The Wide World of Almost-Actors: Can I Have an Erlang Pony?

Code BEAM Stockholm
Friday, May 17, 2019
Scott Lystig Fritchie
Wallaroo Labs

#CodeBEAMSTO 2019
Introducing Myself

- I am Scott Lystig Fritchie
- Currently at Wallaroo Labs
- Formerly: VMware Research, Basho, Gemini Mobile, Caspian Networks, Sendmail, and UNIX sysadmin prior
  - 20 year Erlang anniversary
- @slfritchie at GitHub and Twitter
- I eat and cook a lot of Japanese food
Outline of the Talk

- BEAM $\Leftarrow$ Actors
  - False!
- Actor Model: defined & argued about
- Very brief overview to the Pony language
- 26 extra dimensions to the Actor Model
- Actor implementations: BEAM languages vs. Pony
My Goals For You

- Better understanding of what the Actor Model is
- Many dimensions to design & build an actor system
- BEAM & Pony are quite similar
  - … except where they aren’t
- Pony is interesting enough to learn more about
Happy Constitution Day!
BEAM := Actors
false
The Actor Model: 2019’s View

1. The actor is the fundamental unit of computation
2. An actor has its own state: registers, memory, etc.
3. An actor can read & modify only its own state
   - It is **private state**: no other actor has any access

1. An actor can react to a message that was sent to it
2. An actor can create a new actor
3. An actor can send a message to another actor

Message passing is the only communication mechanism between actors.
Let’s Look at Actor History
First (?) Paper About Actors

- Hewitt, Bishop, & Steiger, 1973

A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

Abstract

ACTOR architecture and definitional method focused on a single kind of object: actors [or, if s, or streams]. The formalism makes no presuppose data structures and control structures. Such and wired in a uniform modular fashion. In fact, even object is "really" represented as a list, a . The architecture will efficiently run the co l intelligence languages including those requir ncy is gained without loss of programming gener efficient; it does not change their behavioral general with respect to control structure and t, or semaphore primitives. The formalism aieved to achieve by other more structured methods.
Book About Actors

- Greif’s Ph.D. Thesis, 1975

SEMANTICS OF COMMUNICATING PARALLEL PROCESSES

by

IRENE GLORIA GREIF

S.B., Massachusetts Institute of Technology, 1969
S.M., Massachusetts Institute of Technology, 1972

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1975
Laws for Communicating Parallel Processes

by

Carl Hewitt and Henry Baker
Book About Actors

- Agha, 1986
Agha’s “Basic Constructs”

A program in an actor language consists of:

- **behavior definitions** which simply associate a behavior schema with an identifier (without actually creating an actor).\(^9\)

- **new expressions** which create actors.

- **send commands** which create tasks.

- **a receptionist declaration** which lists actors that may receive communications from the outside.

- **an external declaration** which lists actors that are not part of the population defined by the program but to whom communications may be sent from within the configuration.
The Actor Model differs from its predecessors and most current models of computation in that the Actor Model assumes the following:

- Concurrent execution in processing a message.
- The following are *not* required by an Actor: a thread, a mailbox, a message queue, its own operating system process, *etc.* \textsuperscript{iv}
- Message passing has the same overhead as looping and procedure calling.
- Primitive Actors can be implemented in hardware.\textsuperscript{i}

\textsuperscript{i} Denning 1973, 1975
\textsuperscript{iv} Note that a number of the following points are not true of current implementations of Actors.
IV. CELLS

One of the simplest examples of an actor which depends on its arrival ordering for correct behavior is the cell. The cell in actor theory is analogous to the program variable in modern high-level programming languages in that it has a value which can be changed through assignment. This value is encoded as the cell’s single, changeable acquaintance which is initialized to the name of some actor when the cell is created. A cell responds to two types of messages, “contents?” messages and “store!” messages. When a cell receives a request [contents? reply-to: c], the cell sends the name of its acquaintance to the actor c. When a cell receives a request [store! y reply-to: c], it memorizes new contents by making y its new acquaintance, forgetting its previous acquaintance, and then sending an acknowledge message to c.
III. LOCALITY

