose massive re-writing. Perhaps most interestingly, the core technique may prove useful for other cases where a functional dependency causes instance selection to be problematic.

This work provides the following contributions:

- Particular monadic side-effects (such as state maintenance) may be used multiple times simultaneously, without affecting each other's usage.
- Type-indexed monads are compatible with existing monadic code, so we do not have to prepare our existing code to accomodate type-indexed monads. They can be stacked together in a monad transformer stack, also with the monads found in the Monad Transformer Library provided with GHC [2].
- Libraries that use common monadic effects don't need a local copy of any monad along with all the requisite instances. We discuss why that libraries might resort to re-implementing monads, providing a mechanism for removing the code duplication in favor of a re-usable solution. This saves effort and mitigates the error-prone process of monad implementation.
- When we combine monad transformers, we need an `instance` describing how every pair of transformers can be lifted through each other; this quadratic number of required `instance`s is mitigated, as we provide instances that address the entire indexed family of monads. As long as type-indexed monads can define the needed semantics, there are no more `instance`s to create.
- The `State`, `Reader`, `Writer`, `RWS`, and `Error` monads are implemented as type-indexed monads (available on Hackage [3]).

We identify the concept of a type-indexed monad, then build upon the initial definitions of monads to guide the type-indexed versions of monads. Type-indexed monads do not attempt to provide a simplified interface to monads. Rather, the point is to make heterogeneous monad usage simpler and more convenient for a programmer who has already overcome the hurdle of understanding monads.

2 Problem

Monads are a mechanism often used for simulating effectful computation in a pure language like Haskell. They are pervasive in Haskell programs, yet the mechanism used to define them in the Monad Transformer Library—type classes and the related instances—has drawbacks.

Type classes indicate that any given instance type has the overloaded functions defined for it. Consider the monad `State s`, and the associated type class `MonadState s m`.

```
class (Monad m) ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()

newtype State s a = State {runState :: (s → (a,s))}

instance MonadState s (State s) where
  get   = State $ λs → (s, s)
  put s = State $ λ_ → ((), s)
```

`MonadState` defines a set of 'non-proper morphisms', `get` and `put` (as opposed to 'proper' morphisms like `return` and >>=). We have an instance of `MonadState` defined for the `State` monad, meaning that we can use `get` and `put` to construct monadic values. The problem arises that `s` must be determinable by `m`. For a given monadic computation for some `m`, `get` always gets a value of a particular type. We can't use `get` and `put` to store multiple `s`'s, and we can't use them to store values of different types. The functional dependency `m→s` at once allows us to use `State` at various types for `s` in separate places in our code, and restricts us from using `State` at various types in the *same* code.

Example—Design Decisions, Implications Suppose we are writing a library of monadic code. We want to provide some abstractions of operations that happen to maintain an environment of name-value pairs and pass around an integer as state; the actual purpose of the library is irrelevant. If we were free to be direct, we might actually use the `State` and `Reader` monads to manage our `[(String,v)]` and `Int` values for us. As we develop our library, we realize that we need some extra values to be kept in `State`; since we've already used `State`, we end up instead moving to a record as our state–indeed, many programmers would have started with this approach to avoid the tedious translation through the code. Now we can add to this record all the state we want, as long as we are designing the library and not just using it. An issue arises, in that `State` computations now have access to all fields of the record. Just as we would like to have separation in our processes' memory, we would like a guarantee of the separation of access to our different pieces of state. Some [4] use separate copies of the `State` monad to guarantee that separation. We will see that type-indexed monads are, at the type level, incapable of accessing or modifying each other's contents, and may be ideally suited to such work.

Our library will surely be exciting and popular, and we want to be savvy to our users' needs–they might want to use our library in their own monadic code that already uses `State` and `Reader`–so we create our own `MyLibM` monad (and transformer version) that provides the exact features of `State` and `Reader`, as well as the many `instance`s needed for it to be a monad and be 'stackable' (combined via the monad transformers) with the original `State` and `Reader`.

As our library gains in popularity, some users want to expand on the state stored in our record–but it is *closed*–just as we can't add constructors to a `data` definition except at the source, we can't add fields to a record except in its initial definition. Another user decides they want to use our library in a way we hadn't anticipated–they want to use it for a few different things at once, and they have to play a rousing game of 'count the `lift`s' in order to use the `MyLibM` monad twice or more in the same transformer stack. Another user wants to use their own hand-written monad with ours, and has to write a few more instances to make the two interact appropriately, even though they already have instances for `State` and `Reader`.

