Dakka: A dependently typed Actor framework for Haskell

Philipp Dargel

2018-08-31

Commit: 52df070fd3f19ed9277f6c40861a3407e08612fd

sha256: 1givsnqkl1ihdnbgdzdxv34xdc56sqh4x2apgny96cynw9px4a9i

To verify the hash compare the sha256 hash with the output of:

`nix-prefetch-url --unpack https://github.com/chisui/dakka/archive/52df070fd3f19ed9277f6c40861a3407e08612fd.tar.gz`.

Contents

1 Introduction **3**

1.1 Goals . . . 3

1.2 Result . . . 3

2 Fundamentals **3**

2.1 Actor Model . . . 3

2.2 Akka . . . 4

2.3 Cloud Haskell . . . 4

2.4 Dependent Typing . . . 4

2.5 Haskell Language features . . . 5

2.5.1 Heterogenous collections . . . 5

2.5.2 Typeable . . . 6

3 Implementation **7**

3.1 Actor . . . 7

3.2 ActorContext . . . 8

3.2.1 send . . . 9

3.2.2 create . . . 10

3.2.3 ActorRef . . . 10

3.2.4 Flexibility and Effects . . . 16

4 Testing **19**

5 Results **20**

5.1 Dependent types in Haskell . . . 20

5.2 Cloud Haskell . . . 20

6 Bibliography **20**

1 Introduction

The goal of this thesis is to create an Actor framework, similar to akka for Haskell. Haskell gives us many tools in its typesystem that together with Haskells purely functional nature enables us to formulate more strict constraints on actor systems. To formulate this I will leverage some of Haskells dependent typing features. Another focus of this reimplementation is the testability of code written using the framework.

I will show that leveraging Haskells advantages can be used to create an akka like actor framework that enables the user to express many constraints inside the typesystem itself that have to be done through documentation in akka. I will also show that this approach has some downsides that mostly relate to the maturity of Haskells dependent typing features.

1.1 Goals

I want to create an actor framework for Haskell that leverages the typesystem to provide safety where possible. The main issue where the typesystem can be helpful is by ensuring that only messages can be sent that can be handled by the receiving actor. It should ideally be possible for the user to add further constraints on messages and actors or other parts of the system.

Runtime components of this actor framework should be serializable if at all possible to provide. Serializeability is very desirable since it aids debugging, auditing, distribution and resilience. Debugging and auditing are aided since we could store relevant parts of the system to further review them. If we can sore the state of the system we can also recover by simply restoring a previous system state or parts of it. These states could then also be sent to different processes or machines to migrate actors from one node to another.

1.2 Result

2 Fundamentals

2.1 Actor Model

The Actor Model is a way of modeling concurrent computation where the primitive of computation is called an actor. A finite set of Actors that can communicate which each other is an Actor System. Actors can be sent messages and are characterized by the way they respond to these Messages. In Response to a message an Actor may:

1. Send a finite number of messages to other actors inside the same Actor System.

2. Add a finite number of new Actors to the Actor System.
3. Designate the behavior to be used for the next message it receives.

The Actor Model keeps these definitions very abstract. As a result of that aspect like identifying actors inside an Actor System and message ordering become implementation details.

2.2 Akka

Akka is an implementation of the Actor Model written in Scala for the JVM. In akka some design decisions where made while implementing the Actor Model that turned out to be very useful. In particular it enforces a strict order on messages. For every pair of actors in the Actor System it is ensured that messages from one of those Actors to the other are handled in the same order they where sent.

2.3 Cloud Haskell

Cloud Haskell is described by its authors as a platform for Erlang-style concurrent and distributed programming in Haskell.

Since Erlang-style concurrency is implemented using the actor model Cloud Haskell already provides a full fledged actor framework for Haskell. In addition there are rich facilities to create distributed Haskell system. It doesn't make creating distributed systems in Haskell easy but is capable of performing the heavy lifting.

Unfortunately Cloud Haskell has to be somewhat opinionated since some features it provides wouldn't be possible otherwise. The biggest problem is the fact that Haskell does not provide a way to serialize functions at all. Cloud Haskell solves this through the `distributed-static` package which requires some restrictions in the way functions are defined to work.

