Mid-term paper for Programming Theory class

Ogiwara

November 25, 2019

1 Objective

Pick one programming language, and explain about that. Here, I explain about Pony language [1] through descriptions of features, performance, sample codes, and reference capability type system.

1.1 Why I chose Pony

This is really boring reason. I just seek "200 minor programming language list"[3], and I found this.

2 Features and Performance

2.1 Simple introduction from official page [1]

Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language.

Pony is type safe. Really type safe. On top page, there is a link for mathematical proof paper [2]. I will explain about that in later section.

Pony is memory safe. There are no dangling pointers and no buffer overruns. The language doesn't even have the concept of `null`.

Pony is exception-Safe. There are no runtime exceptions. All exceptions have defined semantics, and they are always caught.

Pony is data-race free. Pony doesn't have locks nor atomic operations or anything like that. Instead, the type system ensures at compile time that your concurrent program can never have data races. So you can write highly concurrent code and never get it wrong.

Pony is deadlock-free. This one is easy because Pony has no locks at all. So they definitely dont deadlock, because they don't exist.

2.2 Example Code

```
use "time"

actor Main
    """
    Timer program.
    Sleep for 5 seconds, and repeat to run Notify
    .apply method by 2 seconds.
    """
    new create(env: Env) =>
        let timers = Timers
        let timer = Timer(Notify(env), 5
    _000_000_000, 2_000_000_000)
        timers(consume timer)

class Notify is TimerNotify
  let _env: Env
  var _counter: U32 = 0

  new iso create(env: Env) =>
    _env = env

  fun ref apply(timer: Timer, count: U64): Bool
   =>
    _env.out.print(_counter.string())
    _counter = _counter + 1
    true
```

timer.pony

Here is timer program in Pony. At first sleeps for 5 seconds. After that, repeat to run Notify.apply method in every 2 seconds.

There are `Main` actor and `Notify` class. `Main` actor has `new create` symbol, which works as constructor. `Notify` class has `_env`, `_counter` fields, `create` constructor, and `apply` method.

First, `Main` actor's `create` constructor is called. It is initial function as same as `int main()` in C or `public static void main(String[] args)` in Java.

Values are assigned into `timers` and `timer`, and then `Timers.apply` method is called. At here, `Timers` is another actor, so we have to move `timer` data to the actor by `consume` expression.

`Timers` actor calls passed object(At here `Notify`)'s apply method repeatedly (At here sleeps for 5 seconds run every 2 seconds).

`Notify.apply` outputs current `_counter`, and then increment it.

2.3 Compare to other languages

```
defmodule Actor1 do
  def call() do
    ...
  end
end

GenServer.start_link(Actor1, [:call])
```

Make actor in Elixir

To use actor, Elixir have to define module, and make actor by specifying both module name and method name.

```
actor Actor1
  be call() =>
    ...

let actor1 = Actor1
```

Make actor in Pony

However in Pony, `actor` is primitive syntax, so you can just make instance of actor as same as classes.

```
let a = String::new("hello")
let b = a
// You can't use a at here anymore.
```

ownership system in Rust

```
let a : String iso = "hello"
let b = consume a
// You can't use a at here anymore.
```

consuming in Pony

Pony has much more stronger reference capability system than Rust's ownership system.

At here, `iso` is one of reference capability types, which mean "this value is readable and writable in one actor, and it can move to another actor".

`iso` has *read and write uniqueness*, which means only single variable can bind its reference, and you have no way to read and write data without using the varibale.

```
let a = "not iso"
let b = a // Aliasing

let c: String iso = "iso"
let d = c // error! iso has read and write
    uniqueness, so it cannot make alias.
let e = consume c // it works. move reference to
    left-hand variable.
```

read and write uniqueness

At here, `b` has copied reference from `a`, which is called *aliasing* in Pony. However if a variable have some *uniqueness*, you cannot make alias.

About reference capability, I will explain about it in later section.

```
class A:
  def b():
    ...
```

Python indentation

```
class A
  fun b() =>
    ...
```

Pony indentation

Pony uses indentation as block as same as Python.

```
for i in values do

end
```

do end block style both in Ruby and Pony

Pony uses do~end style block as same as Ruby.

```
class A[T]{
  def apply(i: T){}
}

val a: A[Int] = new A()
a(8)
```

Type parameter and apply method in Scala

```
class A[T]
  fun apply(i: T) =>
    ...

let a: A[Integer] = A
a(8)
```

Generics and apply method in Pony

Pony uses square brackets([T]) for Generics, and functional programming style `apply` method as same as Scala.

```
type geometry interface {
    area() float64
    perim() float64
}

type rect struct {
    width, height float64
}

func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
// Now type rect implements geometry interface.
```

interface in Go

```
interface Animal
  fun bark(): String

class Dog
  fun bark(): String
    => "Bow!"

// Now type Dog implements Animal interface.
// You can annotate Dog implements Animal by
// `class Dog is Animal`
```

interface in Pony

Pony has structural subtyping as same as Go.