Through this entire process, we find problems whenever we want to expand a monad's usage or re-use it. We have some closed definitions, code copies of some monads, and some unhappy library users that had to create their own work-arounds for our code. What if we could use a monad for its effects in multiple ways without having to resort to records (and other similar approaches, such as `HList`s)? What if we didn't have to create our own `State` and `Reader` monads just to make sure the library users still had free use of it?

Type-indexed monads aleviate these problems by allowing us to use different type-level indexes to differentiate between intended uses of a monad. In our library example, this means that we can have several `State` constraints over our code to thread different states through our code at once (whether or not the states' types happen to match). Instead of using a record type, we could just use another index of the monad if we chose. Instead of copying the functionality of monads into our own `MyLibM` monad, we can create an index to use; whether we export that index or not also gives us control over how the library may be used. The indexed versions are distinct from the original definitions, and they may be used together. The library users can now use `State` to their own liking (and multiple times as well) without interfering with library code. They can even use the library code at different indexes in the same code, and it only requires different type-indexes, and no tedious `lift`ing, which can easily be abused. We also gain a guarantee of separation–we don't worry about one `get` affecting the wrong monad, as the types wouldn't line up–we gain this separation by the parametricity of the type-indexed monads' definitions. We can have one set of instances that work for any number of indexes, meaning if we can define our monad in terms of those offered in indexed style, we won't have any new instances to write, nor will users downstream. The number of instances is usually quadratic in the number of monads in the transformer stack, so this becomes more valuable as more monads are stacked.

3 Type-Indexed Monads

In order to provide a mechanism for type-indexed monads, we must account for differentiation between type-indexed monads at both the term-level for evaluation, and also at the type-level during typing. We consider some possible example uses to visualize type-indexed monad usage, and then provide details of a realiz-

able implementation. The concept itself arose as a realization that McBride [5] uses type-level representations of numbers to create indexes that simulate terms at the type level; the dependent feel of explicitly indexing monads seems like a plausible avenue for type-level programming.

We use the `State` monad as our running example, though of course others are also implemented. Instead of using a non-proper morphism and expecting Haskell to infer the index we are using, we explicitly label each usage. If we relied on an inference mechanism (with no index argument), that would preclude the opportunity of using multiple copies of a monad that happened to operate on the same type, e.g. using two different `State` monads to store different sets of available registers in a compiler. If we are creating type-indexed monads, they should closely resemble usage of the original non-indexed version. We add the explicit indexing parameter as a first parameter to all non-proper morphisms. By convention, we add an 'x' (or 'X') to all labels to differentiate them from the original monad definition rather than rely on name qualifications. This explicit indexing will allow us to use different type-indexes with `StateX` to store the same type of state.

As an introduction to the syntax and feel of type-indexed monads, consider a basic monadic successor function, and a similar function that increases two separate states, using `StateX` monads:

```
succM :: (MonadState Int m) ⇒ m ()
succM = do n ← get
           put $ n+1

succ2M :: (MonadStateX Index1 Int m, MonadStateX Index2 Int m) ⇒ m ()
succ2M = do x ← getx Index1
            y ← getx Index2
            putx Index1 $ x+1
            putx Index2 $ y+1
```

Instead of providing separate `get1` and `get2` functions, we parameterize the `get` function to operate over the index as well. This feels similar to the record-as-state approach mentioned in the introduction, except that we can leave previous uses of `State` untouched. Also, this approach is open to further indexed uses. If we were to try to use `lift`s in order to use `State` twice, we might specify that `(put 1 >> lift (put 5)) :: StateT Int (State Int) ()`. To express this via constraints, the type may be written as `(MonadState Int (t m), MonadState Int m, MonadTrans t) ⇒ t m ()`. It's possible, but this gets messier as we add more constraints: we are specifying the transformer stack in our type. We would like to separate this concern, especially if we want to combine the code with code that has a more constrained type. We could perhaps use abstractions to hide the lifting, but this is still one more step that we have to do, and that we can get wrong. Phantom typing [6] won't help us here: an expression such as `(put 5)` doesn't give enough information to know into which `Int`-state to put the `5`–either one would be plausible embedded in a do-expression, so there's no 'best' answer for the phantom to find or check for us.