2.4 Dependent Typing

A dependent type is a type that depends on a value. Dependent types are are a way to express relationships between values inside of a typesystem. The canonic example for dependent types is a length indexed vector. A length indexed vector is a list which length is derivable from its type. This can be defined as a Haskell GADT:

```haskell
data Vec (l :: Nat) (a :: *) where
    VNil  :: Vec 0 a
    VCons :: a -> Vec l a -> Vec (l + 1) a
```

Where `Nat` is a kind that represents positive integers as types. This example illustrates one of the core properties of dependent types: Values and types are interchangeable, that means `0` and `l + 1` are types. In Haskell this behavior can be enabled using Language extensions.

2.5 Haskell Language features

Modern Haskell development involves many language features that are not present in the base language of *Haskell2010*. These features have to explicitly be enabled by enabling language extensions. Especially working with dependent types and using more advanced features of Haskells typesystem require many of these language extensions. Language extensions are enabled using `LANGUAGE` pragmas at the beginning of the file for which the extension should be enabled.

- `DataKinds`: Allows types to be promoted to kinds and values to types.
- `TypeFamilies`: Adds the ability to define type and data families. A type family can be thought of as a function on types.
- `PolyKinds`: Allows mixing different kinds. For example `k` in `l :: [k]` could normally only be of kind $*$ but with `PolyKinds` it may be any kind.

These two extensions are the foundation for dependent typing in Haskell. This enables the definition of `not` on types of kind `Bool`:

```
type family Not (a :: Bool) :: Bool where
    Not 'True  = 'False
    Not 'False = 'True
```

Or even `elem`:

```
type family Elem (e :: k) (l :: [k]) :: Bool where
    Elem e (e ': as) = 'True
    Elem e (a ': as) = Elem e as
    Elem e '[]       = 'False
```

2.5.1 Heterogenous collections

Another example of the usage of some of these extensions are heterogeneous lists. That is lists that can hold values of different types at once. This can be achieved by defining a GADT `HList` that is parametrized by a list of types such that each element of `HList` has a corresponding entry in the list of types:

```
data HList (l :: [*]) where
    HNil  :: HList '[]
    HCons :: a -> HList as -> HList (a ': as)
infixr 5 `HCons`
```

With this we can now create Lists with where each element is of a different type:

```
l :: HList '[Int, String, Bool]
l = 42 `HCons` "Hello World" `HCons` False `HCons` HNil
```

It is also possible to create a lookup function for elements of a given type that is only defined if the list contains an element of that type:

```
class HElem e (l :: [*]) where
    hElem :: HList l -> e

instance {-# Overlaps #-} HElem e (e ': as) where
    hElem (HCons e _) = e
instance {-# Overlappable #-} HElem e as => HElem e (a ': as)
    where
    hElem (HCons _ as) = hElem as
```

Unlike the previous example here a type class is used instead of a type family. Matching rules differ between type families and type classes. Type families allow Non-Linear Patterns, that is the same variable may occure multiple times inside of the pattern, but type classes don't. Type classes are matched exclusively by structure. As a result both instance declarations of `HElem` look the same to compiler. Constraints are only checked after the compiler already commited to a given declaration. In this context `HElem e (e ': as)` is equvalent to `(e ~ a)=> HElem e (a ': as)`. To prioritize which instance declaration will be chosen by the compiler the instances have to be annotated with overlapping instance pragmas.

2.5.2 Typeable

In the process of compiling Haskell all type information is removed since the aren't needed at runtime. Type information may be useful at runtime sometimes.

3 Implementation

3.1 Actor

Since akka is not written in a pure functional language each actor can also invoke any other piece of code. This implicit capability very useful for defining real world systems. So we have to provide ways to doing something like this as well if we want to use this framework in a real world situation. Invoking any piece of code also includes managing the actor system itself. For example stopping it all together, which also turns out to be very useful.

We need a way to identify specific actors at compile time to be able to reason about them. The best way to do so is by defining types for actors. Since Actors have a state this state type will be the type we will identify the actor with. We could have chosen the message type but the state type seems more descriptive.

```haskell
data SomeActor = SomeActor
  deriving (Eq, Show, Generic, Binary)
```

Note that we derive `Generic` and `Binary`. This allows the state of an actor to be serialized.