About parallel processing programming, Pony has variety of variable types and strong type system for actor model. Table 1 is summary of some parallel processing programming features of Pony and compare with Scala(Akka streams[4]), Elixir, and Rust.

2.4 Performance

Figure 1 and Figure 2 are actor performance comparison graph with Erlang, Scala, CAF[5], Charm++, and Pony (These graphs are from the paper[2]). Figure 1 shows

	Pony	Scala(Akka)	Elixir	Rust
Zero-Copy	✓	✓		✓
Data-race free	✓		✓	✓
Statically data-race free	✓			
Read unique (iso)	✓			
Write unique (trn)	✓			
Mutability (ref)	✓	✓		✓
Immutability (val)	✓		✓	✓
Identity (tag)	✓			
Actors	✓	✓	✓	
Formal proof	✓			
Concurrent GC			✓	

Table 1: Feature comparison table from[2].

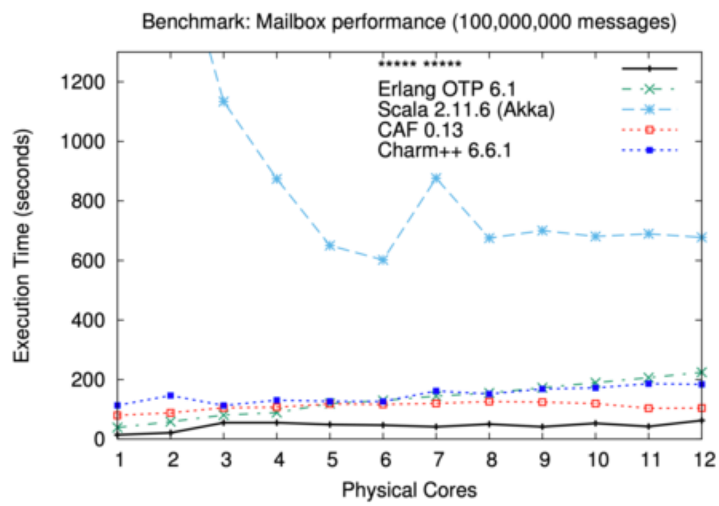

Figure 1: Actor creation, where **** is Pony(from[2]).

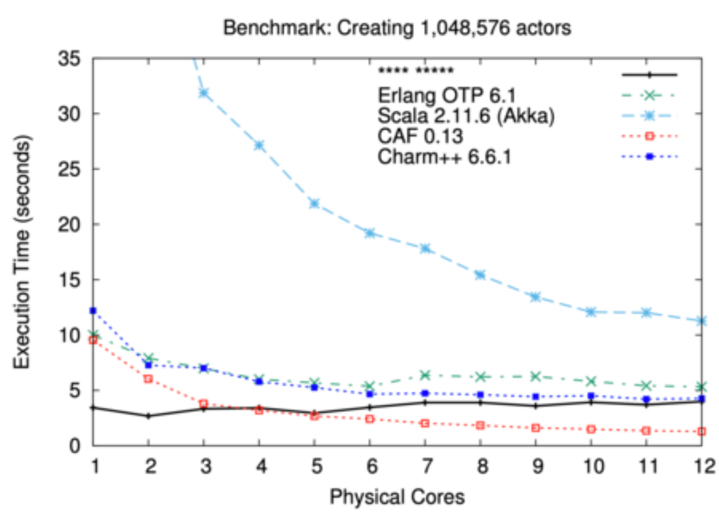

Figure 2: Mailbox performance, where **** is Pony(from[2]).

actor creation performance, and figure 2 shows mailbox performance.

3 Reference capability type system

Combining the actor-model with shared memory for performance is efficient but can introduce data-races. Well known approaches to static data-race freedom are based on uniqueness and immutability, but lack flexibility and high performance implementations. Pony's approach, based on deny properties allow reading, writing and traversing unique references, introduced a new form of write uniqueness, and guaranteed atomic behaviors.

3.1 Deny properties

Rather than indicate which operations are allowed on a reference, Pony's capabilities indicate what operations are denied on other references to the same object. It is distinguished what is denied to the actor that holds a reference(local aliases) from what is denied to all other actors(global aliases). Each capability stands for a pair of local and global deny properties. These are shown in table 2, and we call it "Reference capability matrix". For example, `ref` denies global aliases that can read from or write to the object, but it allows local aliases to both read from and write to it.

No capability can deny local aliases that it allows globally. Therefore, some cells in the matrix are empty. These deny properties are used to derive the operations permitted on a reference, and determine the ability of each capability references.

`iso` is for references to isolated data structures. If you have an `iso` varibale then you know that there are no other variables that can access that data. So you can change it however you like and give it to another actor.

`iso` is read and write unique, there can only be one reference at a time that can only be one reference at a time that can be used for reading or writing.

Because `iso` has read and write uniqueness, it can send the value to other actors(This is called *Sendable*).

`val` is for references to immutable data structures. If you have a val variable then you know that no-one can change the data. So it is *Sendable*, and you can read it and share it with other actors.