Information in an actor computation is intended to be transmitted by, and only by, messages. The most fundamental form of knowledge which is conveyed by a message in an actor computation is knowledge about the existence of another actor. This is because an actor A may send a message to another actor B only if it “knows about” B, i.e. knows B’s name. However, an actor cannot know an actor’s name unless it was either created with that knowledge or acquired it as a result of receiving a message. In addition, an actor cannot send a message to another actor conveying names he does not know. In the next section we give restrictions which enable actor computations to satisfy these intentions.
Locality and security mean that in processing a message: an Actor can send messages only to addresses for which it has information by the following means:

1. that it receives in the message
2. that it already had before it received the message
3. that it creates while processing the message.
Forbidden to (Real) Actors!

```
list_to_pid(String) -> pid()

Types

String = string()

Returns a process identifier whose text representation is a String, for example:

```
> list_to_pid("<0.4.1>").
<0.4.1>
```

Failure: badarg if String contains a bad representation of a process identifier.

Warning

This BIF is intended for debugging and is not to be used in application programs.
Erlang’s Other Differences with Actors

[erlang-questions] Erlang is *not* a implementation of the Actor model Re: Go vs Erlang for distribution

Peer Stritzinger <peerst@gmail.com>
Sun Jun 22 23:58:34 CEST 2014

- Previous message: [erlang-questions] Go vs Erlang for distribution
- Next message: [erlang-questions] Erlang is *not* a implementation of the Actor model Re: Go vs Erlang for distribution
- Messages sorted by: [date] [thread] [subject] [author]

On 2014-06-22 02:07:12 +0000, Miles Fidelman said:
> I see Erlang as an implementation of the Actor model, a la Carl Hewitt -

This crops up again and again but still isn’t true.
Arguing That Erlang /= Actors

- Selective receive reorders message delivery
- Process links that kill processes is very un-actor’ish
  - Monitors also
- Preemptive scheduling
- Actors must use Messaging exceptions, not die
  - without exception?
- No garbage collection of inactive actors
Robert Virding <rvirding@gmail.com>

Wed Jun 25 00:09:35 CEST 2014

- Previous message: [erlang-questions] Erlang is *not* a implementation of the Actor model
- Next message: [erlang-questions] Erlang is *not* a implementation of the Actor model
- Messages sorted by: [date] [thread] [subject] [author]

I think it is very lucky that we weren't interested in, or worried about, the theoretical aspects, or that we had heard about the actor model. If we had we would probably still be discussing whether we were doing the actor model and which parts of it, or where we differed and how important that was? Or should we differ and maybe we should drop the differences to we would comply, etc ... :-)

We were trying to solve *THE* problem and this was the best solution we could come with. It was purely pragmatic. We definitely took ideas from other inputs but not from the Actor model.

Robert
Agha’s “Insensitive Actor” can buffer messages while waiting for a particular message (page 54)
What is Pony?
Pony language & runtime safety guarantees

- Type system is fully aware of actors types & concurrency
- Type safe
- Memory safe
- Exception safe
- “If it compiles, it is data-race free.”
- All messaging is pass-by-reference
- Sharing data between actors is guaranteed safe
Pony compiles to target hardware CPU

- Erlang, Elixir, LFE, etc.
  - Runs on BEAM VM with optional compilation to native code via HiPE
- Pony
  - Compiles to native code via LLVM toolchain
  - DWARF symbols, “looks like C++” to debuggers and profilers
Side-Effect of Exception Safety

- Pony actors do not crash
- All errors must be handled explicitly
