Dakka: A dependently typed Actor framework for Haskell

Philipp Dargel

2018-11-10

Contents

1 Introduction

The goal of this thesis is to explore Actor frameworks, similar to Akka for Haskell, in particular how Haskells type system can be leveraged to improve uppon Akkas design. 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 these constraints I will leverage some of Haskells dependent typing features. Another focus of the thesis is the testability of code written using the created 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. The implementation of the Actor framework and important design decisions will be discussed in detail. I will also show that exessive usage of the typesystem 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 constraint Actor behaviors to minimize unexpected sideeffects. The main issue the typesystem may assist in 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 as they choose.

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

1.2 Result

I explored many aspects of Haskells typesystem and dependent typing features and how to apply them to the demain of Actor frameworks. As a result I created an API that fulfills many of the target features. Actors implemented in the created API can be executed in a test environment and to a certain degree in a destributed environment. Since the main focus was the API and how to constraint Actors written with it, the runtime aspect is not yet fully implemented.

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 with each other is an Actor System. Actors can recieve 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 this abstractness aspects 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 were made in the implemention the Actor Model that turned out to be very useful.

In Akka Actors are represented as classes that extend a common base class. When an Actor is created a new instance of the Actors class is created. The Actor classs' constructor may require additional arguments. Constructor arguments have to be supplied when an Actor is created. Actor classes have to provide an initial `receive` property which represents the Actors inital behavior. The type of the `receive` property is `PartialFunction[Any, Unit]` which means it's a possibly partial function that takes arguments of any type and returns a unit. An Actor class may have fields which represent internal state. In addition to fields they inherit the `become` method which provides a way to switch the behavior of the current Actor. Inside of its behavior the Actor has access to a reference to itself as well as to the sender of the currently handled message. Inside an Actor system messages of any type can be sent to any reference. There is a special message called `PoisonPill` which will terminate an Actor when received. When an Actor terminates it's designated supervisor is notified. Normally an Actors supervisor is the Actor that created it.[1]

In addition to these foundational Actors Akka provides more control and features for Actors like control over Actors mailboxes[2], message routing[3], clustering of Actor Systems[4] and more.

The way Akka is implemented lets it differ from the traditional Actor Model in some cases and extends it:

- Actors have two kinds of state: The internal state of the Actor class instance and the current `Receive` behavior.
- A strict order on messages is enforced. 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. A notable exception to this is the `Kill` message which terminates an Actor as soon as possible.
- Actors are named when they are created.
- Each Actor has access to the current Actor system via the `context` property. This gives any Actor access to every other Actor in the current Actor system. Actors can be enumerated or searched for by path.
- When an Actor terminates a message is sent to it's supervisor(s).
- Since Scala isn't a pure language Actors can perform arbitrary operations in response to their behavior.
- Akka expects messages to be immutable.

There is also an alternative package to the described Actor base package which adds type information to Actors[5]. The main differences betweene those two packages is that Actor references are parametrized by the type of message that the defining Actor may handle and Actors have to define what kind of message they may receive.

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 systems in Haskell. 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 the function serialization problem 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 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.

In Haskell types and values are fundamentally different from each other. For dependent typing to be possible, there has to be a way to convert between types and values. To convert some values to types the `DataKinds` language extension was introduced. `DataKinds` allows data types to be promoted to kinds and their value constructors to types. The type equivalent to functions are type families. Haskell itself doesn't provide a mechanism to promote functions to type families. The *singletons* library provides a way to promote functions, as well as other facilities helpful for dependent typing. Since I ended up not using the *singletons* library I wont go into detail describing it here.

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 data types to be promoted to kinds and value constructors 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 extensions are the foundation for dependent typing in Haskell. This enables the definition of `not` on types of kind `Bool`:

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

Or even `elem`:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
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 Heterogeneous Maps

A heterogeneous map may hold values of different types at once. A type of a value is determined by the type of the key it is associated with. The easiest way to associate a value type with a key is to parametrize the key by the type of the value. The map itself is parametrized by the type of key used. A lookup function may then have the signature `lookup :: k v -> HMap k -> Maybe v` and `insert :: k v -> v -> HMap k -> HMap k`.

There are ways to implement a completly typesafe variant of `HMap`, but if there is no way of manipulating the map directly it is safe to use `unsafeCoerce` as long as the API is safe.

The base for this `HMap` will be a standard `Data.Map.Map`. To be able to use that map both keys and values have to be of a single type. This can be achieved by creating custom `Key` and `Elem` types that capture and hide the concrete value type.

```
1  data Key k where
2      Key :: k a -> Key k
3
4  data Elem where
5      Elem :: a -> Elem
6
7  newtype HMap k = HMap (Map (Key k) Elem)
8      deriving Eq
```

To be able to use `Key` and `Elem` as key and value of `Map Key` has to implement `Ord`. Additionally we need equality on `HMap` for which both `Key` and `Elem` have to implement `Eq`.