An actor now has to implement the `Actors` type class. On this typeclass we can ensure that the actor state is serializable and can be printed in human readable form to be included in error messages and log entries.

```haskell
class (Show a, Binary a) => Actor a where
```

The first member of this class will be a typefamily that maps a given actor state type (actor type for short) to a message type this actor can handle. If the message type is not specified it is assumed that the actor only understands `()` as a message.

```haskell
  type Message a
  type Message a = ()
```

To be able to send these messages around in a distributed system we have to ensure that we can send them around. They have to essentially fulfill the same constraints as the actor type itself. For this we create a constraint type alias (possible through the language extension `ConstraintKinds`):

```haskell
type RichData a = (Show a, Binary a)
```

Now the class header can be changed to:

```haskell
class (RichData a, RichData (Message a)) => Actor a where
```

Instead of a constraint type alias we could also have used a new class and provided a single `instance (Show a, Binary a)=> RichData a`. This would allow `RichData` to be partially applied. There is currently no need to do this though.

Next we have to define a way for actors to handle Messages.

```haskell
  behavior :: Message a -> ActorContext ()
```

`ActorContext` will be a class that provides the actor with a way to perform its actions.

Additionally we have to provide a start state the actor has when it is first created:

```haskell
  startState :: a
  default startState :: Monoid a => a
  startState = mempty
```

3.2 ActorContext

We need a way for actors to perform their actor operations. To recall actors may

1. Send a finite number of messages to other actors.
2. Create a finite number of new actors.
3. Designate the behavior to be used for the next message it receives. In other words change their internal state.

The most straight forward way to implement these actions would be to use a monad transformer for each action. Creating and sending could be modeled with `WriterT` and changing the internal state through `StateT`. The innermost monad wont be a transformer of course.

But here we encounter several issues:

1. To change the sate me must know which actors behavior we are currently describing.
2. To send a message we must ensure that the target actor can handle the message.
3. To create an actor we have to pass around some reference to the actor type of the actor to create.

The first issue can be solved by adding the actor type to `ActorContext` as a type parameter.

The second and third are a little trickier. To be able to send a message in a type safe way we need to retain the actor type. But if we would make the actor type explicit in the `WriterT` type we would only be able to send messages to actors of that exact type. Luckily there is a way to get both. Using the language extension `ExistentialQuantification` we can capture the actor type with a constructor without exposing it. To retrieve the captured type you just have to pattern match on the constructor. We can also use this to close over the actor type in the create case. With this we can create a wrapper for a send and create actions:

```haskell
data SystemMessage
    = forall a. Actor a => Send (ActorRef a) (Message a)
    | forall a. Actor a => Create (Proxy a)
  deriving (Eq, Show)
```

`ActorRef` is some way to identify an actor inside a actor system. We will define it later

Unfortunately we can't derive `Generic` for data types that use existential quantification and thus can't get a `Binary` instance for free. But as we will later discover we do not need to serialize values of `SystemMessage` so this is fine for now.

With all this we can define `ActorContext` as follows:

```haskell
newtype ActorContext a v
    = ActorContext (StateT a (Writer [SystemMessage]) v)
  deriving (Functor, Applicative, Monad, MonadWriter [
      SystemMessage], MonadState a)
```

Notice that we only need one `Writer` since we combined create and send actions into a single type. Since `ActorContext` is nothing more than the composition of several Monad transformers it is itself a monad. Using `GeneralizedNewtypeDeriving` we can derive several useful monad instances. The classes `MonadWriter` and `MonadState` are provided by the `mtl` package.

Since we added the actor type to the signature of `ActorContext` we need to change definition of `behavior` to reflect this:

```haskell
  behavior :: Message a -> ActorContext a ()
```

By deriving `MonadState` we get a variety of functions to change the actors state. The other actor actions can now be defined as functions:

3.2.1 send

```haskell
send :: Actor a => ActorRef a -> Message a -> ActorContext b ()
send ref msg = tell [Send ref msg]
```

Notice that the resulting `ActorContext` doesn't have `a` as its actor type but rather some other type `b`. This is because these two types don't have to be the same one. `a` is the type of actor the message is sent to and `b` is the type of actor we are describing the behavior of.The `send` function does not have a `Actor b` constraint since this would needlessly restrict the use of the function itself. When defining an actor it is already ensured that whatever `b` is it will be an `Actor`.

