Pony for Fintech

or

How I Stopped Worrying and Learned to Love an Exotic Product

Sylvan Clebsch

QCon London 2016

What is Pony?

Pony is an actor-model capabilities-secure native language

@ponylang

Freenode: #ponylang

http://ponylang.org

What do I want you to walk away with?

Pony has a powerful, data-race free, concurrency-aware type system

The Pony runtime can help you solve hard concurrency problems

You might be able to use Pony for fintech in production now

My talks are usually a bit...

$$\frac{\mathtt{x} \in \Gamma}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \textsc{T-Local}$$

$$\frac{\mathtt{S} \in \mathtt{P}}{\Gamma \vdash \mathtt{null} : \mathtt{S}\,\mathtt{iso}\circ}\ \textsc{T-Null}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{S}\,\kappa}{\Gamma \vdash \mathtt{x} = \mathtt{e} : \mathcal{U}(\mathtt{S}\,\kappa)}\ \textsc{T-AsnLocal}$$

$$\frac{\mathcal{M}(\mathtt{S}, \mathtt{m}) = (\mathtt{T}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{ET}) \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T} \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{ET}}\ \textsc{T-Sync}$$

$$\frac{\mathcal{M}(\mathtt{C}, \mathtt{k}) = (\mathtt{C}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{C}\,\mathtt{ref}\circ) \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}}{\Gamma \vdash \mathtt{C.k}(\overline{\mathtt{e}}) : \mathtt{C}\,\mathtt{ref}\circ}\ \textsc{T-Ctor}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{ET}' \quad \mathcal{A}(\mathtt{ET}') \leq \mathtt{T}}{\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T}}\ \textsc{T-Alias}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa\circ}{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa}\ \textsc{T-Subsume}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \quad \mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'}{\Gamma \vdash \mathtt{e.f} : \mathtt{S}'\,\kappa \triangleright \kappa'}\ \textsc{T-Fld}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{ET} \quad \Gamma \vdash \mathtt{e}' : \mathtt{ET}'}{\Gamma \vdash \mathtt{e}; \mathtt{e}' : \mathtt{ET}'}\ \textsc{T-Seq}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e}' : \mathtt{S}'\,\kappa' \quad \mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'' \quad \kappa' \leq \kappa'' \quad \vdash \kappa \triangleleft \kappa' \vee \vdash \kappa \triangleleft \kappa''}{\Gamma \vdash \mathtt{e.f} = \mathtt{e}' : \mathcal{U}(\mathtt{S}'\,\kappa \triangleright \kappa'')}\ \textsc{T-AsnFld}$$

$$\frac{\mathcal{M}(\mathtt{A}, \mathtt{b}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{A}\,\mathtt{tag} \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}}{\Gamma \vdash \mathtt{e.b}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag}}\ \textsc{T-Async}$$

$$\frac{\mathcal{M}(\mathtt{A}, \mathtt{k}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}}{\Gamma \vdash \mathtt{A.k}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag}}\ \textsc{T-Ator}$$

$$\frac{\Gamma \backslash \{\mathtt{x} \mid \neg \mathit{Sendable}(\Gamma(\mathtt{x}))\} \vdash \mathtt{e} : \mathtt{ET}}{\Gamma \vdash \mathtt{recover}\ \mathtt{e} : \mathcal{R}(\mathtt{ET})}\ \textsc{T-Rec}$$

Today is more...

```
bool messageq_push(messageq_t* q, pony_msg_t* m)
{
  m->next = NULL;

  pony_msg_t* prev = (pony_msg_t*)_atomic_exchange(&q->head, m);

  bool was_empty = ((uintptr_t)prev & 1) != 0;
  prev = (pony_msg_t*)((uintptr_t)prev & ~(uintptr_t)1);

  _atomic_store(&prev->next, m);

  return was_empty;
}
```

Some fintech code "asset classes"

High Frequency Trading

Risk Engines

Trade Surveillance

Aggregate Limits Checks

Matching Engines (FX, Dark Pool, Exchange, etc.)