To implement either `Eq` or `Ord` it is necessary to have an instance `Ord (k a)` for all `a`. Unfortunately it isn't possble to use the `forall` keyword in the context of instance declrations (yet [6]). A work around until `GHC 8.6` is to capture all commonly used classes inside of the `Key` and `Elem` constructors.

```haskell
data Key k where
    Key :: (Typeable (k a), Ord (k a), Show (k a))
        => k a -> Key k

instance Show (Key k) where
    showsPrec d (Key k) = showsPrec d k

instance Eq (Key k) where
    Key a == Key b = Just a == typeRep b

instance Ord (Key k) where
    -- first order by type then, if type are the same use Ord
    Key a `compare` Key b = typeRep [a] `compare` typeRep [b]
                          <> Just a      `compare` cast b
```

2.5.3 Typeable

In the process of compiling Haskell, all type information is removed since it isn't needed at runtime. Type information may be useful at runtime sometimes. If a type is hidden via existential quantification it may be useful to be able to get a `String` representation of the captured type for debug and/or `Show` purposes for example. Without some way of retrieving type information at runtime it would also be impossible to define an `Eq` instance for data types using existential quantification.

Runtime type information is provided by `Data.Typeable` in Haskell2010. The type class `Typeable` provides a single function `typeRep# :: TypeRep a` where `TypeRep a` is a representation of the type `a`. `typeRep#` and `TypeRep a` are only used internally. The module `Data.Typeable` exports ways to leverage this functionallity. GHC will derive an instance of `Data.Typeable` for every data type, type class and promoted data constructors automatically[7]. Manually defining an instance of `Data.Typeable` will cause an error to ensure that the type representation is valid.

Showing a type

Since `TypeRep` implements `Show` we can print any type at runtime. The `Show` implementation of `TypeRep` doesn't produce output that is equivalent to the way types are represented in Haskell error messages. This missmatch is partly due to the fact that there is no way to represent type aliases using `TypeRep` and some issues with the `Show` implementation itself[8].

```haskell
showsType :: forall a. Typeable a => ShowS
```

```
2 showsType = showString "<<"
3           . shows (typeRep (Proxy @a))
4           . showString ">>"
```

Dynamic values and type casting

`Typeable` enables the creation of Dynamic values in Haskell. To represent a dynamic value, all we have to do is capture the `Typeable` instance of the given type. Dynamic values are implemented by `Data.Dynmaic` in `base`. To construct a dynamic value `toDyn :: Typeable a => a -> Dynamic` is used. To extract a value `fromDynamic :: Typeable a => Dynmaic -> Maybe a` which only returns a value if the expected type `a` is the same as the captured one. Data extraction is only possible because there are runtime type representations that can be compared.

In the same way values can be extracted from dynamic values, it is possible to define a way to conditionally cast a value of one type to another, as long as those two types are the same, where it is only known at runtime if that is the case:

```
1 cast :: forall a b. (Typeable a, Typeable b) => a -> Maybe b
```

3 Implementation

3.1 Overview

The API is designed to be close to the API of Akka where appropriate. That means an Actors behavior is modeled by a function from a message to an action. An Actors action is a monad where all interactions with other Actors and the Actor system itself are functions that produce values in that monad.

To be able to perfom any type level computations on Actors and Actor systems there has to be some way of identifying specific kinds of Actors by type. Actors have to implement a typeclass `Actor a` where `a` is the type we can use do identify Actors by. The `Actor` class has a single function called `behavior`, which describes the behavior of the Actor. What kind of messages an Actor can handle and what kind of Actors it may create in response has to be encoded in some way as well.

The monad which models an Actors action is also a typeclass, that has roughly the form `Mondad m => ActorContext m`. This `ActorContext` is an *mtl* style monad class, which makes it possible to have different implementations of Actor systems at once. This makes it possible for

example to create one implementation that is meant for testing Actors and another one that actually performs these actions inside of a distributed Actor system. Defining the monad as a typeclass also makes it possible to use something else then *cloud-haskell* as a backend without having to rewrite any Actor implementations.

3.2 Actor

Since Akka is not written in a pure functional language, each Actor can also invoke any other piece of code. The implicit capability to perform arbitrary actions is very useful for defining real world systems. So we have to provide a way to perform `IO` actions as well if we want to use this framework in a real world situation. Actors may also want to manage the Actor system itself in some capacities that exceed the Actor model axioms. 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.

```
1 data SomeActor = SomeActor
2   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.

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

The first member of this class will be a type family 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.