We can also provide an akka-style send operator as a convenient alias for `send`:

```
(!) = send
```

3.2.2 create

```
create' :: Actor b => Proxy b -> ActorContext a ()
create' b = tell [Create b]
```

As indicated by the `'` this version of create is not intended to be the main one. For that we define:

```
create :: forall b a. Actor b => ActorContext a ()
create = create' (Proxy @b)
```

In combination with `TypeApplications` this enables us to create actors by just writing `create @TheActor` instead of the cumbersome `create'(Proxy :: Proxy TheActor)`.

3.2.3 ActorRef

We need a way to reference actors inside an actor system. The most straight forward way to do this is by creating a data type to represent these references. This type also has to hold the actor type of the actor it is referring to. But how should we encode the actor reference? The simplest way would be to give each actor some kind of identifier and just store the identifier:

```
newtype ActorRef a = ActorRef ActorId
```

References of this kind can't be be created by the user since you shouldn't be able to associate any `ActorId` with any actor type, since there is no way of verifying at compile time that a given id is associated a given actor type. The best way to achieve this is to modify the signature of `create` to return a reference to the just created actor.

```
create :: forall a. Actor a => ActorContext b (ActorRef a)
```

Additionally it would be useful for actors to have a way to get a reference to themselves. We can achieve this by adding:

```
self :: ActorContext a (ActorRef a)
```

To `ActorContext`.

Composing references

If we assume that a reference to an actor is represented by the actors path relative to the actor system root we could in theory compose actor references or even create our own. To do this in a typesafe manner we need to know what actors an actor may create. For this we add a new type family to the `Actor` class.

```
    type Creates a :: [*]
    type Creates a = '[]
```

This type family is of kind `[*]` so it's a list of all actor types this actor can create. We additionally provide a default that is the empty list. So if we don't override the `Creates` type family for a specific actor we assume that this actor does not create any other actors.

We can now also use this typefamily to enforce this assumption on the `create'` and `create` functions.

```
create' :: (Actor b, Elem b (Creates a)) => Proxy b ->
    ActorContext a ()
```

Where `Elem` is a typefamily of kind `k -> [k] -> Constraint` that works the same as `elem` only on the type level.

```
type family Elem (e :: k) (l :: [k]) :: Constraint where
    Elem e (e ': as) = ()
    Elem e (a ': as) = Elem e as
```

There are three things to note with this type family:

1. It is partial. It has no pattern for the empty list. Since it's kind is `Constraint` this means the constraint isn't met if we would enter that case either explicitly or through recursion.
2. The first pattern of `Elem` is non-linear. That means that a variable appears twice. `e` appears as the first parameter and as the first element in the list. This is only permitted on type families in Haskell. Without this feature it would be quite hard to define this type family at all.

3. We leverage that n-tuples of `Constraints` are `Constraints` themselves. In this case `()` can be seen as an 0-tuple and thus equates to `Constraint` that always holds.

The `Creates` typefamily is incredibly useful for anything we want to do that concerns the hierarchy of the typesystem. For example we could ensure that all actors in a given actor system fulfill a certain constraint.

```
type family AllActorsImplement (c :: * -> Constraint) (a :: *) :: Constraint where
    AllActorsImplement c a = (c a, AllActorsImplementHelper c (Creates a))
type family AllActorsImplementHelper (c :: * -> Constraint) (as :: [*]) :: Constraint where
    AllActorsImplementHelper c '[]       = ()
    AllActorsImplementHelper c (a ': as) = (AllActorsImplement c a, AllActorsImplementHelper c as)
```

We can also enumerate all actor types in a given actor system.

What we can't do unfortunately is create a type of kind `Data.Tree` that represents the whole actor system since it may be infinite. The following example shows this.

```
data A = A
instance Actor A where
    type Creates A = '[B]
    ...

data B = B
instance Actor B where
    type Creates B = '[A]
    ...
```

The type for an actor system that starts with `A` would have to be `'Node A '[Node B '[Node A '[...]]]`. What we can represent as a type though is any finite path inside this tree.

Since any running actor system has to be finite we can use the fact that we can represent finite paths inside an actor system for our actor references. We can parametrize our actor references by the path of the actor that it refers to.