Pricing Engines

What are the common elements?

Mostly Java or C++

Niche languages: Scala, OCaml, Erlang, C, R, NumPy

Performance critical

Not formally verified

What is "performance"?

Latency?

Throughput?

Uptime?

Graceful degradation?

Hit rate?

Best execution?

Profit?

Performance = Profit

Fintech has a narrow world view, but it gives a precise definition of performance

Is speed more important than correctness?

Trick question!

When performance is profit, fast and correct are both aspects of performance

What do we want?

Tools that help us exploit our domain knowledge to write profit-maximising systems

To maximise profit, tools must be as fast and as correct as possible

Some examples:

- Aeron
- Cap'n Proto
- OpenOnload
- 29West LBM

How do those tools help?

Each tool manages a collection of hard problems

They aren't single purpose mini-libraries

They completely encapsulate the solution

They are not (necessarily) drop-in replacements for existing code

A programming language is just another tool

It's not about syntax

It's not about expressiveness

It's not about paradigms or models

It's about managing hard problems

Using Pony to manage hard problems

Actor-model concurrency

Powerful, concurrency-aware type system

Zero-copy fast message queues

Work-stealing topology-aware scheduling

Fully concurrent GC with no stop-the-world

Example: a naive order manager

This took *nearly an hour* to write

Gee whiz, that's a lot of hard work

Actor-model concurrency

An object combines state-management with synchronous methods (functions)

An actor combines state-management with asynchronous methods (behaviours)

What does an actor have that an object doesn't?

In addition to fields, an actor has a message queue and its own heap

Actor heaps are independently garbage collected, and the actors themselves are garbage collected

It's ok to have millions of actors in a program (they're cheap)

What can actors do?

1. Handle the next message in their queue
2. Modify their own state
3. Create other actors
4. Send messages to actors

Pony actors

In Pony, actors have no blocking constructs

They handle each message by executing a behaviour

That behaviour is *logically atomic*: it cannot witness any heap mutation it does not itself perform

All communication between actors is by message passing

Order

```
actor Order
  embed _state: OrderState
  let _exch_conn: ExchangeConn
  embed _exch_order: ExchangeOrder
  embed _observers: MapIs[OrderObserver tag, OrderObserver] =
    _observers.create()
```

These are just the fields: some state, an exchange connection (all traffic about an order needs to go over the same connection), and a collection of observers.

How cheap is an actor?

240 byte overhead vs. an object (156 on a 32-bit architecture)

No CPU time overhead: only on a scheduler queue when there is work to be done

Order Information

```
class val OrderInfo
  let client_id: String
  let instrument: String
  let side: Side
  let qty: U32
  let price: U32

  new val create(client_id': String, instrument': String, side': Side,
    qty': U32, price': U32)
  =>
    client_id = client_id'
    instrument = instrument'
    side = side'
    qty = qty'
    price = price'
```

A simple immutable data structure

The *type* isn't immutable, but this particular constructor returns immutable objects

Order State

```
class OrderState
  embed info: OrderInfo
  embed fills: Array[Fill] = fills.create()
  var cum_qty: U32 = 0
  var status: Status = PendingNew

  new create(client_id: String, instrument: String, side: Side,
    qty: U32, price: U32)
  =>
    info = OrderInfo(client_id, instrument, side, qty, price)
```

Mutable order state, notice the embedded `OrderInfo` and the generic `Array[Fill]`

Creating an Order

```
new create(exch_conn: ExchangeConn, info: OrderInfo,
  observers: (ReadSeq[OrderObserver] iso | None) = None)
=>
  _state = OrderState(info)
  _exch_conn = exch_conn
  _exch_order = ExchangeOrder(this, info)

  try
    let obs = consume observers as ReadSeq[OrderObserver]

    for observer in obs.values() do
      _observers(observer) = observer
      observer.initial_state(_state)
    end
  end

  _exch_conn.place(_exch_order)
```

There's a lot going on here!