Type-indexed monads clearly need to have distinct types. Haskell does not have dependent types, so our indexing must appear at the type level as well. Seeking openness, we express the type of a monadic computation by constraining it rather than constructing it. We define what characteristics a monadic computation must include, rather than directly defining it. The type of `succ2M` states that `m` is a monad exibiting the behavior of the `(MonadStateX Index1 Int)` instance, as well as the `MonadStateX Index2 Int)` instance. It is important to realize that `succ2M` can be used in any monadic computation that includes at least these non-proper morphisms. Other type-indexed uses may be later incorporated with use of `succ2M`.

3.1 Creating Type-Indexes

We want to create an index that exists at both the type and value level uniquely. The type-level representation is used in differentiating the type of one type-indexed monad from another, and the value-level representation is used in differentiating a value of one type-indexed monad from another. We also need a link between the two—type inference needs to know that a particular index value always refers to a particular type-level index, and constructing values of a particularly-indexed monad requires knowing how to represent the type-level index in order to generate a value of that type-indexed monad, for instance when `return`ing a value. The `Index` type class exactly represents that correspondence between the term-level and type-level.

```
class Index ix where
   getVal :: ix
```

Creating a new type index comes in two predictable steps: we generate a simple atomic datatype, and provide an instance for `Index`.

```
data MyIndex = MyIndex deriving (Show, Eq)

instance Index MyIndex where   getVal = MyIndex
```

A singleton datatype and a trivial instance for each index are all we need for a new index to index into the monads. Template Haskell could be used to further-simplify the process, but it is already short. This simple addition of an index at both levels is all we need to completely introduce type-indexed monads. The idea is simple, direct, and gives us more options in how we use and think of monads.

We can now proceed to use these values at the type and term level interchangeably (via `getVal` and `::`) in order to differentiate between instances of a monad.

3.2 Implementation

Just as in the implementation of the `mtl` monads, indexed monads will each require (i) a data constructor or newtype; (ii) an instance for the `Monad` type class;

(iii) a type class for the non-proper morphisms; (iv) an instance for the datatype at that type class; and (v) a transformer version that satisfies the `MonadTrans` type class in order to be combined with other monads in a transformer stack. We develop the `StateX` monad, showing how automatic a translation it is from the original definition of `State`. We will underline all indexing code—the remaining code would define the original, non-indexed monad. The type-indexed version should directly arise from this prescriptive process of adding indexes. The same process works on other monads such as `Reader` and `Writer`, but is not shown for brevity's sake.

The StateX Monad We create the necessary data structure to represent a computation of the `StateX` monad, as well as a run function. We use `newtype` just as `mtl` does. We additionally define `mkStateX` to allow tagging the index type without directly ascribing a type, though this is only needed in the monad's definition and not in usage. We split the `run` function in two for the same reason.

The recurrent theme in indexing a monad is to have a value of the index type (its index value) be the first parameter to every non-proper morphism, and to include the index as a type parameter to the data structure and type class. The index is simply a label at the type level, and we use those labels to help identify which 'instance' of the monad is affected. The run function states that, given an index `ix`, a monadic computation of the *same* index of the monad and a starting state, we should execute the computation with that starting state. It is precisely the same as the original definition, except that we now index the monad at each usage.

```
newtype StateX ix s a = StateX {runStateX' :: s -> (a, s)}
mkStateX :: (Index ix) => ix -> (s->(a,s)) -> StateX ix s a
mkStateX _ v = StateX v
runStateX :: (Index ix) => ix -> StateX ix s a -> (s->(a,s))
runStateX _ m s = runStateX' m s
```

For `StateX` to be a monad, it must provide definitions for $>>=$ and `return`. Again, notice we must always ensure the index matches. We also see the way in which the only function of the `Index` type class is used, to generate a value corresponding to a particular type, effectively converting the type down to the only value that inhabits the type (ignoring bottom). Otherwise, the code is quite similar to the `State` monad's `Monad` instance.

```
instance (Index ix) ⇒ Monad (StateX ix s) where
  return a = mkStateX (getVal::ix) $ λ s → (a,s)
  ((StateX x)::StateX ix s a) >>= f = mkStateX (getVal::ix) $ λ s →
                 case (x s) of (v,s') → runStateX' (f v) s'
```