Unfortunately creating references yourself isn't as useful as one might expect. The actor type is not sufficient to refere to a given actor. Since an actor may create multiple actors of the same type you also need a way to differenciate between them to reference them directly. The easiest way would be to order created actors by creation time and use an index inside the resulting list.

There are two problems with this approach though. Firstly we lose some typesafty since we can now construct actor references to actors that we can not confirm that they exist at compile time. Secondly this index would not be unambiguous since an older actor may die and thus an index inside the list of child actors would point to the wrong actor. We could take the possibility of actors dying into account which would result in essentially giving each immidiate child actor an uniqie identifier. When composing an actor reference then requires the knowledge of that exact identifier which is essentially the same as knowing the actor reference already.

The feature to compose actor references was removed because of these reasons. Actor references may now only be obtained from the `create` function and `self` for the current actor.

Typefamilies created for this feature are still useful though. They allow type level computation on specific groups of actors deep inside of an actor system.

Implementation specific references

Different implementations of `ActorContext` might want to use different datatypes to refer to actors. Since we don't provide a way for the user to create references themselves we don't have to expose the implementation of these references.

The most obvious way to achieve this is to associate a given `ActorContext` implementation with a specific reference type. This can be done using an additional type variable on the class, a type family or a data family. Here the data family seems the best choice since it's injective. The injectivity allows us to not only know the reference type from from an `ActorContext` implementation but also the other way round.

```haskell
    data CtxRef m :: * -> *
```

Additionally we have to add some constraints to `CtxRef` since we need to be able to serialize it, equality and a way to show them would also be nice. For this we can reuse the `RichData` constraint.

```haskell
class (RichData (CtxRef m)), ...) => ActorContext ... where
```

In our simple implementation I'm using an single `Word` as a unique identifier but we can't assume that every implementation wants to use it.

Now we have another problem though. Messages should be able to include actor references. If the type of these references now depends on the `ActorContext` implementation we need a way for messages to know this reference type. We can achieve this by adding the actor context as a parameter to the `Message` type family.

```haskell
type Message a :: (* -> *) -> *
```

Here we come in a bind because of the way we chose to define `ActorContext` unfortunately. The problem is the functional dependency in `ActorContext a m | m -> a`. It states that we know `a` if we know `m`. This means that if we expose `m` to `Message` the message is now bound to a specific `a`. This is problematic though since we only want to expose the type of reference not the actor type of the current context to the `Message`. Doing so would bloat every signature that wants to move a message from one context to another with equivalence constraints like

```haskell
forall a b m n. (ActorContext a m, ActorContext b n, Message a m ~
    Message b n) => ...
```

This is cumbersome and adds unnecessary complexity.

What we might do instead is add the reference type itself as a parameter to `Message`. This alleviates the problem only a little bit though since we need the actual `ActorContext` type to retrieve the concrete reference type. So we would only delay the constraint dance and move it a little bit. These constraints meant many additional type parameters to types and functions that don't actually care about them. Error messages for users would also suffer.

In the end I decided to ditch the idea of `ActorContext` implementation specific reference types. And went another route.

Since actor references have to be serializable anyway we can represent them by a `ByteString`.

```haskell
newtype ActorRef a = ActorRef ByteString
```

This might go a little against our ideal that we want to keep the code as typesafe as possible but it's not as bad as you might think. Firstly other datatypes that might have taken the place of `ByteString` wouldn't be any safer. We can still keep the user from being able to create references by themselves by not exporting the `ActorRef` constructor. We could expose it to `ActorContext` implementers through an internal package.

Sending references

A core feature that is necessary for an actor system to effectively communicate is the ability to send actor references as messages to other actors.

