Protocols in Akka Typed

Programming Reactive Systems

Roland Kuhn

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`
- remove `context.sender`

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`
- remove `context.sender`

Incompatible changes open the possibility for some more cleanup:

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`
- remove `context.sender`

Incompatible changes open the possibility for some more cleanup:

- (*) turn stateful `trait Actor` into pure `Behavior[T]`

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`
- remove `context.sender`

Incompatible changes open the possibility for some more cleanup:

- (*) turn stateful `trait Actor` into pure `Behavior[T]`
- remove `system.actorOf`, instead require guardian behavior

Typing Akka actors

The goal: restrict `ActorRef` to sending the right message.

- add a type parameter: `ActorRef[T]`
- only allow sending type `T` with `tell`/`!`
- add type of understood messages to trait `Actor` (*)
- add type of understood messages to `ActorContext[T]`
- remove `context.sender`

Incompatible changes open the possibility for some more cleanup:

- (*) turn stateful `trait Actor` into pure `Behavior[T]`
- remove `system.actorOf`, instead require guardian behavior
- `ActorSystem[T]` is an `ActorRef[T]` for the guardian

Akka Typed: hello world!

Minimal protocol: accept one message and then stop.

```
val greeter: Behavior[String] =
    Behaviors.receiveMessage[String] { whom =>
        println(s"Hello $whom!")
        Behaviors.stopped
    }
```

Akka Typed: hello world!

```
object Hello extends App {
    val greeter: Behavior[String] =
        Behaviors.receiveMessage[String] { whom =>
            println(s”Hello $whom!”)
            Behaviors.stopped
        }

    // start a system with this primitive guardian
    val system: ActorSystem[String] = ActorSystem(greeter, ”helloworld”)

    // send a message to the guardian
    system ! ”world”

    // system stops when guardian stops
}
```

Proper channels with algebraic data types

```
sealed trait Greeter
final case class Greet(whom: String) extends Greeter
final case object Stop extends Greeter

val greeter: Behavior[Greeter] =
    Behaviors.receiveMessage[Greeter] {
        case Greet(whom) =>
            println(s"Hello $whom!")
            Behaviors.same
        case Stop =>
            println("shutting down ...")
            Behaviors.stopped
    }
```

Tangent: running actor programs

The best way is to start an `ActorSystem` and place the initialization code in the guardian's behavior:

```
ActorSystem[Nothing](Behaviors.setup[Nothing] { ctx =>
    val greeterRef = ctx.spawn(greeter, "greeter")
    ctx.watch(greeterRef) // sign death pact

    greeterRef ! Greet("world")
    greeterRef ! Stop

    Behaviors.empty
}, "helloworld")
```

Handling typed responses

A response of type `T` must be sent via an `ActorRef[T]`:

```
sealed trait Guardian
case class NewGreeter(replyTo: ActorRef[ActorRef[Greeter]]) extends Guardian
case object Shutdown extends Guardian
```