We also require a type class for the non-proper morphisms of our type-indexed monad, and we replicate the `MonadState` type class to handle our type-indexed versions.

```
class (Monad m, Index ix) => MonadStateX ix s m | ix m → s where
  getx :: ix → m s
  putx :: ix → s → m ()
```

The `getx` and `putx` functions are identical to those found in `MonadState`, except for the extra parameter for the type index. We now provide the implementation of the special effects of `StateX` to show how any `StateX` monad can perform the special behavior of the `MonadStateX` class. As before, we repeat the original definition's code, with our type-level indexing labels.

```
instance (Index ix) ⇒ MonadStateX ix s (StateX ix s) where
  getx (ixv::ix)   = StateX ixv $ λx → (x,x)
  putx (ixv::ix) s = StateX ixv $ λ_ → ((),s)
```

We now have the basic definition of a monad that we want. However, we have not yet created a transformer version of the monad, nor have we handled the special circumstances that arise when we use multiple monads of different indexes. Nor have we enabled `StateX` to work alongside the original `State` monad. We turn our attention next to handling these concerns.

The StateTX Transformer We now create a transformer version of the `StateX` monad, filling the same purpose as the `StateT` transformer does for the `State` monad. We create a new data structure and run function. Again, we must have the same type index to run the transformer. To complete the definition of the `StateTX` monad, we must provide the relevant instances for `Monad`, `MonadTrans`, and `MonadStateX`. Furthermore, to connect the `StateTX` transformer to the `StateX` monad, we need an instance for the `MonadStateX` type class in order to support the non-proper morphisms, and we need a means of lifting monadic computations of the transformer.

```
newtype StateTX ix s m a = StateTX  runStateTX' :: s -> m (a,s)
mkStateTX :: (Index ix) => ix -> (s->m(a,s)) -> StateTX ix s m a
mkStateTX _ v = StateTX v
runStateTX :: (Index ix) => ix -> StateTX ix s m a -> s -> m (a,s)
runStateTX _ m s = runStateTX' m s

instance (Index ix, Monad m) ⇒ Monad (StateTX ix s m) where
  return a = mkStateTX (getVal::ix) $ λs → return (a,s)
  ((StateTX x)::StateTX ix s m a) >>= f = mkStateTX (getVal::ix)
                                  $ λs → do (v,s') ← x s
                                            runStateTX' (f v) s'

--lifting a state transformer's operations
instance (Index ix) ⇒ MonadTrans (StateTX ix s) where
   lift x = mkStateTX (getVal::ix) $ λs'→x >>= λx'→return(x',s')

instance(Index ix,Monad m) ⇒ MonadStateX ix s (StateTX ix s m) where
   getx (ixv::ix)   = mkStateTX ixv $ λ(s1::s) → return (s1,s1)
   putx (ixv::ix) s = mkStateTX ixv $ λ_ → return ((),s)
```

By now this should look familiar. We have a transformer version of the `StateX` monad, and this transformer itself can be indexed. Up to this point, we haven't dealt with multiple indexes of a single kind of monad. This is the part that makes all the previous preparation worthwhile.

The following instance provides a way for index `ix2` to provide the functionality of the index `ix1` by explaining what to do when a `getx x1` or `putx ix1` computation is encountered. Note that the instance manually pipes its own state through behind the scenes by labeling the `ix2` index's state (`s::s2`) in an abstraction, performing the computation from index `ix1`, and then returning the pair that further threads the `ix2` index's state into the next computation. This is the key to separation of the two indexes' state.

```
instance (Monad m, Index ix1, Index ix2, MonadStateX ix1 s1 m )
  ⇒ MonadStateX ix1 s1 (StateTX ix2 s2 m) where
  getx (ixv::ix1) = mkStateTX (getVal::ix2) $ λ(s::s2) → do
                                              v1 ← getx (ixv::ix1)
                                              return (v1,s))
  putx (ixv::ix1) v1 = mkStateTX (getVal::ix2) $ λ(s::s2) → do
                                                    putx (ixv::ix1) v1
                                                    return ((),s)
```

In short, this defines how two indexes can coexist without affecting each other. It relies on the type information of the index, and *not* on the type information of what state is held by each monad. Each type-indexed monad could hold the same type of state and never be confused for another.