Notice the isolated (`iso`) readable sequence (`ReadSeq`) of order observers: `ReadSeq[OrderObserver] iso`

It's in a union type with `None` and defaults to `None`

Adding and removing observers

```
be observe(some: OrderObserver iso) =>
  let some' = consume ref some
  _observers(some') = some'
  some'.initial_state(_state)

be unobserve(some: OrderObserver tag) =>
  try
    (_, let some') = _observers.remove(some)
    some'.dispose(_state)
  end
```

To add an observer, we send the order a message (`be` for behaviour)

The argument is an isolated order observer `OrderObserver iso`

To remove an observer, we send the identity of the observer to remove, as an `OrderObserver tag`

iso, val, tag, what?

These type annotations are called **reference capabilities** (rcaps)

Rcaps make Pony data-race free

And the data-race freedom guarantee is exploited in the runtime

Powerful, concurrency-aware type system

Reference capabilities (rcaps) for data-race freedom

Nominal (trait) and structural (interface) typing

Algebraic data types (union, intersection, tuple)

Generics and parametric polymorphism

Fully reified types

No "null"

No uninitialised or partially initialised values

Weak dependent types are on the way

But this isn't a type system talk

Let's have a brief look at reference capabilities (rcaps)

Because it's the most novel part of the type system

Rcaps for data-race freedom

Pony uses reference capabilities (rcaps) to express isolation and immutability

These guarantee at compile time that your program is data-race free

Rcaps are type annotations

Every reference to an object indicates a level of isolation or immutability

```
x: Foo iso // An isolated Foo
x: Foo val // A globally immutable Foo
x: Foo ref // A mutable Foo
x: Foo box // A locally immutable Foo (like C++ const)
x: Foo tag // An opaque Foo
```

This affects how you can use a reference (read, write, alias, send)

Rcaps are about *deny*, not *allow*

Rather than grant permissions, an rcap prevents certain kinds of other rcaps to the same object

This is used to enforce the golden rule of data-race freedom:

"If I can write to it, nobody else can read from it"

Adrian's write up

the morning paper

Deny Capabilities for Safe, Fast Actors

http://blog.acolyer.org/2016/02/17/deny-capabilities/

Adrian's local and global rcap compatibility chart


local
global
read alias
write alias
read alias
write alias
tag alias
iso
trn
ref
val
box
tag
OR

Rcaps use local typing, not global analysis

Data-race freedom using rcaps is handled by the compiler during type checking

There is no global static analysis step

Ensuring data-race freedom doesn't get harder for the compiler as your code base grows

This is one part of *composable concurrency*

If it compiles, it's data-race free

No locks, no deadlocks

No memory model issues

No concurrency composition problems

No runtime overhead

Back to the Order: getting a fill

```
be fill(some: Fill) =>
  _state.fills.push(some)
  _state.cum_qty = _state.cum_qty + some.qty

  _state.status = match _state.status
  | New if _state.cum_qty >= _state.info.qty => Filled
  | PartiallyFilled if _state.cum_qty >= _state.info.qty => Filled
  | PendingNew | Rejected => Invalid
  else
    _state.status
  end

  for observer in _observers.values() do
    observer.fill(_state, some)
  end
```

When a `Fill` is received, the order state is updated

Observers are notified of the fill and the new order state

This is a behaviour (`be`), so it happens asynchronously

An observer might be synchronous or asynchronous

```
interface OrderObserver
  fun ref initial_state(order: OrderState box) => state_change(order)
  fun ref state_change(order: OrderState box) => None
  fun ref fill(order: OrderState box, some: Fill) => None
  fun ref dispose(order: OrderState box) => None
```

An `OrderObserver` is a strutural type: anything that has these methods is an `OrderObserver`

An observer may execute code synchronously, send asynchronous messages, or both

How cheap are asynchronous messages?