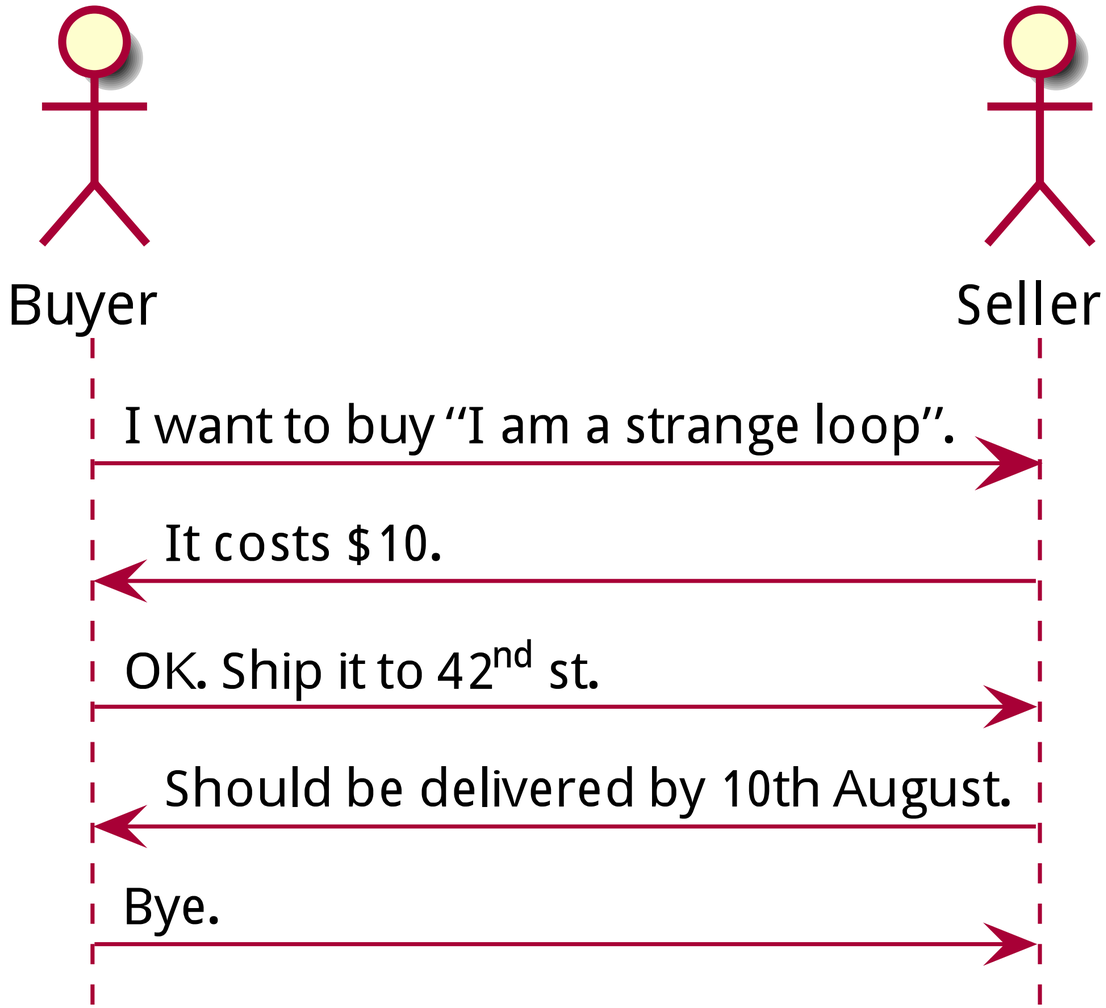
Handling typed responses

A response of type `T` must be sent via an `ActorRef[T]`:

```
sealed trait Guardian
case class NewGreeter(replyTo: ActorRef[ActorRef[Greeter]]) extends Guardian
case object Shutdown extends Guardian

val guardian = Behaviors.receive[Guardian] {
    case (ctx, NewGreeter(replyTo)) =>
        val ref: ActorRef[Greeter] = ctx.spawnAnonymous(greeter)
        replyTo ! ref
        Behavior.same
    case (_, Shutdown) =>
        Behavior.stopped
}
```

Modeling protocols with algebraic data types



Modeling protocols with algebraic data types

Modeling protocols with algebraic data types

```
case class RequestQuote(title: String, buyer: ActorRef[Quote])

case class Quote(price: BigDecimal, seller: ActorRef[BuyOrQuit])

sealed trait BuyOrQuit
case class Buy(address: Address, buyer: ActorRef[Shipping]) extends BuyOrQuit
case object Quit extends BuyOrQuit

case class Shipping(date: Date)
```

Summary

In this video we have seen:

- the philosophy behind how types were added to Akka actors
- the usage of typed communication channels with algebraic data types
- how to convey responses
- how references between message types can be used to model progress in a protocol