Actor Capabilities for Message Ordering

Colin S. Gordon, GulFest

Which Variant of Actors?

Each actor has an address + registered message handler

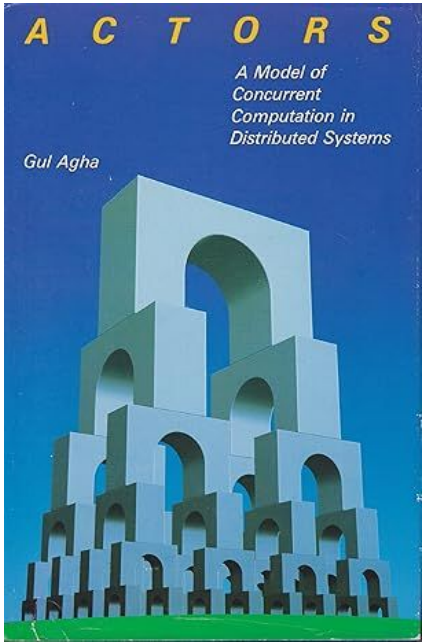
Actors send each other messages (async, by value)

Messages handled one by one

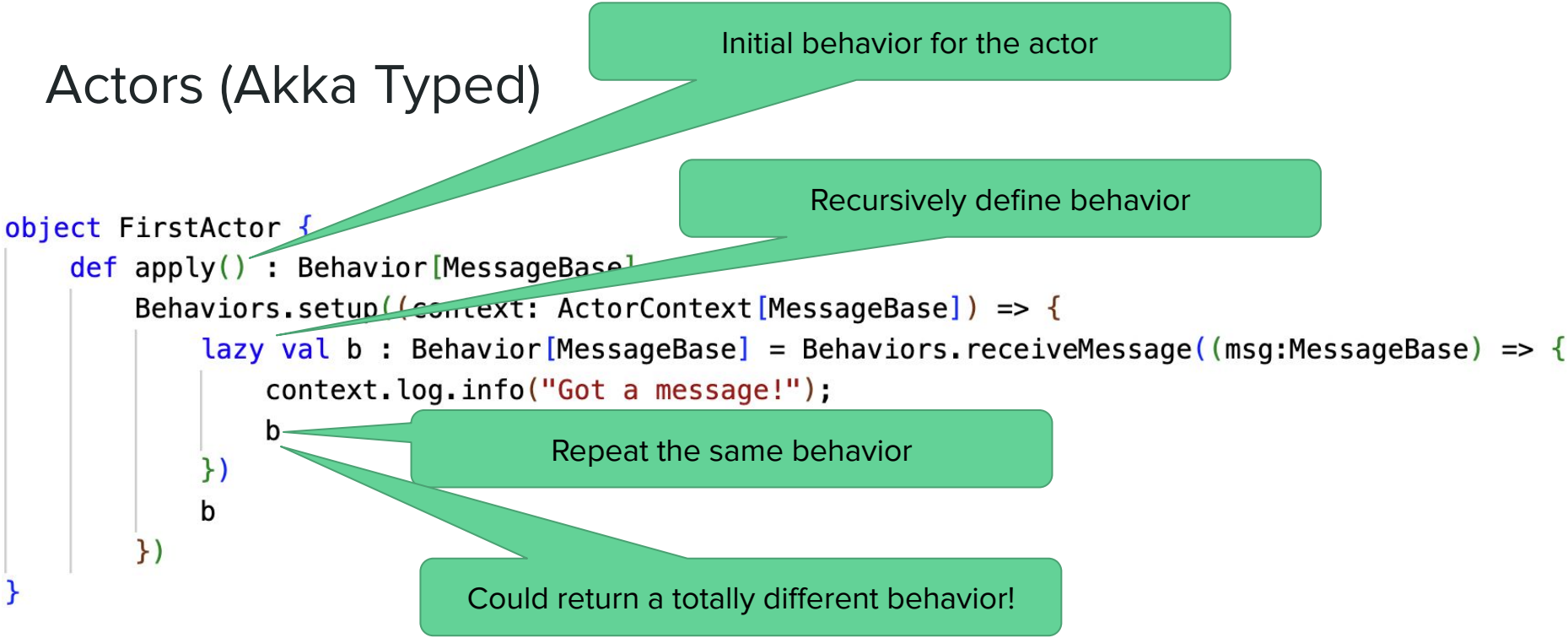
The handler:

1. Probably does some work (e.g., sending more messages)
2. Must return the handler to use for the next message

✅ No data races, no (blocking) deadlock, easy to learn



Actors (Akka Typed)



