An Exceptional Actor System (Functional Pearl)

Patrick Redmond
University of California, Santa Cruz
USA

Lindsey Kuper
University of California, Santa Cruz
USA

Abstract

The Glasgow Haskell Compiler is known for its feature-laden runtime system (RTS), which includes lightweight threads, asynchronous exceptions, and a slew of other features. Their combination is powerful enough that a programmer may complete the same task in many different ways — some more advisable than others.

We present a user-accessible actor framework hidden in plain sight within the RTS and demonstrate it on a classic example from the distributed systems literature. We then extend both the framework and example to the realm of dynamic types. Finally, we raise questions about how RTS features intersect and possibly subsume one another, and suggest that GHC can guide good practice by constraining the use of some features.

CCS Concepts: • **Software and its engineering → Concurrent programming structures**.

Keywords: actor framework, asynchronous exceptions, runtime system

ACM Reference Format:
Patrick Redmond and Lindsey Kuper. 2023. An Exceptional Actor System (Functional Pearl). In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell '23), September 8–9, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3609026.3609728

1 Introduction

Together with its runtime system (RTS), the Glasgow Haskell Compiler (GHC) is the most commonly used implementation of Haskell [3]. The RTS is featureful and boasts support for lightweight threads, two kinds of profiling, transactional memory, asynchronous exceptions, and more. Combined with the `base` package, a programmer can get a lot done without ever reaching into the extensive set of community packages on Hackage.

In that spirit, we noticed that there is nothing really stopping one from abusing the tools `throwTo` and `catch` to pass data between threads. Any user-defined datatype can be made into an asynchronous exception. Why not implement message-passing algorithms on that substrate?

We pursued this line of thought, and in this paper we present an actor framework hidden just under the surface of the RTS. The paper is organized as follows:

- Section 2 provides a concise summary of asynchronous exceptions in GHC and the actor model of programming.
- Section 3 details the implementation of our actor framework. We first show how actors receive messages of a single type, and then extend the framework to support dynamically typed actors, which receive messages of more than one type.
- Section 4 shows an implementation of a classic protocol for leader election using our actor framework. We then extend the actors with an additional message type and behavior without changing the original implementation.
- We reflect on whether this was a good idea in Section 5, by considering the practicality and performance of our framework, and conclude in Section 6 that asynchronous exceptions might be more constrained.

This paper is a literate Haskell program.[1]

2 Brief Background

In this section, we briefly review the status of asynchronous exceptions in GHC (Section 2.1) and the actor model of programming (Section 2.2); readers already familiar with these topics may wish to skip this section. Readers unfamiliar with the behavior of `throwTo`, `catch`, or `mask` from the `Control.Exception` module may wish to first scan the documentation of `throwTo` [4].

2.1 Asynchronous Exceptions in GHC

The Glasgow Haskell Compiler (GHC) is unusual in its support for *asynchronous exceptions*. Unlike synchronous exceptions, which are thrown as a result of executing code in the current thread, asynchronous exceptions are thrown by threads distinct from the current one, or by the RTS itself. They are used to communicate conditions that may require the current thread to terminate: thread cancellation, user interrupts, or memory limits.

Haskell '23, September 8–9, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0298-3/23/09.
https://doi.org/10.1145/3609026.3609728

[1] We use `GHC 9.0.2` and `base-4.15.1.0`. Our actor framework imports `Control.Exception` and `Control.Concurrent`, and we use the extensions `NamedFieldPuns` and `DuplicateRecordFields` for convenience of presentation. The leader election example of Section 4 additionally imports the module `System.Random` and uses the `ViewPatterns` extension. The appendices have other imports, which we do not describe here.

Asynchronous exceptions allow syntactically-distant parts of a program to interact in unexpected ways, much like mutable references. A thread needs only the `ThreadId` of another to throw a `ThreadKilled` exception to it. The standard library function `killThread` is even implemented as `(\x -> throwTo x ThreadKilled)`.[2] There is no permission or capability required to access this powerful feature.

Asynchronous exceptions are peculiar because they aren't constrained to their stated purpose of "signaling (or killing) one thread by another" [7]. A thread may throw any exception to any thread for any reason. This absence of restrictions means that standard exceptions may be reused for any purpose, such as to extend greetings: `(\x -> throwTo x $ AssertionFailed "hello")`. Even user-defined datatypes may be thrown as asynchronous exceptions by declaring an empty instance of `Exception` [6]. For example, with the declarations in Figure 1, it is possible to greet in vernacular: `(\x -> throwTo x Hi)`.

Asynchronous exceptions may be caught by the receiving thread for either cleanup or, surprisingly, recovery. An example of recovery includes "inform[ing] the program when memory is running out [so] it can take remedial action" [7]. The ability to recover from a termination signal seems innocuous, but it leaves asynchronous exceptions open to being repurposed.

2.2 The Actor Model

The actor model is a computational paradigm characterized by message passing. Hewitt et al. [5] write that "an actor can be thought of as a kind of virtual processor that is never 'busy' [in the sense that it cannot be sent a message]." In our setting, we interpret an actor to be a green thread[3] with some state and an inbox. When a message is received by an actor, it is handled by that actor's *intent function*. An intent function may perform some actions: send a message, update state, create a new actor, destroy an actor, or terminate itself. Unless terminated, the actor then waits to process the next message in its inbox. We will approximate this model with Haskell's asynchronous exceptions as the mechanism for message passing.

More concretely, we think of an actor framework as having the characteristics of a *concurrency-oriented programming language* (COPL), a notion due to Armstrong [1]. After describing our framework, we will make the case (in Section 5.1) that it has many of the characteristics of a COPL. To summarize Armstrong [1], a COPL (1) has processes, (2) which are strongly isolated, (3) with a unique hidden identifier, (4) without shared state, (5) that communicate via unreliable message passing, and (6) can detect when another process halts. Additionally, (5a) message passing is asynchronous so that no stuck recipient may cause a sender to become stuck, (5b) receiving a response is the only way to know that a prior message was sent, and (5c) messages between two processes obey FIFO ordering. While an actor system within an instance of the RTS cannot satisfy all of these requirements (e.g., termination of the main thread is not strongly isolated from the child threads), we will show that our framework satisfies many requirements of being a COPL with relatively little effort.

data *Greet* = *Hi* | *Hello* **deriving** *Show*
instance *Exception Greet*

Figure 1. `Show` and `Exception` instances are all that is required to become an asynchronous exception.

3 Actor Framework Implementation

In our framework, an actor is a Haskell thread running a provided main loop function. The main loop function mediates message receipt and makes calls to a user-defined intent function. Here we describe the minimal abstractions around such threads that realize the actor model. These abstractions are so minimal as to seem unnecessary; we have sought to keep them minimal to underscore our point.