  - “?” syntax used to mark a "partial function"
    - "partial" = "may raise an error"
- Compiler enforced, of course
- No actor crashes => no (?) need for BEAM’s links & monitors to help manage failure
Per-Actor Heaps + Distributed GC

- Distributed GC across all actor heaps
  - No "stop the world" GC
  - Fully concurrent: no sync, no locks, and no barriers (except as needed for lock-free data structures)
- Message passing maintains ref counts on shared objects
  - Dead objects are reaped by allocating actor
- GC and Type System **Co-Designed** with ORCA protocol
  - Actors are 1st class, GC’ed objects in the system
  - Runtime halts when all actors are GC’ed
Can I have an Erlang Pony?
Let's get more specific about what an actor implementation might really need
Actor’ish Design & Implementation Topics

1. Message sending
   - Synchronous? Named? Typed? …
2. Message receiving
3. Scheduling
4. Message delivery guarantees
   - Academic distributed systems people want to know!
BEAM languages vs. Pony

26 Dimensions of Actor-Flavored Models
Message Sending

Synchronous vs. Asynchronous message sending

- BEAM: async
- Pony: async

SAME
Message Sending

Named Processes vs. Unnamed Processes

- BEAM: named
- Pony: named
Message Sending

What is a Message’s Destination?

• BEAM: one process
• Pony: one actor
Message Sending

Typed vs. Untyped Messages

- BEAM: untyped
- Pony: typed

WHOA!
How does data appear in a mailbox?

- BEAM: copy to destination heap
- Pony: ref-counted pointers + distributed GC via ORCA protocol
Message Receiving

Reliable vs. Unreliable Delivery

• BEAM: reliable’ish
• Pony: reliable
Message Receiving

Message delivery order

• BEAM: any order
• Pony: FIFO only
Message Receiving

Causal message order guarantee

• BEAM: yes or no
• Pony: yes always
Message Receiving

Blocking vs. Non-Blocking message receive

- BEAM: yes
- Pony: no

WHOA!
Message Receiving

Time-Aware vs. Time-Ignorant

• BEAM: yes
• Pony: no

WHOA!
Scheduling

What schedules actors?

• BEAM: custom scheduler
  • 1 scheduler/CPU core
• Pony: custom scheduler
  • 1 scheduler/CPU core
Scheduler Overhead

- BEAM: {100’s} bytes/process, {few} usec to create & destroy
- Pony: 240 bytes/actor, {few} usec to create & destroy
- Scheduling millions is fine
- Processes & Actors are cheap
Scheduling

Preemptive vs. Cooperative Scheduling

- BEAM: Preemptive
- Pony: Cooperative

WHOA!

#CodeBEAMSTO 2019
Scheduling

Actor priority schemes?

• BEAM: Yes, 4 levels
• Pony: No

WHOA!

#CodeBEAMSTO 2019
Scheduling

Work stealing?

- BEAM: Yes
- Pony: Yes

SAME
Scheduling

Energy Conservation by Idle Schedulers?

• BEAM: Yes
• Pony: Yes
Scheduling

Mailbox size limits?

• BEAM: No
• Pony: No

SAME
Scheduling

Maximum Heap Size?

- BEAM: No
- Pony: No

SAME
Scheduling

- Actor Lifecycle
  - Cheap vs. Cheap *SAME*
- Actor Crash?
  - Yes vs. No

#WHOA!
Scheduling

Back-pressure to reduce workload of overloaded actors?

- BEAM: No
- Pony: Yes

WHOA!

#CodeBEAMSTO 2019
Theoretical Message Delivery Properties

- Causal order: Yes
  - *SIMILAR*
- Message loss: 0%
  - *SAME*
- Message duplication: 0%
  - *SAME*
- Message reordering: *WHOA!*
Actors & the Outside World

Actor interaction with non-actors

• BEAM: yes
• Pony: yes, but…
Byzantine Actors

Incorrect/Malicious Actors/Corrupted Messages Allowed?

• BEAM: No
• Pony: No

SAME
Review of Similarities by Category

- SAME
  - 13
- SIMILAR
  - 5
- WHOA!
  - 9
WHOA! Summary

- Msg Receiving: message reordering
- Msg Receiving: blocking vs. non-blocking receive
- Msg Receiving: time-aware vs. time-ignorant
- Scheduling: preemptive vs. cooperative scheduling
- Msg Sending: untyped vs. typed messages
- Msg Sending: copy messages vs. shared pointers
- Scheduling: actor priority schemes?
- Lifecycle: actors crash?
- Back-pressure for "overloaded" actors?
In Pony, one does not simply call() a gen_server ever.
In Pony, one does not simply call() a gen_server.