```
1   type Message a
2   type Message a = ()
```

To be able to send these messages around in a distributed systems we have to ensure that they are serializable. They have to fulfill the same constraints as the Actor type itself. For this we create a constraint type alias (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 have allowed `RichData` to be partially applied. There is currently no need to do this.

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
```

3.3 ActorContext

We need a way for Actors to perform their Actor operations. As a reminder, the three Actor operations are:

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`.

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 issue are a little trickier. To be able to send a message in a type safe way, we need to retain the Actor type. If we would make the Actor type explicit in the `WriterT` type though, 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 have to pattern match on the constructor. We can also use `ExistentialQuantification` to close over the Actor type in the create case. With this technique we can create a wrapper for 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` provides some way to identify an Actor inside a Actor system we will define 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 I will show later 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 [
4        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:

```
1    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:

3.3.1 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`. `a` is the type of Actor the message is sent to and `b` is the type of Actor which behavior is beeing described. 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
```

3.3.2 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)`.

3.3.3 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 give Actors a way to refer to themselves 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 allow for Actor reference composition in a typesafe manner we need to know what Actors an Actor may create. To expose which Actors an Actor may create, we add a new type family to the `Actor` class.

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

The type family `Create` has the kind `[*]`, which represents a list of all Actor types Actor `a` 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 the `Create` typefamily to enforce the assumption on the `create'` and `create` functions that the type of any Actor created by an Actor has to be present in the `Create a` list.

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

Where `Elem` is a type family 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 The `Elem` 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 defining assumtions that concern the hierarchy of the Actorsystem. For example we can formulate an assumtion that states 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 to create a type of kind `Data.Tree` that represents the whole Actor system since it may be infinite. The tree representation of the following example would be infinite.

```
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 '[...]]]`. We can represent any finite path inside this tree as a type.

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 Actor references yourself isn't as useful as one might expect. The Actor type is not sufficient to refer to a given Actor. Since an Actor may create multiple Actors of the same type you also need a way to differenciate between them in order 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. Firstly we lose some type-safty since we can now construct Actor references to Actors for which 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 by giving each immidiate child Actor an uniquie identifier. Composing an Actor reference would require the knowledge of the exact identifier in that case. Having to know the unique identify for an Actor to create an Actor reference to it would make composition unfeasible.

I decided to remove the ability to compose Actor references since it would impose to many restrictions onto the form that Actor references could take. Furthermore the usabillity would be limited anyway.

Typefamilies created for in the process of implementing composition are still useful for other purposes. These typefamilies 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.

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

Additionally we have to add some constraints to `CtxRef`. Since we need to be able to serialize `CtxRef`, equality and a way to show them would also be nice. To ensure that `CtxRef` is serializable we can reuse the `RichData` constraint.

```
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.

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

Here we come in a bind because of the way we chose to define `ActorContext`. The functional dependency in `ActorContext a m | m -> a` forces us to create unwieldy typesignatures in this case. 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, 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 would mean many additional type parameters to types and functions that don't actually care about them. Compile errors would also come more cluttered, without adding useful information to the user.

Due to all of the stated concerns, I decided against the idea of `ActorContext` implementation specific reference types.

Instead of trying to create different representations for Actor references I chose to represent them using a `ByteString`. 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, to keep the code as typesafe as possible, but in this case the trade off is acceptable. 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 the Actor is an Actor reference itself.

```
1 instance Actor Sender where
2     type Message Seder = ActorRef Reciever
3     ...
```

This way limits the Actor type of the receiver to be a single concrete type. In particular we have to know the type of the Actor (Receiver in the following) when defining the Actor that handles 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.

```
1 instance (Actor a, c a) => Actor (Sender a) where
2     type Message (Sender a) = ActorRef a
3     ...
```

`c` may take any constraint that the Receiver Actor has to fulfill as well. This is more generic but `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.

```
1 class Actor a => Reciever a
2 instance Reciever a => Actor (Sender a) where
3     type Message (Sender a) = ActorRef a
4     ...
```

This is a variation on the previous implementation, 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 restriction we can create a new message type that encapsulates the constraint like this:

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

With this we can define the Sender like this:

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

`Reciever` should not perform long running tasks, since that would provide a way to circumvent the Actor model somewhat. Since any functions defined on `Reciever` are executed the context of the `Sender`, the message implicitly contains instructions for the `Sender` to run. 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:

```haskell
class Actor a => Understands m a where
    convert :: m -> Message a
```

A `Sender` may use this class like this:

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

Solving the problem of sending generic Actor references presents 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, 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` would require rewriting ab big chunk of the codebase. Sending `ActorRef` values directly is the only possible way for now.