If we have millions of actors, we will have many millions of messages

Messaging has to be cheap

And it has to stay cheap under load

Use cheap messages to move the code to the data

```
interface val OrderReport[A: Any #share]
  fun apply(order: OrderState box): A ?

actor Order
  be report[A: Any #share](generator: OrderReport[A], promise: Promise[A])
    try
      promise(generator(_state))
    else
      promise.reject()
    end
```

We send a report generating function to the order and either fulfil or reject a promise with the generated report, instead of asynchronously asking for the order state

Zero-copy fast message queues

Intrusive unbounded MPSC queue with integral empty queue detection

Intrusive: messages do not need to be in more than one queue

Unbounded: controversial!

Why unbounded?

If the queue was bounded, what would we do when the queue was full?

Block or fail?

Blocking can deadlock and introduce unbounded producer jitter

Failing means application-specific failure handling every time a message is sent

Unbounded queues move the back pressure problem

They don't make it better

They don't make it worse

We'll return to back pressure later

Pushing a message on to a queue

```
bool messageq_push(messageq_t* q, pony_msg_t* m)
{
  m->next = NULL;

  pony_msg_t* prev = (pony_msg_t*)_atomic_exchange(&q->head, m);

  bool was_empty = ((uintptr_t)prev & 1) != 0;
  prev = (pony_msg_t*)((uintptr_t)prev & ~(uintptr_t)1);

  _atomic_store(&prev->next, m);

  return was_empty;
}
```

A single atomic operation, wait-free, integrated empty queue detection

The `_atomic_store` doesn't need an atomic op on X86 or ARM

Popping a message off of a queue

```
pony_msg_t* messageq_pop(messageq_t* q)
{
  pony_msg_t* tail = q->tail;
  pony_msg_t* next = _atomic_load(&tail->next);

  if(next != NULL)
  {
    q->tail = next;
    pool_free(tail->size, tail);
  }

  return next;
}
```

Zero atomic operations, stub nodes for memory management

The `_atomic_load` doesn't need an atomic op on X86 or ARM

Is this the best possible queue?

Probably not!

This is an area with lots of research and engineering happening

This particular queue relies on specifics of the Pony runtime to be efficient

Particularly the underlying memory allocator

Zero-copy messages

The type system guarantees the program is data-race free

Messages can be passed by reference instead of being copied

Without locking or atomic reference counting

Workstealing topology-aware scheduling

Actors encapsulate composably concurrent message handling

But how are they scheduled?

Scheduler threads

The Pony runtime can be started with any number of scheduler threads

Defaults to the physical core count

The program doesn't have to be aware of the thread count

What's does a scheduler thread consist of?

1. A kernel thread pinned to a core
2. A single-producer multiple-consumer queue of actors with pending messages
3. A message queue for quiescence detection
4. A pool allocator

SPMC Queue

Similar to the MPSC message queue

Unbounded, non-intrusive

Zero atomic operations to push an actor on to the queue

CAS loop to pop an actor off of the queue

Pushing an actor on to a scheduler queue

```
void mpmcq_push_single(mpmcq_t* q, void* data)
{
  mpmcq_node_t* node = POOL_ALLOC(mpmcq_node_t);
  node->data = data;
  node->next = NULL;

  // If we have a single producer, don't use an atomic instruction.
  mpmcq_node_t* prev = q->head;
  q->head = node;
  prev->next = node;
}
```

Hey, that says MPMC!

Yeah, the same code with a different push function is used for an MPMCQ

A runtime has only one MPMCQ (the inject queue) and we won't cover it

Popping an actor off of a scheduler queue

```
void* mpmcq_pop(mpmcq_t* q)
{
  mpmcq_dwcas_t cmp, xchg;
  mpmcq_node_t* next;

  cmp.aba = q->tail.aba;
  cmp.node = q->tail.node;

  do
  {
    // Get the next node rather than the tail. The tail is either a stub o
    // already been consumed.
    next = _atomic_load(&cmp.node->next);

    // Bailout if we have no next node.
    if(next == NULL)
      return NULL;
```

So that's a bit more complicated

Message Batching

A scheduler thread executes a batch of messages on an actor

This is to amortise the cost of popping an actor off a scheduler queue

Today this is a fixed number, in the future it will be dynamic

Why are scheduler queues SPMC?