3.1 Sending (Throwing) Messages

To send a message, we will throw an exception to the recipient's thread identifier. So that the recipient may respond, we define a self-addressed envelope data type in Figure 2 and declare the required instances.

Figure 3 defines a send function, `sendStatic`, which reads the current thread identifier, constructs a self-addressed envelope, and throws it to the specified recipient. For the purpose of explication in this paper, it also prints an execution trace.

3.2 Receiving (Catching) Messages

An actor is defined by how it behaves in response to messages. A user-defined intent function, with the type `Intent` shown in Figure 2, encodes behavior as a state transition that takes a self-addressed envelope argument.

Every actor thread will run a provided main loop function to manage message receipt and processing. The main loop function installs an exception handler to accumulate messages in an inbox and calls a user-defined intent function on each. Figure 3 defines a main loop, `runStatic`, that takes an `Intent` function and its initial state and does not return. It masks asynchronous exceptions so they will only be raised at well-defined points within the loop: during `threadDelay` or possibly during the `Intent` function.

[2] These identifiers are variously defined in `Control.Concurrent` and `Control.Exception` in `base-4.15.1.0`.

[3] A *green thread* (also "lightweight thread" or "userspace thread") is a thread not bound to an OS thread, but dynamically mapped to a CPU by a language-level scheduler. As opposed to heavier-weight OS threads, green threads simplify the implementation of a practical actor framework that supports large numbers of actors.

The loop in Figure 3 has two pieces of state: that of the intent function, and an inbox of messages to be processed. The loop body is divided roughly into three cases by an exception handler and a case-split on the inbox list:

(1) If the inbox is empty, sleep for an arbitrary length of time and then recurse on the unchanged actor state and the empty inbox.
(2) If the inbox has a message, call the intent function and recurse on the updated actor state and the remainder of the inbox.
(3) If, during cases (1) or (2), an `Envelope` exception is received, recurse on the unchanged actor state and an inbox with the new envelope appended to the end.

In the normal course of things, an actor will start with an empty inbox and go to sleep. If a message is received during sleep, the actor will wake (because `threadDelay` is defined to be *interruptible*), add the message to its inbox, and recurse. On the next loop iteration, the actor will process that message and once again have an empty inbox. Exceptions are masked (using `mask_`[4]) outside of interruptible actions so that the bookkeeping of recursing with updated state through the loop is not disrupted.

Unsafety. Before moving forward, let us acknowledge that this is *not safe*. An exception may arrive while executing the intent function. Despite our use of `mask_`, if the intent function executes an interruptible action, then it will be preempted. In this case the intent function's work will be unfinished. Without removing the message currently being processed, the loop will continue on an inbox extended with the new message. The next iteration will process the same message as the preempted iteration, effecting a double-send.

To avoid the possibility of a double-send, a careful implementor of an actor program might follow the documented recommendations for code in the presence of asynchronous exceptions: use software transactional memory (STM), avoid interruptible actions, or apply `uninterruptibleMask`. However, recall that message sends are implemented with `throwTo`, which is "*always* interruptible, even if it does not actually block" [4]. A solution is obtained "by forking a new thread" [7] each time we run an intent function, but this sacrifices serializable executions — an actor must be safe to run concurrently with itself. We opt for the simple presentation in Figure 3 and recommend that users write idempotent intent functions.

3.3 Dynamic Types

The actor main loop in Figure 3 constrains an actor thread to handle messages of a single type. An envelope containing

```
data Envelope a = Envelope {sender :: ThreadId, message :: a}
  deriving Show
instance Exception a ⇒ Exception (Envelope a)
type Intent st msg = st → Envelope msg → IO st
```

Figure 2. Message values are contained in a self-addressed envelope. Actor behavior is encoded as a transition system.

```
sendStatic :: Exception a ⇒ ThreadId → a → IO ()
sendStatic recipient message = do
    sender ← myThreadId
    putStrLn (show sender ++ " send " ++ show message
      ++ " to " ++ show recipient)
    throwTo recipient Envelope{sender, message}
runStatic :: Exception a ⇒ Intent s a → s → IO ()
runStatic intent initialState = mask_ $ loop (initialState, [])
  where
    loop (state, inbox) =
      catch
        (case inbox of
          [] → threadDelay 60000000                    (1)
            ≫ return (state, inbox)
          x : xs →
            (,) ⟨$⟩ intent state x ⟨*⟩ return xs)      (2)
        (λe@Envelope{} →
          return (state, inbox ++ [e]))                (3)
      ≫= loop
```

Figure 3. Message sends are implemented by throwing an exception. Actor threads run a main loop to receive messages.

the wrong message type will not be caught by the exception handler, causing the receiving actor to crash. We think the recipient should not crash when another actor sends an incorrect message.[5]

In this section, we correct this issue by extending the framework to support actors that may receive messages of different types. With this extension, our framework could be thought of as dynamically typed in the sense that a single actor can process multiple message types. This is similar to the dynamic types support in the `Data.Dynamic` module.

Furthermore, any actor may be extended by wrapping it ("has-a" style) with an actor that uses a distinct message type and branches on the type of a received message, delegating

[4]It is good practice to use `mask` instead of `mask_`, and "restore" the prior masking state of the context before calling a user-defined callback function. Such functions may be written with the expectation to catch asynchronous exceptions, for reasons mentioned in Section 2.1 or Marlow et al. [7]. For our purpose here, `mask_` is acceptable.

[5]Sending a message not handled by the recipient is like calling a function with wrong argument types, which would cause the thread to crash in a dynamically typed language. However, here both caller and callee are persistent, and we choose to locate the mistake in the caller.

to the wrapped actor where desired.[6] It may seem natural to encapsulate such actor-wrapping in combinators that generalize the patterns by which an actor is given additional behavior. However, here our goal is not to lean into the utility of a dynamically typed actor framework, but to point out how little scaffolding is required to obtain one from the RTS.

3.3.1 Sending Dynamic Messages. Instead of sending an `Envelope` of some application-specific message type we convert messages to the "any type" in Haskell's exception hierarchy, `SomeException` [6]. Figure 4 defines a new `send` function that converts messages, so that all inflight messages will have the type `Envelope SomeException`.

3.3.2 Receiving Dynamic Messages. On the receiving side, messages must now be downcast to the `Intent` function's message type. This is an opportunity to treat messages of the wrong type specially. In Figure 4 we define a new main loop, `runDyn`, that lifts any intent function to one that can receive envelopes containing `SomeException`. If the message downcast fails, instead of the recipient crashing, it performs a "return to sender." Specifically, it throws an exception (not an envelope) using the built-in `TypeError` exception.[7]

These changes do not directly empower actor intent functions to deal with messages of different types. We have only removed application-specific type parameters from envelopes. Actors intending to receive messages of different types will do so by downcasting from `SomeException` themselves. Such actors will use an intent function handling messages of type `SomeException`. We will see an example of this usage pattern in Section 4.2.