This does require GHC's `OverlappingInstances` pragma (among others) to be enabled. However, the overlap should only be required in the above instance to differentiate between two indexes that are easily tested for equality, and the pragma is not required at the site of usage.

Interoperability One of our stated goals is to reuse particular monadic features with existing code that most likely uses the original definitions of monads. We should therefore be able to mix the indexed versions of a monad with the original. We show in this section how to mix the `State` and `StateX` monads. The indexed monads provided in the Hackage package all can be used with the `mtl` monads.

Even in the definitions of Haskell's library-provided monads, they must provide instances for each monad to interact with every other. These 'cooperation' instances occupy a large part of the `mtl`'s codebase. We want the `StateTX` transformer monad to be able to provide the `MonadState` functionality, and we want the `StateT` transformer monad to provide the `MonadStateX` functionality. Each of these needs results in a new instance, simply defining the state management and adding the index labeling at the type level.

```
instance (MonadState s1 m, Index ix) ⇒ MonadState s1 (StateTX ix s2 m)
 where
  get   = mkStateTX (getVal::ix) $ λs → do n ← get
                                           return (n,s)
  put v = mkStateTX (getVal::ix) $ λs → put v >>= return ((),s)
```

```
instance (Monad m, MonadStateX ix s1 m, Index ix)
     ⇒ MonadStateX ix s1 (StateT s2 m) where
  getx (ixv::ix) = StateT $ λs → do n ← getx (getVal::ix)
                                    return (n,s)
  putx (ixv::ix) (v::s1) = StateT $ λs → do putx (getVal::ix) v
                                            return ((),s)
```

Although the code is not included in this paper, there is of course a need for instances often provided by monad definitions: instances for `Functor`, `MonadFix`, and instances that let the transformer version provide the non-proper morphisms of all the other 'standard' monads such as `IO`, `Error`, `Writer`, and `Reader`. This is no different than the original definitions of monads in that there is an initial price to pay for interoperability when defining the stack of monads that combine to create the monad with the desired capabilities. Similarly, we only have to define these once in a library and then simply use them. If we only need to interact with one copy of a monad, we could still just write the instances for `mtl`; if we need two copies, we write instances for the indexed monads; if we need any more copies, there are no more instances to write–and the indexed monad instances are essentially identical to the `mtl` instances. This time we gain an unlimited number of monads from it, not just one. The indexed library could even provide a set of bindings mimicking the original definitions, but implemented via the type-indexed definitions — then the library could become a drop-in replacement for even easier use. By creating another index, we hook into that entire set of instances, and *any* type-indexed monad can fully participate with all other type-indexed monads and the original `mtl` monads without writing more instances.

This does not entirely mitigate the need for instances. In particular, any home-grown monad still needs its own set of instances. If it does not interact with multiple copies of any one monad then we are not required to write those instances, and so no extra work is required; we simply may write one more set of instances that corresponds to an unlimited number of monads. This only serves to highlight the need to support reuse of the monad definitions.

3.3 Separation of Type-Indexed Monads

We should briefly reason about why two type-indexed `StateX` monads cannot access each others' state. We are interested in ensuring that one type-indexed monad cannot access a differently-indexed monad's state. We can devise a simple argument based on the types involved. The only way to access or modify the state of a `StateX` monad is to use the non-proper morphisms with the given index, or to directly create a $\texttt{StateX X}_1\ \texttt{s}_1\ \texttt{v}_1$ value. By having a single value in the index-type, we exclude the possibility of two different indexes existing at the same type. Therefore, an expression like $\texttt{getx X}_1\ \texttt{::}\ \texttt{StateX X}_1\ \texttt{s}_1\ \texttt{v}_1$ has no means of accessing the state $\texttt{s}_2$, which is tied to values of type $\texttt{StateX X}_2\ \texttt{s}_2\ \texttt{v}_2$. Just as Haskell disallows indexing into a list with a Boolean (`xs!!True`), at the type level we are excluding the possibility of using the wrong type of index to access the state. This guarantee cannot be argued as succinctly when using a

record that provides unfettered access to all of its fields. By looking at the type signature of a monadic function, we can tell definitively whether it is capable of seeing or modifying a particular indexed state.

