Polytechnic University of Catalonia

Informatics Faculty of Barcelona

Programming Languages

Ponylang

Javier López-Contreras

May 6th

Contents

1 Introduction

This paper is an exploration of the programming language Pony. It will discuss its philosophy, its technical characteristics, its history and applications. This work was part of the *Programming Languages* course at the Informatics Faculty of the Polytechnic University of Catalonia.

Pony is a programming language developed by Sylvan Clebsch and a community of open source hackers. Its principal aim is to facilitate the development of reliable and fast highly concurrent programs. We will discuss how Pony handles concurrency in the Actor Model section.

2 Language Philosophy

Language development is a very opinionated field amongst programmers. As such, Pony's development team have a strong vision of how the language should be designed. They summarize their working philosophy as *get-stuff-done*, in contrast to the *the-right-thing* and *worse-is-better* approaches described in Richard Gabriel Essay [4]. Their aim is to follow a mixed strategy, not over-concentrating in neither the language theoretical perfection nor its application-oriented side.

2.1 Priorities

The *get-stuff-done* philosophy means it is sometimes adequate to sacrifice certain metrics in favour of others. The language team define Pony's priorities as the following ordered list [7], any point on the list might be sacrificed in favour of the higher ones.

- Correctness. A program should behave as the programmer expected.
- Performance. A program should be fast at execution time. This is the reason Pony takes a lot of computation out of execution time at compilation, like type checks, exception checks, etc. We will discuss these at depth in this paper.

- Simplicity. Human programming speed is an important metric and easing the human programming abilities is an aim for Pony. For example, finding bugs at compile time is always better that running into them at runtime, so Pony makes an effort to give human readable intelligent compilation errors for, for example, wrong type assignment bugs.
- Completeness. As expressed by the *get-stuff-done* slogan, completeness is at the end of the priority list.

2.2 Capabilities-secure

In this section we will describe a series of design decisions that are implemented in Pony that the development team call *capabilities-security.* The common factor in these is to move computation and checks to the compiler to improve performance and the programmers feedback-loop. To do this, the compiler needs to have the maximum information possible before the execution. Hence, Pony falls in the category of **Ahead of Time** compiled languages.

First, Pony is **statically strongly typed**, meaning that all data objects have a type assigned at compilation time and that cannot be changed at runtime. Even more strongly, Pony is fully **type safe** [2], which means that the only operations that can be performed on data are the ones sanctioned by its type. Thus, Pony is said to be **memory-safe**, meaning that it cannot modify memory indirectly thought loose pointers or buffer overruns. Also, memory usage is managed by a **Garbage Collector**, a system that will automatically deallocate memory that is no longer referenced in the program. Garbage collection together with memory-safety implies that memory leaks are not possible at runtime.

Second, Pony is **null-safe**, meaning that if an object of a type will always be instantiated. For example, class fields must be assigned by all constructors. Thanks to this, all the calling-method-on-null bugs are caught at compilation. In Pony there is no concept of *null* but there is a type *None*, that can be unionized with another type to knowingly build nullable types.

Thirdly, Pony is **exception-safe**, meaning that it checks statically that all exceptions are handled. Hence, there are no unexpected runtime exceptions.

Lastly, Pony is **data race free** and **deadlock free**, two properties that are inherited from the Actor Model, which we explain in detail in the following section.

2.3 The Actor Model

Pony follows the Actor Model paradigm, a variant of Object Oriented Programming that we describe below. It gives support for pure OOP and even has some features inspired by Functional Programming.

In the process-oriented and OOP paradigms, sharing mutable state between threads is a hard issue. It can be solved with memory locks, but these generate serious efficiency hits and correctness problems, like deadlocks. Locks are also particularly hard to program and are a typical source of memory bugs.

The *raison d'être* of Pony is to precisely to improve how concurrency is dealt with. Pony is based in the Actor Model, a theoretical programming paradigm that gives importance to the management of concurrent and parallel systems.

In this paradigm, the protagonist object is the Actor. An Actor is seen as the basic unit of sequential code. All the code executed by an actor is executed sequentially, in a single thread, so the programmer doesn't need to worry about the problems that arise with concurrency. On the contrary, all independent Actors are executed concurrently and can communicate with each other via message passing.