The most trivial case would be that the message to actor is an actor reference itself.

```haskell
instance Actor Sender where
    type Message Seder = ActorRef Reciever
```

```
    ...
```

This way limits the actor type of the receiver to be a single concrete type though. In particular we have to know the type of the actor (Receiver in the following) when defining the actor handling the reference (Sender in the following). So we would like this reference type to be more generic. A simple way to do this is to add a type parameter to the Sender that represents the Receiver.

```
instance (Actor a, c a) => Actor (Sender a) where
    type Message (Sender a) = a
    ...
```

`c` here can take any constraint that the Receiver actor has to fulfill as well. This is more generic but we `a` still represents a concrete type at runtime. The way this is normally done in Haskell is by extracting the commonalities of all Receiver types into a typeclass and ensure that all referenced actors implement that typeclass.

```
class Actor a => Reciever a
instance Reciever a => Actor (Sender a) where
    type Message (Sender a) = a
    ...
```

Although this is just a variation on the previous way since we only consolidated `c` into the `Reciever` class. Unfortunately we can't use `forall` in constraint contexts (yet; see `QuantifiedConstraints`). To get around this we can create a new message type that encapsulates the constraint like this:

```
data AnswerableMessage c = forall a. (Actor a, c a) =>
    AnswerableMessage (ActorRef a)
```

With this we can define the Sender like this:

```
class Actor a => Reciever a
instance Actor Sender where
    type Message Sender = AnswerableMessage Reciever
    ...
```

`Reciever` should not perform long running tasks though since that would provide a way to circumvent the actor model somewhat since the task would be performed in the context of the `Sender`. Ideally the class should only provide a way to construct a message the `Reciever` understands from a more generic type. We can express this with a typeclass like this:

```
class Actor a => Understands m a where
```

```
    convert :: m -> Message a
```

A `Sender` may use this class like this:

```
instance Actor Sender
    type Message Sender = AnswerableMessage (Understands SomeType)
    onMessage (AnswerableMessage ref) = do
        ref ! convert someType
```

Solving the problem of sending generic actor references has a huge problem though. Using existential quantification prevents `AnswerableMessage` from being serialized. Serializeability is a core requirement for messages though.

I do have an idea of how to get around this restriction but wasn't able to test it yet. To serialize arbitrary types we would need some kind of sum-type where each constructor corresponds with one concrete type. Since we can enumerate every actor type of actors inside a given actor system from the root actor we could use this to create a dynamic union type. An example of a dynamic union type would be `Data.OpenUnion` from the `freer-simple` package. To construct this type we need a reference to the root actor though so that type has to be exposed to the actor type in some way, either as an additional type parameter to the `Actor` class or to the `Message` typefamily. Adding a type parameter to `Actor` or `Message` requires would require rewriting ab big chunk of the codebase though. Sending `ActorRef` values directly is the only possible way for now.

3.2.4 Flexibility and Effects

By defining `ActorContext` as a datatype we force any environment to use exactly this datatype. This is problematic since actors now can only perform their three actor actions. `ActorContext` isn't flexible enough to express anything else. We could change the definition of `ActorContext` to be a monad transformer over `IO` and provide a `MonadIO` instance. This would defeat our goal to be able to reason about actors though since we could now perform any `IO` we wanted.

Luckily Haskells typesystem is expressive enough to solve this problem. Due to this expressiveness there is a myriad of different solutions for this problem though. Not all of them are viable of course. We will take a look at two approaches that integrate well into existing programming paradigms used in Haskell and other functional languages.

Both approaches involve associating what additional action an actor can take with the `Actor` instance definition. This is done by creating another associated typefamily in `Actor`. The value of this typefamily will be a list of types that identify what additional actions can be performed.

What this type will be depends on the chosen approach. The list in this case will be an actual Haskell list but promoted to a kind. This is possible through the `DataKinds` extension.

mtl style monad classes

In this approach we use mtl style monad classes to communicate additional capabilities of the actor. This is done by turning `ActorContext` into a class itself where `create` and `send` are class members and `MonadState a` is a superclass.

The associated typefamily will look like this:

```
type Capabillities a :: [(* -> *) -> Constraint]
type Capabillities a = '[]
```

With this the signature of `behavior` will change to:

```
behavior :: (ActorContext ctx, ImplementsAll (ctx a) (
    Capabillities a)) => Message a -> ctx a ()
```

Where `ImplementsAll` is a typefamily of kind `Constraint` that checks that the concrete context class fulfills all constraints in the given list:

```
type family ImplementsAll (a :: k) (c :: [k -> Constraint]) ::
    Constraint where
    ImplementsAll a (c ': cs) = (c a, ImplementsAll a cs)
    ImplementsAll a '[]       = ()
```

To be able to run the behavior of a specific actor the chosen `ActorContext` implementation has to also implement all monad classes listed in `Capabillities`.

```
newtype SomeActor = SomeActor ()
  deriving (Eq, Show, Generic, Binary, Monoid)
instance Actor SomeActor where
    type Capabillities SomeActor = '[MonadIO]
    behavior () = do
        liftIO $ putStrLn "we can do IO action now"
```

Since `MonadIO` is in the list of capabilities we can use its `liftIO` function to perform arbitrary `IO` actions inside the `ActorContext`.

