Mid-term paper for Programming Theory class

Ogiwara

November 18, 2019

1 Objective

Pick one programming language, and explain about that. At here, I'll explain about Pony language [1] through description of features and performance, sample codes, and reference capability type system, which is the key concept of this language.

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`.

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

Data-race Free. Pony doesnt 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.

Deadlock-Free. This one is easy because Pony has no locks at all. So they definitely dont deadlock, because they dont exist.

2.2 Example Codes

```
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, and 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 ' `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[3] 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 type system than Rust [4] 's ownership type system.

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

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[5].

```
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 [6].

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

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

Type parameter and apply method in Scala

```
interface 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 style `apply` method as same as Scala[7].

```
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[8].

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 behaviours.

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] Elixir language official page
`https://elixir-lang.org`

[4] Rust language official page
`https://www.rust-lang.org`

[5] Python language official page
`https://www.python.org`

[6] Ruby language official page
`https://www.ruby-lang.org/en/`

[7] Scala language official page
`https://www.scala-lang.org`

[8] Go language official page
`https://golang.org`