We have checked a couple of properties over the indexed state monads using QuickCheck [7]. A problem arises in that the types change when we use different indexes. This property is great for understanding separation, but horrible for generating test cases. The approach was to design a small domain-specific language (DSL) for representing a computation, create our `Arbitrary` instances of that, and then translate it into a computation constrained with all of the indexes that we allowed in the DSL. This process is complex enough that it starts to obfuscate the properties being checked. In short, we looked at properties such as showing that using a `StateX` monad with the same operations will yield the same result as using just the `State` monad; we also tested that a put and get with a particular state monad (indexed or not), interrupted by any number of puts and gets from other distinct state monads, will still result in the originally placed value. We observed that the properties held, assuming we trust the DSL and its conversions. When a test approaches the complexity of the system on which we are checking properties, the value is not as clear.

3.4 Usage

Using indexed monads is virtually the same as using the original monads. We construct our computations using >>= and `return` (or more familiarly, `do`-notation) and the non-proper morphisms, and then run the computation in a combination of the run functions of the monads involved. Type ascriptions are similar in necessity as when using the basic monads. We assume that `StateX`, `StateTX`, `ReaderX`, and `ReaderTX`, are all defined.

Using a type-indexed monad by itself is only distinguished by the addition of the index in using the non-proper morphisms and in ascribing the type. In this example, type ascriptions are voluntary.

```
comp::(MonadReaderX MyIndex Int m) ⇒ Int → m Int
comp x = do a ← askx MyIndex
            return (x+a)

runcomp :: Int → Int
runcomp x = runReaderX MyIndex (comp x) 4
```

Indexed monads also work with their ancestors (the non-indexed versions), and do not interfere with each other as they are independently defined. They can also work with other indexes of themselves, as this example also shows. Note that we use the original `State` monad with an integer for its state, and that two differently-indexed `StateX` monads also use integers as their state without disturbing each other. We also see another indexed `StateX` monad containing boolean state, showing that it does not prohibit heterogeneous usage between the type-indexed monads. Also, note that the run function stacks the original monad between the indexed monads. Type-indexed monads impose no additional

restriction on the order in which you run them. The type ascriptions for `quad` and `runquad` are not necessary.

```
data Ix1 = Ix1 deriving (Show, Eq)
instance Index Ix1 where getVal = Ix1
-- and similarly for Ix2, Ix3.

quad :: (MonadStateX Ix1 Bool m, MonadStateX Ix2 Int m,
         MonadStateX Ix3 Int  m, MonadState Int m)
    ⇒ m Int
quad = do a ← getx Ix1
          b ← getx Ix2
          c ← getx Ix3
          d ← get
          return (if a then b+c else d)

runquad :: Bool → ((((Int,Int),Int),Int),Bool)
runquad b = flip (runStateX Ix1) b
          . flip (runStateTX Ix2) 2
          . flip runStateT 10
          . flip (runStateTX Ix3) 3
          $ quad

%> runquad True
((((5,3),10),2),True)
%> runquad False
((((10,3),10),2),False)
```

Next, we use two unrelated type-indexed monads to showcase their usage in conjunction with each other. Note e.g. that `ReaderX` and `StateX` can use the same index safely, as there is no confusion between which monad is referenced.

```
compM::(MonadReaderX Ix Int m, MonadStateX Ix String m) ⇒ Int → m Int
compM x = do  a ← askx MyIndex
              putx MyIndex $ "var"++(show a)
              return (a + x)

comp::Int → (Int,String)
comp x = flip (runReaderX Ix) 4  .  flip (runStateTX Ix) "" $ compM x

%> comp 5
(9,"var4")
```

We can use the `ErrorX` monad to throw and catch multiple errors in the same code. We use the `ascribe` function to streamline the examples — otherwise it would be unclear what instance to use to satisfy `ErrorX ix e`. We also use `runIdentity` — Just as there is no `runError` but only `runErrorT`, there is no `runErrorX`, only `runErrorTX`.

```
data Err = E1 | E2 Int | E3 String deriving (Show, Eq)

instance (Index ix) => ErrorX ix Err where
   noMsgx ix = E1; strMsgx ix = E3

ascribe::(MonadErrorX X1 Err m, MonadErrorX X2 String m) => m a->m a
ascribe = id

run = runIdentity . runErrorTX X1 . runErrorTX X2

%> run . ascribe $ return 5
   Right (Right 5)
%> run . ascribe $ throwErrorx X1 E1
   Left E1
%> run . ascribe $ throwErrorx X2 "no"
   Right (Left "no")
```