3.4 Safe Initialization

When creating an actor thread, it is important that no exception arrive before the actor main loop (`runStatic` in Figure 3) installs its exception handler. If this happened, the exception would cause the newly created thread to die. To avoid this, the fork prior to entering the main loop must be masked (in addition to the mask within the main loop).

Figure 5 defines the main loop wrapper we will use for examples in Section 4. It performs a best-effort check and issues a helpful reminder to mask the creation of actor threads.[8]

[6] It is not sufficient to wrap a message type in a sum and write an actor that takes the sum as its message. Such an actor will fail to receive messages sent as the un-wrapped type. To correct for this, one would need to change existing actors to wrap their outgoing messages in the sum. Section 3.3 generalizes this correction without requiring changes to existing actors.

[7] The extensions `ScopedTypeVariables`, `TypeApplications`, and the function `Data.Typeable.typeOf` can be used to construct a helpful type error message for debugging actor programs.

[8] We do not define a wrapper around `forkIO` to perform this masking because actors that perform initialization steps can currently do so before calling `run`. Section 4.2.3 is an example of this.

```
send :: Exception a ⇒ ThreadId → a → IO ()
send recipient = sendStatic recipient ∘ toException

runDyn :: Exception a ⇒ Intent s a → s → IO ()
runDyn intentStatic = runStatic intentDyn
  where
    intentDyn state e@Envelope{sender, message} =
      case fromException message of
        Just m → intentStatic state e{message = m}
        Nothing
          → throwTo sender (TypeError "...")
          ≫ return state
```

Figure 4. The dynamically typed framework upcasts before sending and downcasts before processing.

```
run :: Exception a ⇒ Intent s a → s → IO ()
run intent state = do
  ms ← getMaskingState
  case ms of
    MaskedInterruptible → runDyn intent state
    _ → error "mask the forking of actor threads"
```

Figure 5. Remind users to prevent initialization errors by masking forks.

4 Example: Ring Leader Election

The problem of *ring leader election* is to designate one node among a network of communicating nodes organized in a ring topology. Each node has a unique identity, and identities are totally ordered. Nodes know their immediate successor, or "next" node, but do not know the number or identities of the other nodes in the ring. A correct solution will result in exactly one node being designated the leader. This classic problem from the distributed systems literature serves to illustrate our actor framework, despite leader election being unnecessary in the context of threads in a process.

Chang and Roberts [2] describe a solution to the ring leader election problem that begins with every node sending a message to its successor to nominate itself as the leader (Figure 6). Upon receiving a nomination, a node forwards the nomination to its successor if the identity of the nominee is greater than its own identity. Otherwise, the nomination is ignored. We implement and extend that solution below.

4.1 Implementing a Leader Election

Each node begins uninitialized, and later becomes a member of the ring when it learns the identity of its successor. To represent this we define two constructors in Figure 7 for node state type, `Node`.

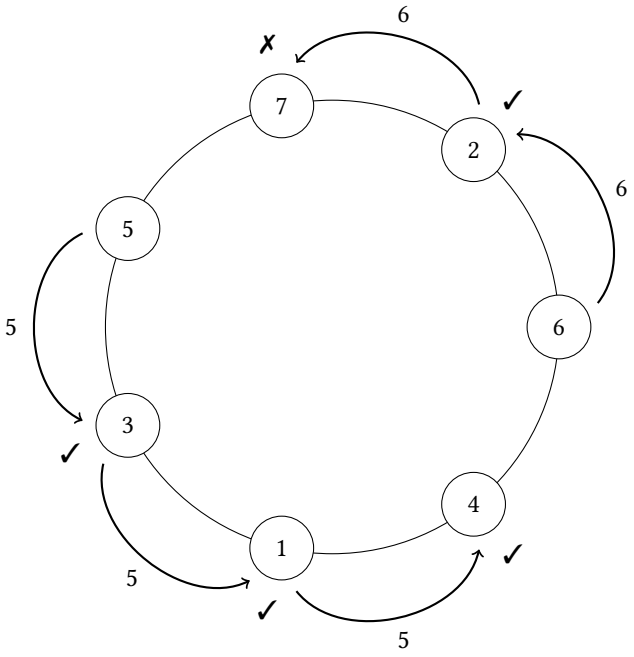


Figure 6. In-progress ring leader election with seven nodes (Chang and Roberts' 1979 solution). The node identities are unique and randomly distributed. Two nomination chains are shown: Node 5 nominated itself and was accepted by nodes 3, 1, and 4; next node 4 will nominate 5 to node 6 (who will reject it). Concurrently, node 6 nominated itself and was accepted by node 2 but rejected by node 7. For this election to result in a leader, node 7 must nominate itself.

Three messages (also defined in Figure 7 as type, `Msg`) will be used to run the election:

- `Init`: After creating nodes, the main thread initializes the ring by informing each node of its successor.
- `Start`: The main thread rapidly instructs every node to start the leader election.
- `Nominate`: The nodes carry out the election by sending and receiving nominations.

4.1.1 Election Termination. The node with the greatest identity that nominates itself will eventually receive its own nomination after it has circulated the entire ring. That same node will ignore every other nomination. Therefore the algorithm will terminate because node identities are unique and only one nomination can circumnavigate the ring.[9]

[9]In the context of this paper, termination is guaranteed because we have reliable message passing (see Section 5.1). In the context of a distributed system, with unreliable message passing, it is possible that no nomination makes it all the way around the ring. In such a situation, the algorithm could terminate without a winner.

```
data Node = Uninitialized | Member {next :: ThreadId}

data Msg
  = Init {next :: ThreadId}
  | Start
  | Nominate {nominee :: ThreadId}
  deriving Show

instance Exception Msg
```

Figure 7. Election nodes can be in one of two states, and they accept three different messages.