What happens if a handler isn't ready for a message? Bugs!

```
object ExampleActor {
    def apply() : Behavior[MessageBase] =
        Behaviors.setup((context: ActorContext[MessageBase]) => {
            // A steady-state behavior that doesn't handle Init
            lazy val running_behavior : Behavior[MessageBase] =
                Behaviors.receiveMessage((msg : MessageBase) => {
                    msg match {
                        case Act() => { context.log.info("Acting!"); running_behavior };
                        case Nop() => { context.log.info("Nothing!"); running_behavior };
                        case _ => throw new IllegalStateException("This handler should never see Init!")
                    }
                })
            Behaviors.receiveMessage((msg:MessageBase) => {
                msg match {
                    case Init() => { context.log.info("Initialized!"); running_behavior };
                    case _ => throw new IllegalStateException("This handler should only see Init!")
                }
            })
        })
}
```

This actor should see 1 Init message first, then many Act and Nop messages

First behavior

Switches to next behavior

Problem: Clients can use ActorRef[MessageBase] to send Init, Act, Nop in any order

What if they were more restricted?

Capabilities

Classic model for reasoning about program behaviour

> Arrange so the program **must present a capability** to do X
>
> ➜ No capability ⇒ No X!

Actor References As Capabilities

Actor References are dynamic capabilities

Can’t send messages to an actor you can’t name!

?? ! Message(3)

Akka ActorRef[T] is also a static *reference capability*

Statically requires only T is sent

ActorRef[T] may alias ActorRef[U] for U != T

```
val self : ActorRef[MessageBase] = context.self
val weaker : ActorRef[Nop] = self // subtyping!
```

Capability-Style Akka

```
object CapStyleActor {
    def apply() : Behavior[MessageBase] =
        Behaviors.setup((context: ActorContext[MessageBase]) =
            lazy val running_behavior : Behavior[MessageBase] =…
            Behaviors.receiveMessage((msg:MessageBase) => {
                msg match {
                    case InitWithReturn(r) => {
                        context.log.info("Initialized!")
                        val nextReference : ActorRef[Act|Nop] = context.self // (Act|Nop) <: MessageBase
                        r ! Ack(nextReference) // This reference cannot be used to init again
                        running_behavior // In Scala+Akka, must still be Behavior[MessageBase]
                    };
                    case _ => throw new IllegalStateException("This handler should only see Init!")
                }
            })
        })
}
```

This actor *chooses* what other can send it!

Typed as ActorRef[MessageBase]

3-way Agreement For A Global Invariant

Types guarantee an upper bound on type of message sent to an actor

From 3 things:

1. Behavior-Capability Consistency:
   ActorRef[T] originates from the creation of a Behavior[T]
   ➔ Initial agreement on message type
2. Capability Preservation:
   ActorRef[T] cannot be changed/split into a capability to send non-T things
   ➔ Send-side evolution preserves upper bound
3. Behavior Preservation: Behavior[T]'s next behavior is also Behavior[T]
   ➔ Receive-side evolution preserves upper bound

Let's adapt for message types changing over time, guided by this 3-way agreement

Actor Capabilities with Formal Languages (in Types)

Replace ActorRef[T] with ActorRef[L] for formal language L over message types

E.g., ActorRef[**Act**] can be used exactly once, to send an **Act** message

E.g., ActorRef[**Nop***] can be used many times, but only for **Nop** messages

When sending messages via an ActorRef[L], require:

1. ActorRef type *changes* after sending, to only allow valid continuations
2. Message type is a valid start of a string in L

$$\textsc{T-Send}\ \frac{\Gamma \vdash e : \tau \dashv \Gamma' \mid \chi \qquad \Gamma' \vdash_{\mathsf{path}} p : \mathsf{ActorRef}(L) \rightsquigarrow \mathsf{ActorRef}(\tau^{-1} L) \vdash \Gamma'' \qquad \tau^{-1} L \neq \emptyset}{\Gamma \vdash \mathsf{send}_\tau(p, e) : \mathbf{0} \dashv \Gamma'' \mid \chi}$$

Derivative: $\tau^{-1}L = \{ w \mid \tau w \in L \}$

Naively duplicating these actor capabilities is unsound

```
val x : ActorRef[(Nop*) · Act · (Nop*)] = … // ← Should only send one act
val y : ActorRef[(Nop*) · Act · (Nop*)] = x // ← Copy value w/ same type
// But now we have doubled our permissions!
x ! Act() // ← Send one act
y ! Act() // ← Send a second act! This shouldn't happen!
```

Return of the Revenge of the Formal Language Theory

Shuffle is the arbitrary interleaving of arbitrary words

$$L_1 \sqcup\!\sqcup L_2 = \{ w \mid x \in L_1, y \in L_2, x = x_0 \ldots x_n \wedge y = y_0 \ldots y_n \wedge w = x_0 y_0 \ldots x_n y_n \}$$