you cannot block awaiting for the reply.
In Pony, all messaging is !()-style or cast-style
... but it’s possible to work around. Pony is fun!

Did I mention that Pony programs are usually really, really fast?
Sources & Where to Look For More

On the Actor Model and CSP:
• https://en.wikipedia.org/wiki/Actor_model
• https://en.wikipedia.org/wiki/Communicating_sequential_processes

On Pony:
• http://ponylang.io (also Pony logo source)
• https://github.com/ponylang/ponyc/
• http://blog.acolyer.org/2016/02/17/deny-capabilities/
• https://www.youtube.com/watch?v=e0197aoljGQ

Source of microbenchmark graphs:
S Clebsch, J Franco, S Drossopoulou, AM Yang, T Wrigstad, J Vitek
“Orca: GC and type system co-design for actor languages”. Proceedings of the ACM on Programming Languages 1 (OOPSLA), 72

Sean Bean image:
http://knowyourmeme.com/memes/one-does-not-simply-walk-into-mordor
https://memegenerator.net/Does-Not-Simply-Walk-Into-Mordor-Boromir
Overflow slides
ORCA GC Comparison on μB’marks

Fig. 17. Responsiveness. X-axis: request ID, Y-axis: Jitter/difference between finishing time (seconds) of subsequent requests. Java measurements are from a warmed-up VM and does not include JIT’ing.

#CodeBEAMSTO 2019
ORCA GC Comparison on μB’marks

(c) rings

Orca: GC and Type System Co-Design for Actor Languages

(d) mailbox

Time (microsec)

Orca  Erlang  C4

1e7  8  16  32  64  4  8  16  32  64  4  8  16  32  64

mutator time  mutator overhead  concurrent gc  stw gc

Fig. 16. Strong scalability on 4–64 cores. (stw=stop-the-world.)
ORCA GC Comparison on \(\mu\)B’marks

Fig. 16. Strong scalability on 4–64 cores. (stw=stop-the-world.)
Pony Is Not a Functional Language

- Pony is very imperative
- … but the type system provides lovely safety properties
Pony Has Lambdas & More

- lambdas / unnamed functions
- map() & friends, hooray
- persistent data structures in the standard library
Pony Is Object-Oriented

- … but not Java-style
- Not **everything** is an object
  - You control the class hierarchy
- Has both structural & nominal subtyping
  - Pony’s **interface** = structural typing
Pony Has Generic Types

// map over a List[A] to // create a List[B]

fun box map[B: B](
    f: {(this->A!) : B}^[A, B] box
) : List[B] ref^
Pony Has Pattern Matching!

- `match` statement to match:
  - basic data types
  - sub-/super-types in class hierarchy
  - tuple element destructuring
- Function head matching is gone
  - … but will return again soon (I hope)
Pony Is Open Source

- BSD-style license
- https://github.com/ponylang/ponyc/
- Target CPUs
  - x86_64, ARM
- Target operating systems:
  - Linux, Windows, OS X
  - FreeBSD & DragonflyBSD (limited support)
- "Get Stuff Done" development model
  - Correctness > Performance > Simplicity > Consistency > Completeness
Pony Is Young

- The standard library is small
- The open source community is small
- Ecosystem of Pony language libraries & apps is small
Pony’s FFI to C/C++ ABI

- Easily interface to C & C++ ABI functions
- Runtime's requirements for memory & threads are modest
- Many Pony primitive data types map directly to target CPU
  - I8, I16, I32, I64, I128
  - U8, U16, U32, U64, U128
  - Array[U8] for contiguous unstructured bytes
Pony's Reference Capabilities

- Strong, static type checker is the price to pay for safety
- It’s a big mind shift to adjust to both:
  - Mutable data (even if it is safe!)
  - Pony’s type system (based on affine types)
- The end advantages:
  - Zero runtime cost for safety
  - Very quick GC
Get Involved!

- Web: [http://ponylang.org](http://ponylang.org)
- GitHub: [https://github.com/ponylang/ponyc/](https://github.com/ponylang/ponyc/)
- Twitter: @ponylang
- Freenode IRC: #ponylang
- Mailing list info: [https://pony.groups.io/g/user](https://pony.groups.io/g/user)
- Pester me about Erlang, Pony, and/or Wallaroo:
  - Anytime here at the conference
  - @slfritchie on Twitter
  - slfritchie@ on gmail.com