3.3.4 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, 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. 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: 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.

In category theory a free monad 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 interpretations immediately. Instead 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 can also define these operations without a new datatype using the predefined effects for `State` and `Writer`:

```
1  send :: (Member (Writer [SystemMessage]) effs, Actor b) =>
       ActorRef b -> Message b -> Eff effs ()
2  send ref msg = tell (Send ref msg)
3
4  create :: forall b a effs. (Member (Writer [SystemMessage]), Actor
       b) => Eff effs ()
5  create = tell $ Create (Proxy @b)
```

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

3.4 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 a 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`.

```
1  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 known to the context.

```
data CtxState = CtxState
    { nextId :: Word
    , states :: HMap ActorRef
    }
  deriving
    ( Show
    , Eq
    )
```

`MonadState` is a prerequisite for `ActorContext` so an instance of that has to be provided.

```
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'
```

With this the actual definition of `ActorContext` for `MockActorContext` is pretty short.

```
instance (Actor a, MockActorContext a `CanRunAll` a) =>
   ActorContext a (MockActorContext a) where

```

```
3       self = ask
4
5       create' _ = do
6         ref <- MockActorContext ctxCreateActor
7         tell [Left $ Creates ref]
8         pure ref
9
10      p ! m = tell [Right $ Send p m]
```

To execute a single `MockActorContext` action, all monad transformer actions have to be executed. It doesn't make sense though, to export this capability directly, since `CtxState` should not be visible to the user. So exported variants on the core running function construct `CtxState` values themselves.

```
1 runMockInternal :: forall a v. Actor a => MockActorContext a v ->
     ActorRef a -> CtxState -> ((v, CtxState), [SystemMessage])
2 runMockInternal (MockActorContext ctx) ref = runWriter . runStateT
      (runReaderT ctx ref)
```

3.5 executing in a destributed environment

When executing an Actor inside a destributed environment, one has to take care of message passing and the actual concurrent execution. This is a hard problem. Luckily cloud-haskell already exists and provides exactly what is needed here. It is itself a Erlang-style Actor framework. Although all it's messages are untyped and every Actor has access to `IO`. This enables us to execute the previously defined typed Actors ontop of it.

As with the mock case the cetral entry point will be the `ActorContext`. All actions in cloud-haskell are inside of the `Process` monad. So we need to keep track of the Actors state and access to the `Process` monad. This can be achieved using a `newtype` wrapper around `StateT` transformer of `Process`:

```
1 newtype DistributedActorContext a v
2     = DistributedActorContext
3         { runDAC :: StateT a Process v
4         }
5   deriving (Functor, Applicative, Monad, MonadState a, MonadIO)
```

`GeneralizedNewtypeDeriving` is used to derive the normaly not derivable instances like `MonadIO`. `ActorContext` has to be implemented manually.

```haskell
instance (DistributedActorContext a `A.CanRunAll` a) => A.
    ActorContext a (DistributedActorContext a) where
     self = A.ActorRef . encode <$> liftProcess getSelfPid
     (A.ActorRef pid) ! m = liftProcess $ send (decode pid) m

     create' a = liftProcess $ do
         nid <- processNodeId <$> getSelfPid
         pid <- spawn nid (staticRunActor a)
         return . A.ActorRef . encode $ pid
```

All we have to do to implment `self` and `(!)` is to wrap/unwrap the process id in an `ActorRef` and use the functions that `Process` gives us. Notice that `create'` uses `staticRunActor` instead of `runActor`. More on this in the next section

Executing an `Actor` in this context, now means dispatching a `Created` signal and continously polling for messages.