We can run our indexed `ErrorX` monads in whatever order we choose. Throwing the same `X2` error but running the monads in different orders naturally affects the nesting of the resulting `Either` type.

```
throw2no = throwErrorx X2 "no"

%> runIdentity . runErrorTX X1 . runErrorTX X2 . ascribe$ throw2no
   Right (Left "no")
%> runIdentity . runErrorTX X2 . runErrorTX X1 . ascribe$ throw2no
   Left "no"

%> run.ascribe$ catchErrorx X1 (throwErrorx X1 (E3 "err3"))
                               (λ(E3 s) -> throwErrorx X2 s)
   Right (Left "err3")
```

The original `mtl` couldn't handle multiple errors at once — that could only be simulated with a closed datatype, as we did with `Err`. Indexed monads allow us to throw and catch various types of errors within the same monadic code.

We have seen that type-indexed monads are used nearly identically to non-indexed monads. We have gained the ability to extend our usage of particular non-proper morphisms without re-defining them; instead, we only must generate a new type index, a trivial task.

4 Related Work

Monad Transformers. Moggi [8] introduces monads as a model of computation, and others [9–11] continue this invaluable work to introduce and develop the idea of monad transformers. GHC [2] distributes with the Monad Transformer Library [1]. The current work with type-indexed monads and type-indexed monad

transformers extends this work in a new direction, parameterizing the monads themselves (as opposed to parameterizing over monads), and allowing for more versatile use via indexing while leaving intact current patterns of non-indexed use. The new contribution is to *open* the monads in order to allow concurrent distinct instances of the monads to operate separately.

MonadLab. MonadLab [12] creates a domain-specific language utilizing Template Haskell [13] in order to encapsulate monad construction and abstract away implementation details. Being both a pedagogical tool for learning monads and a positive contribution to the expressivity and convenience of monads, MonadLab uses the meta-programming of Template Haskell to create an entirely new monad with the DSL-specified side-effects, replete with the required instances and regularly-specified non-proper morphisms. Type-indexed monads go in a different direction–rather than encapsulate and hide the details of monads, they expand on possible usage of the existing monads. Type-indexed monads provide a means to combine monadic code (avoiding index clashes rather than intersecting usage) and add more side-effects ad-hoc (via another indexed copy).

Parameterized Monads. Atkey [14] takes a categorical approach to monads that also introduces the notion of type-varying state. Rather than require that e.g. the `State` monad always inputs and outputs a particular state `s`, a `State` computation accepts state of type `s1` and outputs state of type `s2`. This of course requires a multiplicative property that chained `State` computations' outputs and inputs align to compatible types. This does not afford the ability to store multiple pieces of state, but does relax our state requirements by allowing us to change throughout our computation exactly what type of state is stored. Similar ideas are spread throughout the Haskell-Cafe mailing list, notably by David Roundy and Oleg Kiselyov in late 2006.

Monatron. Jaskelioff [15] takes a ground-up approach to monad transformers, approaching the issue of the quadratic lifting instances by standardizing the lifting procedure between transformers. Monatron accomplishes this by separating the usage from the implementation of non-proper morphisms, meaning that we can lift through *any* transformer, as opposed to defining lifting instances through particular transformers. However, in order to re-use monadic effects as we've discussed, the user still must define the lifting-depth for each interacting use of the monadic effects; this does not enable using library definitions (written in Monatron) in multiple ways once the lifting depths are set.

Type-indexed monads do not solve the issue of quadratic lifting instances per se, but mitigate the issue by providing the instances as necessary, so long as the monad with the desired non-proper morphisms can be indexed. Jaskelioff discuses an example of two `Error` monads stacked with a `State` monad and the confusion between which errors to throw; this is a direct translation into type-indexed monads that implement `Error` and `State`.

