Composability and the actor model

Jan Macháček[‡]

April 17, 2018

Abstract

1 Actors

There are many actor frameworks and toolkits[1, 2, 5, 4]. Some toolkits make it very difficult to compose the behaviour of the actors[1, 4], other toolkits make it much easier[2, 5]. I consider the actor model as a way to *decompose* a system into self-contained units of functionality and state. I consider these units of functionality and state to be the smallest deployable entities; consequently, the communication between two actors must take form of messages travelling over a boundary that introduces latency and the risk of message loss. Given this definition, it does not make sense to require convenient general composition mechanism for general actors. Such composition mechanism is an abstraction that must attempt to hide the underlying latency and the risk of message loss[1].

An order processing system that fulfils the order contained within a request by decoding the order data from the request, calculates discounts and books the best delivery method, before finalising the order and encoding it into a response can compute the discount and delivery method at the same time (to use a loose term for now), though the other operations must be sequential; see algorithm 1.

$$decode \rightarrow \left\{ \begin{array}{l} delivery \\ discount \end{array} \right\} \rightarrow finalise \rightarrow encode$$

Algorithm 1. Order fulfilment

This type of processing is a typical result of decomposing the *order fulfilment* into its steps. The decomposition and the algorithm assume that the entire fulfilment operation is atomic and that its effects on the world are completely isolated until the *encode* operation successfully completes. (Imperative pseudo-code is shown in Listing 1.)

```
order = decode(request)
delivery = delivery(order)
discount = discount(order)
fulfiled = finalise(order, delivery, discount)
response = encode(fulfiled)
```

Listing 1: Fulfilment implementation I

This imperative style works when the *decode*, *delivery*, *discount*, *finalise*, and *encode* functions are "bare": that is, their types are $I \Rightarrow O$ for the appropriate types I and O. Changing the type O to higher-kinded type $\mathbb{F}$ provides a way to express that the returned value is not the raw value of type O, but that it is wrapped in some container $\mathbb{F}$. For $\mathbb{F} = Future$, *decode(Request): Order* becomes *decode(Request): Future⟦Order⟧*, which expresses the fact that the result of applying these functions to the input does not yield the result immediately; it *starts* the computation and returns the conainer $\mathbb{F} = Future$ which will hold the computed value. Viz Listing 2.

```
order ← decode(request)
(delivery, discount) ← delivery(order)
                     ∧ discount(order)
fulfiled ← finalise(order, delivery, discount)
response ← encode(fulfiled)
```

Listing 2: Fulfilment implementation II

The only difference between the two listings is the changing of assignment = operator to the sequence / shove operator ← ; and the usage of the ∧ operator to mean zip the two computations together. The type of *response* is *Future⟦Response⟧*

The *Future* can succeed or fail, making the code in Listing 2 feel right. The code does not handle any failures

The fulfilment operation isn't instant, and the world does not stop while it is processing. If the entire fulfilment operation could be done in the same memory and by only affecting that memory, it would be possible to use software transactional memory[3]. If the decomposed steps in the fulfilment process change the world (by initiating network communication, by printing labels, etc.) they cannot be simply retried: is it OK to make a delivery booking twice?; is it acceptable to print the packaging label multiple times? Deduplicating non-idempotent requests encounters storage limits: any deduplicating code can only deduplicate on a fixed number of requests. Locking reduces the throughput of a system as

[1] Consider the *Network File System*, which abstracts over unreliable networks to provide the illusion that a filesystem on a remote machine is the same as the filesystem on the local machine. The abstraction *leaks* when the network is slow or lossy, when a lot of small files are being accessed over the network, when a different application generates a lot of traffic on the NIC that also handles the NFS mount, etc.

the number of locks grows: locking code has to be able to old locks.

Even if isolation and atomicity were achievable in the context of this system, the *delivery* booking system might not be able to participate in any form of transaction. Worse still, the *delivery* booking step might have returned a successful booking, but by the time the *finalise* step executes, the delivery booking becomes invalid.

encode can fail.

DeliveryEstimate: Order => Days

Discount: Order => Days

Fulfilment(deliveryEstimate, discount): Order => Fulfilment = order => deliveryEstimate(order) and discount(order)

Pure actors do not compose.

2 Ramblings & notes

Asynchrony and concurrency is difficult; it is tempting to dismiss it as *too complicated* for applications that "simply" take a request and produce a response, where the work to produce the response involves simple data transformations and straight-forward logic.

```
The actual business logic that computes the delivery estimate for
the given item
deliveryEstimate(item: Item): Days

Mechanics for transforming between the request and response and
the application's data model
fromRequest(request: Request): Order
toResponse(estimate: Days): Response

The main handling code that computes the delivery estimate for
the order in the request, producing the estimate in the response
handle(request: Request): Response =
  items    ← parseItems(request)
  estimate ← 0
  for (item in items) {
    e ← deliveryEstimate(item)
    if (e > estimate) estimate = e
  }
  return toResponse(estimate)
}
```

Listing 3: Delivery estimate

If the *deliveryEstimate* function is *pure* (its result depends only on the given parameter and nothing else) making the *handle* function also pure, then it would appear that this code is a perfect candidate to use a thread-per-request model and not worry about any concurrency at all. The answer depends on how the application gets the *Request* and where the *Response* ends up[2], how many requests the application handles concurrently, what response time guarantees the application makes, and on the underlying implementation of the thread of execution.

Given N_t number of threads and N_r, there are the following scenarios:

* $N_r \ll N_t$ and low response time critical
* $N_r \leq N_t$
* $N_r > N_t$
* $N_r \gg N_t$

Thread-per-request model works if the delivery estimate module is used by another module that is also pure (see Listing 4)

```
The delivery estimate module with its specific handle function
module DeliveryEstimate =
  handle(request: Request): Response

The discount module with its specific handle function
module Discount =
  handle(request: Request): Response

toResponse(r1: DeliveryEstimate.Response,
           r2: Discount.Response): Response
handle(request: Request): Response =
  r1 ← DeliveryEstimate.handle(request)
  r2 ← Discount.handle(request)
  return toResponse(r1, r2)
```

Listing 4: Delivery and discount

References

[1] *Akka*. https://akka.io.

[2] *Scalaz Actors*. https://github.com/scalaz/scalaz/tree/series/7.3.x/concurrent/src/main/scala/scalaz/concurrent.

[3] *Software transactional memory*. https://wiki.haskell.org/Software_transactional_memory.

[4] *Thespian*. https://bitbucket.org/alinabi/thespian.

[5] *Transient*. https://github.com/transient-haskell/transient.

[2] I am not going to follow the rabbit hole of layering in more and more pure functions: programs run to interact with the world by the means of impure I/O. However, I do not dismiss purity or the ability to track the spread of arbitrary effects such as I/O.