When an actor is sent a message...

- If the actor's message queue wasn't empty
  - Then it is already on some scheduler thread: do nothing
- If the actor's message queue was empty
  - Then the actor wasn't scheduled anywhere
  - Schedule it on the queue of the thread that is sending the message
  - Because the actor's working set probably isn't in cache
  - So get cache locality for the message contents, if possible

Only the scheduler thread ever puts an actor on its own queue

That's the SP, what about the MC?

If a scheduler thread has no actors on its queue, it picks another scheduler thread and tries to steal an actor

This is why we need MC

"Near" cores are mildly preferred

Because cache and NUMA

But the preference isn't too strong, to clustering and spending too much time trying to steal from scheduler threads with no available work

Actors with pending messages tend to stay on the same core

It's more important to use all available cores than to keep an actor always on the same core

A scheduler thread never gives up its only available actor, to avoid ping-pong scheduling

Once a core has work, it won't steal any more unless it runs out of work

Because cache and NUMA

Quiescence detection

Scheduler threads are actually actors

They use a quiescence detection messaging protocol to terminate a program

The protocol is I/O aware, and can correctly terminate a program before all actors are garbage collected

A program has terminated when it is not listening for input, there are no pending messages, and no new messages will ever be generated

Generalised back pressure

VAPOURWARE ALERT

When an actor sends a message to an actor with a loaded message queue, its scheduler priority should drop

Back pressure questions

What is a "loaded" message queue and how can it be cheaply determined? Is it as simple as a non-empty queue?

What is a scheduler priority? Not a priority queue (too slow)

Use a separate list of "deprioritised" actors?

Is influencing the message batch count enough?

Should a scheduler thread with only "deprioritised" actors try to steal work?

Fully concurrent GC with no stop-the-world

GC'ing an actor heap

GC'ing things shared across heaps

GC'ing actors themselves

GC'ing an actor heap

Actors GC their own heaps independent of other actors

Mark-and-don't sweep algorithm

O(n) on reachable graph

Unreachable memory has no impact

What actor heap GC doesn't need

No safepoints

No read or write barriers

No card table marking

No compacting, so no pointer fixups

GC'ing objects shared across heaps

Rcaps allows isolated (iso), immutable (val), and opaque (tag) objects to be safely passed by reference (zero-copy) between actors

Objects in messages or reachable by other actors must be protected from premature collection

Use a message protocol

Deferred distributed weighted reference counting

The shape of the heap does not influence the reference count

Trace objects reachable from a message on send and receive

No round trips or acknowledgement messages

Actors can still independently collect heaps without coordination

Shared object GC is a distributed system

An actor can't know what other actors can reach an object it has allocated

Or what undelivered messages the object may be reachable from

The object must not be collected prematurely but must be collected in a timely manner

Teaser: this has been extended to the distributed runtime

Immutability and tracing

Isolated (iso) objects must trace their fields in messages

Because they can be mutated by an actor other than the allocator

Immutable (val) objects need not be traced, either in message or by any non-allocating actor when reached during heap GC

Because the allocator can safely be responsible for tracing an immutable graph

Sharing an immutable object graph amongst N actors does not increase aggregate GC times

Adrian's write up

Ownership and Reference Counting Based Garbage Collection in the Actor World

http://blog.acolyer.org/2016/02/18/ownership-and-reference-counting-based-garbage-collection-in-the-actor-world/

GC'ing actors

An extension of the same protocol allows actors themselves to be GC'd

It's an even more complex distributed system

A cycle detector (implemented as a system actor) is required

Plus a CNF-ACK protocol that can cheaply determine that a view of an actor's topology received in the past was a true view of the global state when it was recieved

Further work

VAPOURWARE ALERT

Distributed Pony

Weak dependent types

Reflection

Meta-programming

Hot code loading

REPL

Come get involved!

Join our community of industry developers, tooling developers, library writers, compiler hackers, type system theorists, students, and hobbyists.

Why not you?

@ponylang

Freenode: #ponylang

http://ponylang.org