`MonadIO` may be a bad example though since it exposes to much power to the user. What we would need here is a set of more fine grain monad classes that each only provide access to a limited set of IO operations. Examples would be Things like a network access monad class, file system class, logging class, etc. These would be useful even outside of this actor framework.

the Eff monad

The `Eff` monad as described in the `freer`, `freer-effects` and `freer-simple` packages is a free monad that provides an alternative way to monad classes and monad transformers to combine different effects into a single monad.

A free monad in category theory is the simplest way to turn a functor into a monad. In other words it's the most basic construct for that the monad laws hold given a functor. The definition of a free monad involves a hefty portion of category theory. We will only focus on the aspect that a free monad provides a way to describe monadic operations without providing an interpretations immediately. Instead the there can be multiple ways to interpret these operations.

When using the `Eff` monad there is only one monadic operation:

```
send :: Member eff effs => eff a -> Eff effs a
```

`effs` has the kind `[* -> *]` and `Member` checks that `eff` is an element of `effs`. Every `eff` describes a set of effects. We can describe the actor operations with a GADT that can be used as effects in `Eff`:

```
data ActorEff a v where
    Send   :: Actor b => ActorRef b -> Message b -> ActorEff a ()
    Create :: Actor b => Proxy b -> ActorEff a ()
    Become :: a -> ActorEff a ()
```

With this we can define the functions:

```
send :: (Member (ActorEff a) effs, Actor b) => ActorRef b ->
    Message b -> Eff effs ()
send ref msg = Freer.send (Send ref msg)

create :: forall b a effs. (Member (ActorEff a), Actor b) => Eff
    effs ()
create = Freer.send $ Create (Proxy @b)

become :: Member (ActorEff a) effs => a -> Eff effs ()
become = Freer.send . Become
```

We could also define these operations without a new datatype using the predefined effects for `State` and `Writer`:

```haskell
send :: (Member (Writer [SystemMessage]) effs, Actor b) =>
    ActorRef b -> Message b -> Eff effs ()
send ref msg = tell (Send ref msg)

create :: forall b a effs. (Member (Writer [SystemMessage]), Actor
    b) => Eff effs ()
create = tell $ Create (Proxy @b)
```

`become` does not need a corresponding function in this case since `State` already defines everything we need.

4 Testing

One of the goals of the actor framework is testability of actors written in the framework. The main way that testability is achieved is by implementing a special `ActorContext` that provides a way to execute an actors behavior in an controlled environment. The name of this `ActorContext` is `MockActorContext`. `MockActorContext` has to provide implementations for `create`, `send` and `MonadState`. Additionally we need a way to execute a `MockActorContext`. One way to define `MockActorContext` is using monad transformers in conjunction with `GeneralizedNewtypeDeriving`.

```haskell
newtype MockActorContext a v = MockActorContext
    ( ReaderT (ActorRef a)
        ( StateT CtxState
            (Writer [SystemMessage])
        ) v
    )
  deriving
    ( Functor
    , Applicative
    , Monad
    , MonadWriter [SystemMessage]
    , MonadReader (ActorRef a)
    )
```

Where `CtxState` is used to keep track of actor instances that currently are in known to the context.

```haskell
data CtxState = CtxState
    { nextId :: Word
```

```haskell
    , states :: HMap ActorRef
    }
  deriving
    ( Show
    , Eq
    )
```

`HMap`

`MonadState` is a prerequisite for `ActorContext`

```haskell
instance Actor a => MonadState a (MockActorContext a) where
    get = do
        ref <- ask
        MockActorContext . gets $ ctxLookup ref
    put a = do
        ref <- ask
        MockActorContext $ do
            CtxState i m <- get
            let m' = HMap.hInsert ref a m
            put $ CtxState i m'
```

5 Results

5.1 Dependent types in Haskell

5.2 Cloud Haskell

6 Bibliography