So for example, **Act** ⨿ **Nop*** = (**Nop***)·**Act**·(**Nop***)

Splitting Capabilities

Many capability systems have this problem: naive duplication is unsound

We'll borrow *splitting* from prior work

Copies inherit different permissions from original

$$\textsc{S-ActorRef}\ \frac{L' \sqcup L'' \subseteq L}{\Gamma \vdash \mathsf{ActorRef}(L) \prec \mathsf{ActorRef}(L') \divideontimes \mathsf{ActorRef}(L'')}$$

When creating aliases, the (post-aliasing) original and copy must split the caps

e.g., ActorRef[(**Nop***)·**Act**·(**Nop***)] < ActorRef[**Act**] ⁂ ActorRef[**Nop***]

e.g., ActorRef[(**Nop***)·**Act**·(**Nop***)] < ActorRef[(**Nop***)·**Act**·(**Nop***)] ⁂ ActorRef[**Nop***]

Behavior Typing: Behavior[L]

L will represent the sequences of messages the behavior *promises* to support

i.e., those described by the shuffle of all existing capabilities

Consider a Behavior[L] that handles X or Y itself, then returns another Behavior:

After handling X (or Y), next Behavior only needs to handle a “tail” of L

➔ After handling an X, must return a Behavior[$X^{-1}L$]

➔ After handling a Y, must return a Behavior[$Y^{-1}L$]

A catch if the actor uses self reference!

$$\textsc{T-Beh}\ \frac{\forall i \in 1..n.\ \Gamma, \mathsf{msg} : \tau_i \vdash e_i : \mathsf{Beh}\left(\left(\tau_i^{-1} L\right) \boxed{?}\right) \dashv \Gamma_i' \mid L_i' \qquad \mathsf{disjoint}(\overline{\tau})}{\Gamma \vdash \mathsf{beh}_L\left(\overline{\tau \rightsquigarrow e}^{\,n}\right) : \mathsf{Beh}(L) \dashv \epsilon \mid \{\epsilon\}}$$

Now We Can Write This

```
object FinalActor {
    def apply() : Behaviour[Init·Iter[Act|Nop]] = Be...iours.setup((context: BehaviourContext) => {
        lazy val running_behavior : Behaviour[Iter[Act|Nop]] = dist[Act,Nop,Iter[Act|Nop]](
                Behaviours.from((a:Act) => { context.log.info("Acting!"); running_behavior })
                .plus((n:Nop) => { context.log.info("Nothing!"); running_behavior }) )

        Behaviours.from((i:Init) => { context.log.info("Initialized!"); running_behavior })
    })
}
```

Behavior[(Act|Nop)*]

Behavior[Init·(Act|Nop)*]

Body type Behavior[Init^{-1}(Init·(Act|Nop)*)]

Conclusion

We can use (flow-sensitive) capability reasoning to control message ordering

- ActorRef[L] can only send message sequences in L
  - Careful use of *splitting* to deal with aliases
- Behavior[L] is obligated to handle message sequences in L — and maybe more
- All reasoning is local

Thanks!

New Capabilities

$$\textsc{T-Self}\ \frac{}{\Gamma \vdash \mathsf{self}_L : \mathsf{ActorRef}(L) \dashv \Gamma \mid L}$$

What about behaviors that send context.self elsewhere?

We view this as dynamic *creation of new capabilities*

➡ New obligations for next behavior, tracked with effects

The behavior, handling an X, must handle $X^{-1}L$ *and whatever new caps it makes*

L'_i is not reflected in L
The actor doesn't necessarily handle it until τ_i arrives!

$$\textsc{T-Beh}\ \frac{\forall i \in 1..n.\ \Gamma, \mathsf{msg} : \tau_i \vdash e_i : \mathsf{Beh}\left((\tau_i^{-1} L) \sqcup L'_i\right) \dashv \Gamma'_i \mid L'_i \qquad \mathsf{disjoint}(\overline{\tau})}{\Gamma \vdash \mathsf{beh}_L\left(\overline{\tau \rightsquigarrow e}^{\,n}\right) : \mathsf{Beh}(L) \dashv \epsilon \mid \{\epsilon\}}$$

Failures? (Tentative)

Key issue 1: Need to pick a failure model; wide variance here

Auto-restart? Stopped? Configurable?

Key issue 2: Lose messages in flight

Could require suffix-closed L in Behavior ➔ usually too restrictive

Better, but harder (WIP): restrictions depend on supervision policy

➔ Hard to deal with dynamic change in supervision