4.1.2 Node-Actor Behavior. The intent function for a node actor will have state of type `Node` and receive messages of type `Msg`, as defined in Figure 7. We show its implementation and describe each case below.

```
node :: Intent Node Msg
```

When an uninitialized node receives an `Init` message, it becomes a member of the ring and remembers its successor.

```
node Uninitialized
  Envelope {message = Init {next}} = do
    return Member {next}
```

When a member of the ring receives a `Start` message, it nominates itself to its successor in the ring.

```
node state@Member {next}
  Envelope {message = Start} = do
    self ← myThreadId
    send next $ Nominate self
    return state
```

When a member of the ring receives a `Nominate` message, it compares the nominee to its own identity. If they are equal, then the member wins and the algorithm stops. If the nominee is greater, then the member forwards the nomination to its successor.

```
node state@Member {next}
  Envelope {message = Nominate n} = do
    self ← myThreadId
    case () of
      _ | self ≡ n → putStrLn (show self ++ ": I win")
        | self < n → send next (Nominate n)
        | otherwise → putStrLn "Ignored nomination"
    return state
```

4.1.3 Election Initialization. The election initialization function is implemented in Figure 8. It takes the size of the ring and an unevaluated `IO` action representing node behavior, and takes the following steps to start the election:[10]

[10]The implementation shown doesn't handle rings of size 0 or 1. Also, we do not show thread cleanup.

```
ringElection :: Int → IO () → IO [ThreadId]
ringElection n actor = do
  nodes ← sequence ∘ replicate n ∘ mask_ $ forkIO actor    (1)
  ring ← getStdRandom $ permute nodes                      (2)
  mapM_
    (λ(t, next) → send t Init{next})                       (3)
    (zip ring $ tail ring ++ [head ring])
  mapM_ (λt → send t Start) ring                           (4)
  return ring
```

Figure 8. Ring leader election initialization.

(1) Create actors (with asynchronous exceptions masked).
(2) Randomize the order of actor `ThreadIds`.[11]
(3) Inform each actor of the `ThreadId` that follows it in the random order (its successor) with an `Init` message.
(4) Send each actor the `Start` message to kick things off.

To call the election initialization function, we construct an `IO` action by passing the node intent function and the initial node state to the actor main loop from Figure 5:

```
ringElection count $ run node Uninitialized
```

An election execution trace appears in Figure 9.

```
ThreadId 46 send Init {next = ThreadId 50} to ThreadId 49
ThreadId 46 send Init {next = ThreadId 47} to ThreadId 50
ThreadId 46 send Init {next = ThreadId 48} to ThreadId 47
ThreadId 46 send Init {next = ThreadId 49} to ThreadId 48
ThreadId 46 send Start to ThreadId 49
ThreadId 49 send Nominate {nominee = ThreadId 49} to ThreadId 50
ThreadId 46 send Start to ThreadId 50
Ignored nomination
ThreadId 50 send Nominate {nominee = ThreadId 50} to ThreadId 47
ThreadId 46 send Start to ThreadId 47
ThreadId 47 send Nominate {nominee = ThreadId 50} to ThreadId 48
ThreadId 48 send Nominate {nominee = ThreadId 50} to ThreadId 49
ThreadId 47 send Nominate {nominee = ThreadId 47} to ThreadId 48
ThreadId 46 send Start to ThreadId 48
ThreadId 49 send Nominate {nominee = ThreadId 50} to ThreadId 50
Ignored nomination
ThreadId 50: I win
ThreadId 48 send Nominate {nominee = ThreadId 48} to ThreadId 49
Ignored nomination
```

Figure 9. An execution trace of the ring leader election.

4.2 Extending the Leader Election

The solution we have shown solves the ring leader election problem insofar as a single node concludes that it has won. However, it is also desirable for the other nodes to learn the outcome of the election. Since it is sometimes necessary to extend a system without modifying the original, we will show how to extend the original ring leader election to add a winner-declaration round.

Since there is no message constructor to inform nodes of the election outcome, we will define a new message type

[11]The implementation of `permute` is in Appendix A.1.

```
type Exnode = (Node, ThreadId)
data Winner = Winner ThreadId deriving Show
instance Exception Winner
```

Figure 10. Extended nodes store node state alongside the greatest nominee seen. They accept one message in addition to those in Figure 7.

whose constructor indicates a declaration of who is the winner. We will extend the existing node intent function by wrapping it with a new intent function that processes messages of either the old or the new message types, with distinct behavior for each, leveraging the dynamic types support described in Section 3.3. The new behaviors are:

- Each node remembers the greatest nominee it has seen.
- When the winner self-identifies, they will start an extra round declaring themselves winner.
- Upon receiving a winner declaration, a node compares the greatest nominee it has seen with the declared-winner. If they are the same, then the node forwards the declaration to its successor.

Extended nodes will store the original node state (Figure 7) paired with the identity of the greatest nominee they have seen. This new extended node state is shown in Figure 10 as type `Exnode`. The new message type (`Winner`, also in Figure 10) has only one constructor and is used to declare some node the winner.

4.2.1 Declaration-Round Termination.

When an extended node receives a declaration of the winner that matches their greatest nominee seen, they have "learned" that that node is indeed the winner. When the winner receives their own declaration, *everyone* has learned they are the winner, and the algorithm terminates.

4.2.2 Exnode-Actor Behavior.

The intent function for the new actor will have state `Exnode` and receive messages of type `SomeException`. This will allow it to receive either `Msg` or `Winner` values and branch on which is received.

```
exnode :: Intent Exnode SomeException
```

Recall the implementation of the actor main loop function, `runDyn` from Figure 4. When we apply `exnode` to `runDyn`, the call to `fromException` in `runDyn` is inferred to return `Maybe SomeException`, which succeeds unconditionally. The `exnode` intent function must then perform its own downcasts, and we enable `ViewPatterns` to ease our presentation. Next we explain the two main cases, corresponding to the two message types the actor will handle.

The first case of `exnode`, shown in Figure 11, applies when an extended node downcasts the envelope contents to `Msg`. In each of its branches, node state is updated by delegating

```
exnode (n, great)
  e@Envelope{message = fromException → Just m} = do
    self ← myThreadId
    n'@Member{next} ← node n e{message = m}          (1)
    case m of
      Nominate{nominee} →
        if self ≡ nominee
        then send next (Winner self)                 (2)
             ≫ return (n', great)
        else return (n', max nominee great)          (3)
      _ → return (n', great)
```

Figure 11. When exnode receives a Msg, it delegates to node. It may also update the greatest nominee seen or trigger the winner-declaration round.

```
exnode state@(Member{next}, great)
  Envelope{message = fromException → Just m} = do
    self ← myThreadId
    case m of
      Winner w
        | w ≡ self → putStrLn (show self ++ ": Confirmed")
        | w ≡ great → send next (Winner w)
        | otherwise → putStrLn "Unexpected winner"
    return state
```

Figure 12. When exnode receives a Winner, it manages the winner-declaration round.

part of message handling to the held node. We annotate the rest of Figure 11 as follows:

(1) Delegate to the held node by putting the revealed Msg back into its envelope and passing it through the intent function, node, from Section 4.1.2.
(2) If the message is a nomination of the current node, start the winner round, because the election is over.
(3) Otherwise, the election is ongoing, so keep track of the greatest nominee seen.