HLists. One approach to opening the contents of e.g. `State` is to use an `HList` [16] as the state. `HLists` provide a way to use type-level programming to guarantee that an index into the structure will result in a value, and of a particular type. One could use this to gain some flexibility into the 're-use' of a

monad by adding constraints on the state or environment to ensure a desired field is included. This decision still must be made initially, or else downstream uses cannot take the opportunity. Also, there is no guarantee of separation between the states, as all are available for modification. Type-indexed monads can still use a record at the values level instead of at the types level as for `HList`s, and yet we can still add more uses as well. By defining abstractions around usage of a type-indexed monad in a separate module and only exporting those abstractions and a few type synonyms, we can tell by the type of an expression whether it can access a particular state. `HList`s are concerned with heterogeneous lists themselves, and not in opening up usage of monads.

5 Conclusions and Future Work

We have introduced the notion of type-level indexes into monads to provide 'copies' of monads. We've shown how such an implementation compares to non-indexed monads to motivate their usefulness and approachability. We provided a reference of implementation details, and discussed how type-indexed monads allow us to reduce the amount of code necessary as well as reduce a source of possible errors by generalizing the process of duplicating particular side-effects.

Type-indexed monads provide a flexible framework for reusing monadic features. They open up the monad definitions with explicit indexing, allowing us to extend the use of non-proper morphisms without program-wide modification of existing uses. Indexed usage is added, rather than modifying current usage. Type-indexed monads solve the issue of using monads in library code by providing copies of monads, rather than manually generating a copy of needed monads along with all the instances. This both enhances code re-use while minimizing the chances for error introduction. Type-indexed monads also mitigate the number of instances that we need for monad transformers when multiple distinct monads can be replaced by indexed variants rather than hand-coded semantic copies of monadic functionality. If a stack of monads can be defined in terms of type-indexable monads, then all instances are already provided. Even if this is not so, one set of instances now applies to an unlimited number of monads. We write interfaces between kinds of monads, instead of between each implementation of a particular set of side-effects. Type-indexed monads are used in nearly identical fashion to non-indexed monads, providing a familiar interface that should aid in adoption. The reference implementation is available from the Hackage repository [3], an indexed approach to the `mtl` package. We would like to see type-indexed monads for even more monadic definitions in the future.

Acknowledgements

We'd like to thank the reviewers for quite helpful insights and suggestions, and we'd also like to thank Tom Schrijvers for some very helpful correspondence.

References

1. Gill, A.: mtl: The Monad Transformer Library. `http://hackage.haskell.org/package/mtl-1.1.1.0` (September 2010)
2. GHC: The Glasgow Haskell Compiler. `http://haskell.org/ghc/`
3. Snyder, M.: mtlx: Monad transformer library with type indexes, providing 'free' copies. `http://hackage.haskell.org/package/mtlx-0.1.5` (October 2010)
4. Harrison, W.L., Hook, J.: Achieving information flow security through precise control of effects. In: CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations, Washington, DC, USA, IEEE Computer Society (2005) 16–30
5. McBride, C.: Faking It - Simulating Dependent Types in Haskell. J. Funct. Program. **12**(5) (2002) 375–392
6. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
7. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2000) 268–279
8. Moggi, E.: An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Dept. of Comp. Sci., Edinburgh Univ. (1990)
9. Wadler, P.L.: Comprehending Monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, New York, NY, ACM (1990) 61–78
10. Wadler, P.: The Essence of Functional Programming. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albequerque, New Mexico (1992) 1–14
11. Jones, M.P.: Functional Programming with Overloading and Higher-Order Polymorphism. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text, London, UK, Springer-Verlag (1995) 97–136
12. Kariotis, P.S., Procter, A.M., Harrison, W.L.: Making Monads First-Class with Template Haskell. In: Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell, New York, NY, USA, ACM (2008) 99–110
13. Sheard, T., Jones, S.P.: Template Meta-Programming for Haskell. SIGPLAN Not. **37**(12) (2002) 60–75
14. Atkey, R.: Parameterized Notions of Computation. In: Proceedings of Workshop on Mathematically Structured Functional Programming. (July 2006)
15. Jaskelioff, M.: Monatron: An Extensible Monad Transformer Library. In: Programming Languages and Systems. Volume 5502/2009., Springer Berlin / Heidelberg (2008) 64–79
16. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly Typed Heterogeneous Collections. In: Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004) 96–107