`ref` is for references to mutable data structures that are not isolated, in other words, "normal" data. If you have a `ref` variable then you can read and write the data however you like and you can have multiple variables that can access the same data. But you can't share it with other actors.

`box` is for references to data that is read-only to you. That data might be immutable and shared with other actors or there may be other variables using it in your actor that can change the data. Either way, the box variable can be used to safely read the data. This may sound a little pointless, but it allows you to write code that can work for both val and ref variables, as long as it doesn't write to the object.

`trn` has write uniqueness, and is used for data structures that you want to write to, while also holding read-only (box) variables for them. You can also convert the `trn` variable to a `val` variable later if you wish, which stops anyone from changing the data and allows it be shared with other actors.

`tag` is for references used only for identification. You cannot read or write data using a tag variable. But you can store and compare tags to check object identity and share(Sendable) `tag` variables with other actors.

3.2 Type system

By including reference capabilities to its type system, Pony finds all the errors at compile time, and guarantees no runtime error.

Subtyping relationship of reference capabilities are shown in fig 3. At here, κ is reference capability, $\circ$ is temporary type, <:is showing that left-side type is subtype of right-side type.

When reading a field `f` ftom an object ι we obtain a temporary. The capability of this temporary must be a combination of κ, the capability of the path leading to ι, and κ', then capability with which ι sees the field. Pony express this through the operator $\triangleright$, defined in table 2.

Storing a reference into a field of an object ι is legal if the type of the reference is both a subtype of the type of the field and also safe to write into the origin. The relation $\kappa \triangleleft \kappa'$, as defined in table 3.

Through applying subtyping rule and viewpoint adaption to reference capability, type proof theory guarantees that well-formed heap ensures data race freedom, and it is preserved.

Theorem 1 *A well-formed heap ensures data race freedom.*

$\forall \Delta, \chi, \alpha_1, \alpha_2, \mathtt{f}, \mathtt{g}$, *if*

1. $\Delta \vdash \chi\Diamond$, *and*

2. $\chi(\alpha_1) = (\_, \_, \sigma_1, \_, E_1[z_1.\mathtt{f} = z_3])$, *and*

3. $\chi(\alpha_2) = (\_, \_, \sigma_2, \_, E_2[z_2.\mathtt{g}])$

then $\chi(\alpha_1, |\sigma_1| \cdot z_1) \neq \chi(\alpha_2, |\sigma_2| \cdot z_2)$

Theorem 2 *Well-formedness is preserved.*

$\forall \Delta, \chi, if \Delta \vdash \chi\Diamond and \chi \to \chi' then \exists \Delta'.\Delta' \vdash \chi'\Diamond$

Proof of these theorems are written in paper [2].

$$\frac{T <: T'' \qquad T'' <: T'}{T <: T'}$$

$$\frac{}{S\ \kappa\circ <: S\ \kappa}$$

$$\frac{\kappa <: \kappa'}{S\ \kappa <: S\ \kappa'}$$

$$\texttt{iso} <: \texttt{trn} <: \{\texttt{ref}, \texttt{val}\} <: \texttt{box} <: \texttt{tag}$$

$$\mathit{Schedule}(T)\ \mathit{iff}\ T = S\ \kappa \wedge \kappa \in \{\ \texttt{iso}, \texttt{val}, \texttt{tag}\ \}$$

Figure 3: Sub-types and sendable types (from[2]).

	Deny global read/write aliases	Deny global write aliases	Allow all global aliases
Deny local read/write aliases	*Isolated(iso)*		
Deny local write aliases	Transition(`trn`)	*Value(val)*	
Allow all local aliases	Reference(`ref`)	Box(`box`)	*Tag(tag)*
	(Mutable)	(Immutable)	(Opaque)

Table 2: Capability matrix from[2]. Capabilities in *italics* are sendable.

$\kappa \triangleright \kappa'$			κ'			
κ	iso	trn	ref	val	box	tag
iso	iso	tag	tag	val	tag	tag
trn	iso	trn	box	val	box	tag
ref	iso	trn	ref	val	box	tag
val	val	val	val	val	val	tag
box	tag	box	box	val	box	tag
tag	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	

Table 3: Viewpoint adaption from[2].

$\kappa \triangleleft \kappa'$			κ'			
κ	iso	trn	ref	val	box	tag
iso	✓			✓		✓
trn	✓	✓		✓		✓
ref	✓	✓	✓	✓	✓	✓
val						
box						
tag						

Table 4: Safe to write table from[2].

References

[1] Pony language official page
`https://www.ponylang.io`

[2] Deny Capabilities for Safe, Fast Actors
`https://www.ponylang.io/media/papers/fast-cheap-with-proof.pdf`

[3] 200 minor programming language
`https://qiita.com/make_now_just/items/b2ab19f954417c71848d`

[4] Akka Streams
`https://doc.akka.io/docs/akka/current/stream/index.html`

[5] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wahlisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In Proceedings of the 2013 work shop on Programming based on actors, agents, and decentralized control, pages 8796. ACM, 2013.