The second case of exnode applies when a node downcasts the envelope contents to a winner declaration. Its implementation is shown in Figure 12. If the current node is declared winner, the algorithm terminates successfully. If the greatest nominee the current node has seen is declared winner, the node forwards the declaration to its successor. State is unchanged in each of these branches.

4.2.3 Extended Election Initialization.

The extended ring leader election reuses the initialization scaffolding from before (Figure 8). The only change is that the IO action passed to ringElection initializes the greatest nominee seen to itself, prior to calling run:

```
ringElection count $ do
  great ← myThreadId
  run exnode (Uninitialized, great)
```

A trace of an extended election appears in Appendix A.8.

5 What Have We Wrought?

Figure 3 shows that we have, in only a few lines of code, discovered an actor framework within GHC's RTS that makes no explicit use of channels, references, or locks and imports just a few names from default modules. The support for dynamic types, shown in Figure 4 as separate definitions, can be folded into Figure 3 for only a few additional lines.[12] We find it intriguing that this is possible and shocking that it is so easy.

5.1 Almost a COPL

In Section 2.2 we described an actor framework as having the characteristics of a *concurrency-oriented programming language* (COPL) [1]. Which of the COPL requirements does our framework satisfy? Here we review the criteria listed in Section 2.2:

(1) ✓ Threads behave as independent processes.
(2) ✗/✓ Threads are not strongly isolated because termination of the main thread terminates all others. However, if the main thread is excluded as a special case, then the set of other threads are strongly isolated.
(3) ✓ ThreadID is unique, hidden, and unforgeable.
(4) ✗ Threads may have shared state.
(5) ✗ Asynchronous exceptions do not behave as *unreliable* message passing.
(6) ✓ An actor can reliably inform others when it halts using forkFinally.

The message-passing semantics of our actor framework is nuanced. Documentation for the interfaces we use indicates that the framework provides *reliable synchronous message passing with FIFO order*. We call it *synchronous* because "throwTo does not return until the exception is received by the target thread" [4].[13] This means that a sender may block if the recipient is not well-behaved (e.g., its intent function enters an infinite loop in pure computation). We distinguish *well-behaved* intent functions, which eventually terminate or reach an interruptible point, from *poorly-behaved* intent functions, which do not. Assuming intent functions are well-behaved, the framework will tend to exhibit the behavior of *reliable asynchronous message passing with FIFO order* and occasional double-sends, because senders will not observe the blocking behavior of throwTo. By wrapping calls to the send

[12] Instead of wrapping the intent function, the framework's message downcast is performed in the exception handler.
[13] "Synchronous for me, but not for thee" might be the most correct characterization. Senders may experience GHC's asynchronous exceptions as synchronous, but recipients will always perceive them as asynchronous.

function with `forkIO` [7], we can achieve *reliable asynchronous message passing without FIFO order* even in the presence of poorly-behaved intent functions.[14] FIFO can then be recovered by message sequence numbers or by (albeit, jumping the shark) use of an outbox thread per actor. With those caveats in mind, our framework *mostly* satisfies Armstrong [1]'s criteria for message-passing semantics:

(5a) ✗/✓ A stuck recipient may cause a sender to become stuck, unless senders use `forkIO` or we assume the recipient is well-behaved.

(5b) ✗/✓ Actors know that a message is *received* (stored in the recipient inbox) as soon as `send` returns. However, they do not know that a message is *delivered* (processed by the recipient) until receiving a response.

(5c) ✓/✗ Messages between two actors obey FIFO ordering, unless `forkIO` is used when sending.

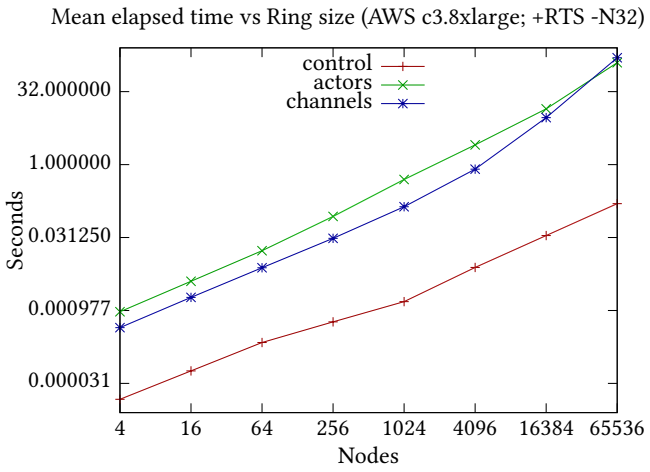
Our choice to wrap a user-defined message type in a known envelope type has the benefit of allowing the actor main loop to distinguish between messages and exceptions, allowing the latter to terminate the thread as intended. At the same time, though, this choice runs afoul of the *name distribution problem* [1] by indiscriminately informing all recipients of the sender process identifier. One strategy to hide to an actor's name and restore the lost security isolation is to wrap calls to the send function with `forkIO`. Another strategy would be to define two constructors for envelope, and elide the "sender" field from one.

We claim that our actor framework is *almost* a COPL. It also meets our informal requirements that actors can send and receive messages, update state, and spawn or kill other actors (though we have not shown examples of all of these). However, we do not mean to imply that our actor framework is practical; we merely mean to point out that it is, indeed, an actor framework.

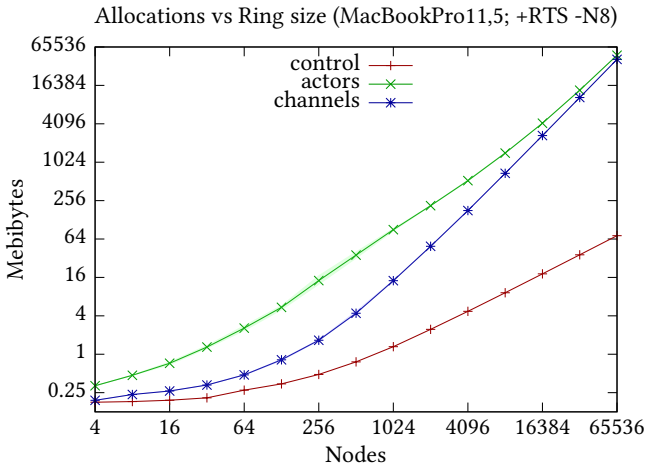
5.2 Summary of Performance Evaluation

We have described a novel approach to inter-thread communication. We believe it is prudent to compare the performance of this *unintended communication mechanism* against the performance of an *intended communication mechanism* to restore a sense that the ship is indeed upright. To that end, we re-implemented the extended ring leader election from Section 4 using channels — a standard FIFO communication primitive. We also implemented a "control"[15] to establish a lower bound on the expected running time of the actor-based and channel-based implementations.

