Dakka: A dependently typed Actor framework for Haskell

Philipp Dargel

2018-??-??

Contents

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 stricter 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.

2 Technical considerations

2.1 Language choice

Even though the usage of Haskell was a hard requirement from the start I will demonstrate why it is arguably the best choice as well.

Akka is written in Scala which is a multi paradigm language with a heavy functional leaning mostly running on the Java Virtual Machine (JVM). Scala tries to aid integration with existing code written that runs on the JVM. Since most of the code written for the JVM is written in Java Scala has to integrate well with its imperative, object oriented nature. Haskell in contrast does

not compromises its purely functional features for integration purposes. Using a purely functional language we can rely more heavily on the typesystem which is essentially a prerequisite for dependent typing.

Haskell is the most widely used language that facilitates dependent typing. Scala also has support for dependent typing but not being purely functional somewhat diminishes that fact. There are other purely functional languages like `agda` and `idris` that support dependent typing but they have limiting factors that let Haskell still be the best choice.

- `agda` is rarely used as an actual runtime system but rather as a proof assistant, so creating a real world, distributed system with it is not feasible.
- `idris` would be a good fit for dependent typing, an even better one than Haskell even since depend types are supported native instead through language extension. The language itself seems to be a little immature at the current time though, especially the library ecosystem is extremely sparse.

Haskell may not have native depended types but they are supported. Being able to rely on a vast number of existing libraries is a huge point in Haskells favor. Especially the `cloud-haskell` platform is extremely useful. In addition to implementing a form of actor framework itself it eases the creation of distributed systems immensely. Most of the heavy lifting on the network and operations side can be done through `cloud-haskell`.

Another concern is my knowledge of other languages. I already have extensive knowledge of Haskell and many of it's more advanced concepts through my bachelors project, TA work in PI3 and private experience. Although this is not the main concern it is another point in Haskells favor.

2.2 Build tool

There are several build tools and build tool combinations for Haskell that are commonly used. None of these is The main one is `cabal` which is essentially required for any kind of Haskell development. There are wrappers for `cabal` that provide additional features and help organize dependencies. It is highly recommended to use one of those wrappers since using cabal without one can be very cumbersome. One of the main issues of cabal is that it installs all dependencies of projects globally. When working with multiple Haskell projects this will inevitably lead to conflicts and will land you in the so called *cabal hell*. There were attempts to mitigate these issues in cabal directly but these are cumbersome (sandboxes) to use or aren't finished yet (cabal new-build).

The most widely that is used most often is `stack`. It's main goal is to provide reproducible builds

for Haskell projects. It provides a good way of managing dependencies and Haskell projects. It works by bundling a GHC with a set of package versions that should all work with one another.

Another wrapper is `nix`. This build tool isn't Haskell specific though, but it lends itself to Haskell development. Nix calls itself a *Purely Functional Package Manager*. Like stack it's main goal is providing reproducible builds. It goes far further than stack in this regard though. It sandboxes the build environment as hard as possible. This goes so far as disabling network connections while building and stripping change dates from files to ensure that a build is performed in the same environment.

If nix can hold what it promises it would be the best build tool period. So for this project I elected nix as the main build tool. I will try to use it for everything I can from building the library itself to typesetting this very text. This will also be my first big nix project. There is also a Linux distribution that uses nix not only as it's default package manager but to build the entire system and it's configuration. I will be using this distribution for development as well.

3 Prior art

3.1 Akka

3.2 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.

4 Implementation

In the course of implementation we assume that several language extensions are enabled. When basic extensions like `FlextibleContexts`, `MultiParamTypeClasses` or

`PackageImports` or those that only provide syntactic sugar are used it wont be mentioned in the text. If the extension is significant for the code to work it will be mentioned. To take a look at which extensions where used you can run the following command inside of the `src` directory.

```
grep -Phore "(?<=LANGUAGE )\w+" | sort -u
```

4.1 Actor

Actors in the traditional actor model may only perform one of three actions in response to receiving a message:

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.

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.

```
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.

```
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.

```
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`):

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

Now the class header can be changed to:

```
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.

```
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:

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

4.2 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:

```
1  data SystemMessage
2      = forall a. Actor a => Send (ActorRef a) (Message a)
3      | forall a. Actor a => Create (Proxy a)
4    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:

```
1  newtype ActorContext a v
2      = ActorContext (StateT a (Writer [SystemMessage]) v)
3    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:

```
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:

4.2.1 send

```
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 descring 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
```

4.2.2 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)`.

4.2.3 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]
```

```
9       ...
```

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.

Implementation specific references

Different implementations of `ActorContext` might want to use different datatypes to refer to actors. In our simple implementation we are using `Word` but we can't assume that every implementation wants to use it.

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.

```
1     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.

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

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.

```
1   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

```
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`.

```
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.

Answering messages

We need a way to respond to messages. This can be done by including a reference to the actor to respond to in the message and capturing it's `Actor` implementation.

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

With this implementation we can't control what actors can be referred to by the references we send. For example we would like to constrain what messages the provided actor can handle. We can achieve this by further constraining the actor type that is captured by the constructor:

```
data AnswerableMessage m
  = forall a.
    ( Actor a
    , Message a ~ m
    ) =>
      AnswerableMessage (ActorRef a)
```

Now only actors that except exactly messages of type `m` can be referred to. This may be a little to restrictive though. Maybe we want to be able to allow some other class of actor. With `ConstraintKinds` we can also externalize these constraints.

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

This way we can express any constraint on actors and messages we want. The exact message constraint from above can expressed like this:

```
type C a = M ~ Message a
AnswerableMessage C
```

We now run into the problem again that type aliases can't be partially applied. So have to use the trick of creating a class that is only implemented once.

```
class (Actor a, c (Message a)) => MessageConstraint c a
instance (Actor a, c (Message a)) => MessageConstraint c a
```

With this we can rephrase the Above `AnswerableMessage` like this:

```
AnswerableMessage (MessageConstraint (M ~))
```

4.2.4 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:

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

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

```
1   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:

```
1 type family ImplementsAll (a :: k) (c :: [k -> Constraint]) ::
     Constraint where
2    ImplementsAll a (c ': cs) = (c a, ImplementsAll a cs)
3    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`.

```
1 newtype SomeActor = SomeActor ()
2   deriving (Eq, Show, Generic, Binary, Monoid)
3 instance Actor SomeActor where
4     type Capabillities SomeActor = '[MonadIO]
5     behavior () = do
6         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:

```
1 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`:

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

With this we can define the functions:

```
1 send :: (Member (ActorEff a) effs, Actor b) => ActorRef b ->
     Message b -> Eff effs ()
2 send ref msg = Freer.send (Send ref msg)
3
4 create :: forall b a effs. (Member (ActorEff a), Actor b) => Eff
     effs ()
5 create = Freer.send $ Create (Proxy @b)
6
7 become :: Member (ActorEff a) effs => a -> Eff effs ()
8 become = Freer.send . Become
```

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

```
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.

5 Testing

6 Results

6.1 Dependent types in Haskell

6.2 Cloud Haskell

6.3 Nix

7 Bibliography