```haskell
runActor :: forall a proxy. (A.Actor a, DistributedActorContext a
    `A.CanRunAll` a) => proxy a -> Process ()
runActor _ = void . runStateT (runDAC (initActor *> forever
    awaitMessage)) $ A.startState @a
  where
    initActor = A.behavior . Left $ A.Created
    awaitMessage = A.behavior . Right =<< liftProcess (expect @(A.
        Message a))
```

3.5.1 Creating Actors

Creating an `Actor` means spawning a new `Process` that executes `runActor` for that specific Actor type. The problem here is that the instruction on what the `Process` should do has to be serializable. Since functions are not Serializeability in Haskell cloud-haskell provides a workaround with the *distirbuted-static* package.

It provides a way to serialize references to values and functions that are known at compiletime and compose these. This is done using a heterogeneous map that has to be manually populated with all static values that the program may encounter while running. Unfortunately there is no way to register polymorphic functions like `runActor` this way. Luckily there is a way to enumerate all Actor types that exist in a given Actor system and could register a version of `runActor` for each one. I wasn't able to do this though for time reasons. As the hirarchy of the Actor system, described by the `Creates` typefamily is potentially infinite, the registration would have to

perform cycle detection on the type level.

As a result dakka currently hasn't the capability to run a full distributed Actor system.

4 Results

Although I do not have a runnable distributed system there are usable results in here.

4.1 Actor framework

I demonstrated that it is possible to create an Actor framework in Haskell that is capable of expressing many constraints about it's hierarchy and the capabilities of the Actors in it, using the typesystem.

4.2 Dependent types in Haskell

Dependent types are a powerful tool in Haskell. Unfortunately their usabillity is somewhat limited since they aren't supported natively. The lack of native support doesn't make it's use impossible but prevelent usage cumbersome. Promotion of values to types and demotion from types to values has to be done manually. For promotion and demotion there is the library *singletons*, which is written by the author of the `TypeInType` language extension, that is central to dependent types in Haskell. The *singletons* way to promote functions, by generating corresponding typefamilies for them but debugging these generated typefamilies is extremely hard. When there is an error in the types GHC either just has the names of the generated typefamilies, that are not very informative or there is just a statement that two types don't unify without a information on how GHC got those two types. These debugging problems make the usage of the *singletons* library and dependent types in general tedious for library authors. The effects are even worse for users of libraries that heavily leverage *singletons* that do not have a good understanding of that library or dependent types since they might get the same kind of cryptic errors unless the library author takes extra steps and exports special interface types that produce more readable errors.

As a result of this I opted to reduce the usage of dependent types in my code and do without the *singletons* library alltogether. Even though they are not dependent types many of the more advanced type-level-computation features Haskell provides were extremely useful.

4.3 Cloud Haskell

Big parts of the backend for the created API were implemented for cloud-haskell. The implementation was very straight forward for the most part since cloud haskell provides similar primitives to the API itself. Parts that required the serialization of polymorphic values were not that easily implemented. In fact the process of serialization itself is a big issue for cloud-haskell as well and requires a big amount of work, more than would be feasable for this thesis. As far as I can tell there shouldn't be any conceptual issues.

Since most of the effort went into creating a typed interface rather then the actual execution of it I can't comment on how cloud haskell actually performes as a backend for the created interface.

Bibliography

[1] Actors - akka documentation. Available from: https://doc.akka.io/docs/akka/2.5.16/actors.html, [31/08/2018].

[2] Mailboxes - akka documentation. Available from: https://doc.akka.io/docs/akka/2.5.16/mailboxes.html, [31/08/2018].

[3] Routing - akka documentation. Available from: https://doc.akka.io/docs/akka/2.5.16/routing.html, [31/08/2018].

[4] Cluster specification - akka documentation. Available from: https://doc.akka.io/docs/akka/2.5.16/common/cluster.html, [31/08/2018].

[5] Actors - akka documentation. Available from: https://doc.akka.io/docs/akka/2.5.16/typed/actors.html, [31/08/2018].

[6] Gert-Jan Bottu TS Georgios Karachalias. Quantified class constraints. Available from: https://i.cs.hku.hk/~bruno/papers/hs2017.pdf, [06/08/2018].

[7] 10.1. Language options — Glasgow Haskell Compiler 8.4.3 User's Guide. Available from: https://downloads.haskell.org/~ghc/8.4.3/docs/html/users_guide/glasgow_exts.html#deriving-typeable-instances, [31/08/2018].

[8] #14341 (Show instance for TypeReps is a bit broken) – GHC. Available from: https://ghc.haskell.org/trac/ghc/ticket/14341, [31/08/2018].

5 Appendix

All code produced including this thesis itself is developed on github at https://github.com/chisui/dakka. This thesis is based on the following commit.

Commit: `a1ddd17cc45f49420fe8355f01d6c1c31500f2f2`
sha256: `00siqqbilgwrgnbf19c3shb2z0w8x3zlm5swrixzmbzv5hnqj5p6`

To browse the repository at this commit visit

https://github.com/chisui/dakka/tree/a1ddd17cc45f49420fe8355f01d6c1c31500f2f2.

To verify the hash run:

```
nix-prefetch-url --unpack \
  https://github.com/chisui/dakka/archive/a1ddd17cc45f49420fe8355f01d6c1c31500f2f2.tar.gz \
  00siqqbilgwrgnbf19c3shb2z0w8x3zlm5swrixzmbzv5hnqj5p6
```