We compared the running time of these implementations at ring sizes up to 65536 nodes on machines with 8, 32, and 192 capabilities. We also compared their total allocations over the program run at various ring sizes.



(a) The channel-based implementation is faster than the actor-based implementation, except at very large numbers of threads. We reproduced this result on machines with 8, 32, and 192 capabilities.



(b) The growth of allocations by the channel-based implementation eventually catches up to that of the actor-based implementation.

Figure 13. Representative selection of experimental results.

Our running time results (Figure 13a) show that the actor-based implementation is significantly slower than the channel-based implementation for ring sizes less than 8192 nodes, but surprisingly, it is marginally faster for more than 32768 nodes. The total-allocations result (Figure 13b) shows that allocations made by the channel-based implementation catch up to that of the actor-based implementation at large ring sizes, and we hypothesize that this convergence explains why the running time results swap places. Additionally, our results show that the running time of the extended ring

[14] If thread T_1 forks thread T_2 to send message M_2, and then T_1 forks thread T_3 to send message M_3, the RTS scheduler may first run T_3 resulting in M_3 reaching the recipient before M_2, violating FIFO if both messages have the same recipient.

[15] The "control" forks some number of threads that do nothing and immediately kills them.

leader election algorithm is invariant to the number of capabilities used by the RTS, making it a poor choice for a general evaluation of our actor framework, but sufficient for our purpose of confirming that channels are faster.

Appendices A.2 to A.5 give the source code for these benchmarks. Appendix A.6 details our experimental setup, and Appendix A.7 discusses more of the results.

6 Conclusion

Can we implement an actor framework with Haskell's threads and asynchronous exceptions? Our implementation and results show that we can, and this fact hints that perhaps asynchronous exceptions are at least as general as actors.

However, the actor framework we present is not an advancement: It is easy to use, but easy to use wrongly. It has acceptable throughput, but is slower than accepted tools. It requires no appreciable dependencies, no explicitly mutable data structures or references, no effort to achieve synchronization, and very little code only because *those things already exist, abstracted within the RTS.*

Should it have been possible to implement the actor framework we present here? Like many people, we choose Haskell because it is a tool that typically prevents "whole classes of errors," and also because it is a joy to use. But with the actor framework we present here, we achieve dynamically typed "spooky action at a distance" with frighteningly little effort. Perhaps the user-accessible interface to the asynchronous exception system should be constrained.

With the 9.6.1 release of GHC, a user of the RTS enjoys software transactional memory, asynchronous exceptions, delimited continuations (and perhaps extensible algebraic effects), together in the same tub. The water is warm — jump in! Will all the members of this new *extended* "awkward squad" [8] bob gently together, or will they knock elbows? Which of them can be implemented in terms of the others, and should their full power be exposed so that we can do so? We hope the reader will draw their own conclusions.

Acknowledgments

This paper grew out of a presentation at Portland State University's Programming Languages & Verification group in May 2023, and we are grateful to the Portland State PLV group for their enthusiasm and encouragement of our work. We also thank José Calderón, the members of the LSD Lab at UC Santa Cruz, and the anonymous Haskell '23 reviewers for their valuable feedback on drafts of this paper. Jonathan Castello, of the LSD Lab, contributed the title of this paper.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2145367. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Data Availability Statement

A version of the literate Haskell and benchmarks for this paper is on Zenodo; a living version is on GitHub [9].

A Appendix

A.1 Permute Function Implementation

In Section 4 we provided the implementation of a ring leader election in our actor framework. The implementation used `permute` to randomize the list of `ThreadId`. The `permute` function repeatedly pops a random element from the input and adds it to the output. Its implementation is as follows:

```
permute :: RandomGen g ⇒ [a] → g → ([a], g)
permute pool0 gen0
    = snd
    ∘ foldr pick (pool0, ([], gen0))
    $ replicate (length pool0) ()
  where
    pick () (pool, (output, g)) =
      let (index, g') = randomR (0, length pool − 1) g
          (x, pool') = pop pool index
      in (pool', (x : output, g'))
    pop (x : xs) 0 = (x, xs)
    pop (x : xs) n = (x:) ⟨$⟩ pop xs (n − 1)
    pop [] _ = error "pop empty list"
```

A.2 Actor Benchmark Implementation

In the extended ring leader election solution, the time to termination is the time necessary for the winner's self-nomination to pass around the ring once, plus the time for the winner-declaration to pass around the ring once. Termination is when a node receives its own winner declaration.

We extend `exnode` (Section 4.2.2) to make a benchmark-node with additional behavior: When a benchmark-node is confirmed as winner, it puts its own `ThreadId` into an `MVar` to signal termination.

```
benchNode :: MVar ThreadId → Intent Exnode SomeException
benchNode done state e@Envelope{message} = do
  state' ← exnode state e
  self ← myThreadId
  case fromException message of
    Just (Winner w) | w ≡ self → putMVar done w
    _ → return ()
  return state'
```

The reason we aren't using message passing to notify about termination is because it is difficult to communicate between the "actor world" and the "functional world." The functional world expects `IO` actions to terminate with return values, but we didn't bother to implement clean termination in our actor framework. Lacking that, we could try spawning an actor and then setting up an exception handler to receive

messages from it, but we choose not to do this because of the potential for race conditions.

We benchmark time to termination using the `criterion` package. For this, we need an `IO` action that executes the algorithm, cleans up its resources, and then returns. The function `benchActors` does this: it runs an election with benchmark-nodes, waits for termination, kills the nodes, and asserts a correct result.

```
benchActors :: Int → IO ()
benchActors n = do
  -- Start the ring-leader election
  done ← newEmptyMVar
  ring ← ringElection n $ do
    great ← myThreadId
    run (benchNode done) (Uninitialized, great)
  -- Wait for termination, kill the ring, assert correct result
  w ← takeMVar done
  mapM_ killThread ring
  assert (w ≡ maximum ring) (return ())
```

A.3 Control Benchmark Implementation

The experimental control, `benchControl`, only forks threads and then kills them. It is useful to establish whether or not, for example, laziness has caused our non-control implementations to perform no work. The other implementations should take longer than the control because they do more work.

```
benchControl :: Int → IO ()
benchControl n = do
  nodes ← sequence ∘ replicate n $ forkIO (return ())
  mapM_ killThread nodes
```

A.4 Channel Benchmark Implementation

Each node has references to a send-channel and a receive-channel in the channel-based implementation. We reuse the message types from before via an `Either`.

```
type ChMsg = Either Msg Winner
type Ch = Ch.Chan ChMsg
```

It is unnecessary to split the channel-based implementation into a simple node and an extended node, but we split them anyway to ease comparison to the actor-based implementation. This structural similarity hopefully has the added benefit of focusing benchmark differences onto the communication mechanisms instead of anecdotal differences.

In `chanNode` we implement the main loop. The only state maintained is the greatest nominee seen. It leaves off with definitions of communication functions in its where-clause.

```
chanNode ::
    MVar ThreadId → (Ch, Ch) → ThreadId → IO ()
chanNode done chans st = do
    chanNode done chans =≪ exnodePart st =≪ recv
  where
    recv = Ch.readChan (fst chans)
    sendMsg = Ch.writeChan (snd chans) ∘ Left
    sendWinner = Ch.writeChan (snd chans) ∘ Right
```

Within the where-clause of `chanNode`, we define `nodePart` to implement the behavior of a ring node from Section 4.1.2. This part has no state and requires no `Init` message.

```
nodePart :: Msg → IO ()
nodePart Start = do
  self ← myThreadId
  putStrLn (show self ++ ": nominate self")
  sendMsg $ Nominate self
nodePart (Nominate n) = do
  self ← myThreadId
  case () of
    _ | self ≡ n → putStrLn (show self ++ ": I win")
      | self < n →
        putStrLn (show self ++ ": nominate " ++ show n)
          ≫ sendMsg (Nominate n)
      | otherwise → putStrLn "Ignored nominee"
```

Still within the where-clause of `chanNode`, we implement `exnodePart` with the behavior of the winner-round node (Section 4.2.2) and the benchmark-node (Appendix A.2). It signals termination by placing the confirmed winner's `ThreadId` into an `MVar`.

```
exnodePart :: ThreadId → Either Msg Winner → IO ThreadId
exnodePart great (Left m) = do
  nodePart m
  self ← myThreadId
  case m of
    Nominate{nominee} →
      if self ≡ nominee
      then sendWinner (Winner self)
          ≫ return great
      else return $ max nominee great
    _ → return great
exnodePart great (Right m) = do
  self ← myThreadId
  case m of
    Winner w
      | w ≡ self →
        putStrLn (show self ++ ": Confirmed")
          ≫ putMVar done self
      | w ≡ great → sendWinner (Winner w)
      | otherwise → putStrLn "Unexpected winner"
  return great
```

Finally, we initialize the algorithm with a function similar to `ringElection`, but using channels instead of passing in `ThreadIds`. (1) Define a function to run a channel-node on

the "done" `MVar` and two provided channels. (2) Construct channels and a ring of un-evaluated nodes *in order*. (3) Finally permute the nodes and fork them out of order. Nodes are assigned random thread identifiers at this point. (4) Start the election. (5) Wait for termination and clean up.

```
benchChannels :: Int → IO ()
benchChannels n = do
  done ← newEmptyMVar
  let mkNode chans = do                                    (1)
    great ← myThreadId
    chanNode done chans great
  chans ← sequence ∘ replicate n $ Ch.newChan
  let nodeActs = map mkNode                                (2)
    (zip chans $ tail chans ++ [head chans])
  ringActs ← getStdRandom $ permute nodeActs
  ring ← mapM forkIO ringActs                              (3)
  mapM_ (λc → Ch.writeChan c ∘ Left $ Start) chans         (4)
  w ← takeMVar done                                        (5)
  mapM_ killThread ring
  assert (w ≡ maximum ring) (return ())
```

A.5 Criterion Benchmark Implementation

Finally, we define a benchmark-heat to run each of the benchmark functions defined above for a given ring size. The main function (not shown) calls `benchHeat` and passes it to Criterion's `defaultMain`.

```
benchHeat :: Int → Cr.Benchmark
benchHeat n = Cr.bgroup ("n=" ++ show n)
  [Cr.bench "control" ∘ Cr.nfIO $ benchControl n
  , Cr.bench "actor ring" ∘ Cr.nfIO $ benchActors n
  , Cr.bench "channel ring" ∘ Cr.nfIO $ benchChannels n]
```

A.6 Experimental Setup and Procedure

In all benchmarks, we replace printlines with `pure ()` to reduce noise and latency in results. We compile with the threaded RTS (`-threaded`) and run on all capabilities (`+RTS -N`). Our test machines included:

- MacBookPro11,5; 8 capabilities (NixOS).
- AWS `c3.8xlarge`; 32 vCPU (Amazon Linux 2023 AMI).
- AWS `c6a.48xlarge`; 192 vCPU (Amazon Linux 2023 AMI).

Our experiment proceeded as follows:

- We ran the `criterion` benchmark for ring sizes up to 16384 on the MacBookPro11,5, clocked to 1.6GHz, without frequency scaling, and with no other programs running (kernel vtty). Channels took a third the time of actors, but the performance gap narrowed as ring size increased.
- Figure 13a: We ran the benchmark on the AWS `c3.8xlarge` instance with 32 vCPU for ring sizes up to 65536. Actors outperformed channels at high ring sizes.
- Figure 14b: We ran the benchmark on the AWS `c6a.48xlarge` instance with 192 vCPU for ring sizes up to 65536. The benchmark segfaulted unpredictably. We used a shell script to call the benchmark executable once per set of parameters to work around segfaults. We confirmed that actors outperform channels at high ring sizes.
- Figure 14a: We repeated the benchmark on the MacBookPro11,5 for ring sizes up to 65536.
- Figure 13b: We ran a different benchmark to measure total-allocations (`+RTS -t --machine-readable`) on the MacBookPro11,5 for ring sizes up to 65536. Here the main function only ran a single algorithm at a specified ring size once, and then terminated. We ran ten trials for each combination of algorithm and ring size, and averaged.

A.7 Experiment Result

Our running time results for 8, 32, and 192 capabilities are in Figures 13a, 14a and 14b, respectively. We group the running time of the channel-based implementation over all three machines in Figure 15 to make its inflection point clearer. Our total-allocations result is in Figure 13b.

The running time of the extended ring leader election is $O(2n)$ in the number of nodes. We hypothesize that it is invariant to the number of capabilities because after an initial flood of nominations, the algorithm degenerates quickly to a single message passing around the ring twice.

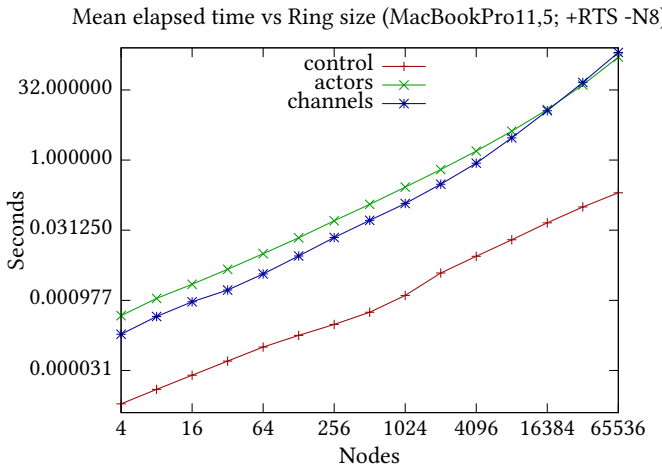
A.8 Actor-based (Dynamic Types) Trace

In Section 4.2.3, we showed how to call `runElection` on `exnode` to run a ring leader election with a winner declaration round. Here is an example trace.

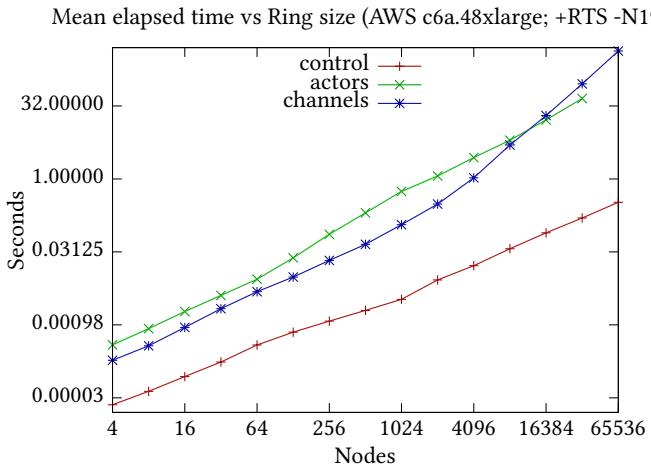
```
> main2 4
ThreadId 53 send Init {next = ThreadId 55} to ThreadId 57
ThreadId 53 send Init {next = ThreadId 56} to ThreadId 55
ThreadId 53 send Init {next = ThreadId 54} to ThreadId 56
ThreadId 53 send Init {next = ThreadId 57} to ThreadId 54
ThreadId 53 send Start to ThreadId 57
ThreadId 57 send Nominate {nominee = ThreadId 57} to ThreadId 55
ThreadId 53 send Start to ThreadId 55
ThreadId 55 send Nominate {nominee = ThreadId 57} to ThreadId 56
ThreadId 56 send Nominate {nominee = ThreadId 57} to ThreadId 54
ThreadId 55 send Nominate {nominee = ThreadId 55} to ThreadId 56
ThreadId 53 send Start to ThreadId 56
ThreadId 54 send Nominate {nominee = ThreadId 57} to ThreadId 57
Ignored nomination
ThreadId 56 send Nominate {nominee = ThreadId 56} to ThreadId 54
ThreadId 53 send Start to ThreadId 54
ThreadId 57: I win
ThreadId 57 send Winner (ThreadId 57) to ThreadId 55
ThreadId 54 send Nominate {nominee = ThreadId 56} to ThreadId 57
ThreadId 55 send Winner (ThreadId 57) to ThreadId 56
Ignored nomination
ThreadId 54 send Nominate {nominee = ThreadId 54} to ThreadId 57
ThreadId 56 send Winner (ThreadId 57) to ThreadId 54
Ignored nomination
ThreadId 54 send Winner (ThreadId 57) to ThreadId 57
ThreadId 57: Confirmed
```

A.9 Channel-based Extended Election Trace

In Appendix A.3 we defined `benchChannels` to run a ring leader election with a winner declaration round using channels for communication. Here's an example trace.



(a) Running time with 8 capabilities.



(b) Running time with 192 capabilities: The missing datapoint is a run that consistently crashed with a segmentation fault.

Figure 14. On different machines we replicate Figure 13a in both the absolute running time, and the actor-based implementation being faster at the highest ring sizes.

```
> benchChannels 4
ThreadId 61: nominate self
ThreadId 62: nominate self
Ignored nominee
ThreadId 64: nominate self
Ignored nominee
ThreadId 61: nominate ThreadId 64
ThreadId 63: nominate self
ThreadId 63: nominate ThreadId 64
ThreadId 62: nominate ThreadId 63
ThreadId 62: nominate ThreadId 64
Ignored nominee
ThreadId 64: I win
ThreadId 64: Confirmed
```

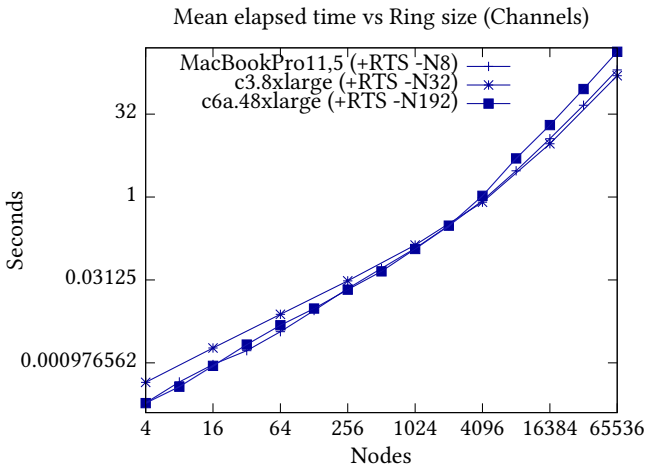


Figure 15. As ring size increases, the running time of `benchChannels` inflects to a higher rate around 2048 nodes.

References

[1] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph. D. Dissertation. http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-22455

[2] Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (May 1979), 281–283. https://doi.org/10.1145/359104.359108

[3] Taylor Fausak. 2022. *2022 State of Haskell Survey Results.* https://taylor.fausak.me/2022/11/18/haskell-survey-results/#s2q0

[4] GHC Contributors. 2021. *Control.Exception module, base package.* https://hackage.haskell.org/package/base-4.15.1.0/docs/Control-Exception.html#v:throwTo

[5] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[6] Simon Marlow. 2006. An Extensible Dynamically-Typed Hierarchy of Exceptions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA) *(Haskell '06)*. Association for Computing Machinery, New York, NY, USA, 96–106. https://doi.org/10.1145/1159842.1159854

[7] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. 2001. Asynchronous Exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 274–285. https://doi.org/10.1145/378795.378858

[8] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.* IOS Press, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/

[9] Patrick Redmond and Lindsey Kuper. 2023. *An Exceptional Actor System (Functional Pearl): Artifact.* https://doi.org/10.5281/zenodo.8162084 *Living.* https://github.com/lsd-ucsc/hakkell.paper/.

Received 2023-06-01; accepted 2023-07-04