In most implementations of the Actor Model, the message-passing is implemented via *Isolated Data*, namely data that only (and exactly) has one pointer pointing to it. It is mutable state but the one pointer at a time restriction makes it safe to share between threads. Nonetheless, coding this in a process-oriented language is hard to get right, as you need to enforce the promise of never using a pointer once its shared. Pony's compiler solves exactly that problem, it makes sure that shared data is isolated. It does so making use of an innovative type annotation called *reference capabilities*, which we describe in the Type System section.

3 Type System

Pony has a very strong type system. It demands a lot from programmers but, in exchange, it can give guarantees on the execution of its programs like no memory bugs, no nullability bugs, no wrong type assignment bugs and no unhandled expressions. Even though it has high demands for correctness, the Pony's compiler includes a strong Type Inference engine that simplifies the programmers job.

Pony's type system also includes the innovative type annotation used to enforce the Actor Model constraints.

3.1 Actors

Actors are the main object type in Pony. They are classes with extra logic to implement the Actor model paradigm. Behaviours are a special type of method that is run asynchronously and that actors can call on themselves or on other actors. Behaviours always return *None* as they have not run yet when they are called.

Actors run single-threadedly but Behaviours are asynchronous calls. Hence, behaviours inside of an Actor will be executed, in no particular order, after the main execution is finished. This can be visualised in the order of the two prints in the following example.

```
actor Main
  new create(env: Env) =>
    call_me_later(env)
    env.out.print("This is printed first")

  be call_me_later(env: Env) =>
    env.out.print("This is printed last")
```

An important characteristic is that Actors, unlike threads, are extremely cheap. The overhead for an empty actor is in the order of bytes. Hence, it is expected that a Pony program can run up to 10^5 actors concurrently. Pony's internal scheduler is optimized for this kind of load balancing.

3.1.1 Reference Capabilities

When an actor calls a behaviour of a different actor, the types of the arguments of the behaviour determine if the message is send as Immutable State or as Mutable Isolated State.

The type annotation that decides the sharing type is called the Reference Capability and every type in Pony has one. It can be any of `val`, for immutable state, `iso`, for mutable isolated state, `box` for read-only data, `ref` for normal non-isolated data that can not be shared. There are a few other Reference Capabilities for technical reasons, but they are outside of the scope of this summary.

When a behaviour is called, the Pony compiler checks that the attributes are either `val` or `iso`. In the first case it shares the content of the variable as inmutable state. In the second, it shares the variable as isolated data and also checks that the variable is never used again in the initial Actor. If it is, it generates a compilation error. This is how Pony automatically enforces the one-pointer-at-a-time invariant in the Action model paradigm.

The following example exemplifies how to share isolated variables between Actors. The keywords `recover-end` and `consume` indicate when the Actor takes (enforces a reference capability on an expression) and leaves control of the data pointer.

```
actor Spam
  let _env: Env
  new create(env: Env) =>
    _env = env
  be accept_array(arr: Array[USize] iso) =>
    _env.out.print(arr.size().string())

actor Main
  new create(env: Env) =>
    let spam: Spam = Spam(env)
    var array: Array[USize] iso = recover
        iso Array[USize]
```

```
end
array.push(USize(42))
spam.accept_array(consume array)
// The following line would be a compilation error, array not isolated
// env.out.print(array(0).string())
```

3.2 Classes

Pony gives support to the object oriented paradigm through classes. They are dealt by the compiler as Actors without the concurrency managing logic. Unlike classical OOP, classes in Pony are organized by composition, not inheritance.

3.2.1 Class composition

Unlike Java, Pony does not implement object inheritance and instead implements object composition.

Classical object inheritance modularizes and reuses code based on polymorphisms between the child and parent classes. The programmer can state that, for example, a `Cat is Animal`, and then the `Animal`'s fields and methods can be overwritten by the `Cat`'s and the compiler always chooses to execute the most specific implementation.

In recent years, the general opinion has realized that the classical inheritance system is too strict and brings a series of problems when pushed to a large scale. One of this problems is that it is too hierarchical, a if `Cat` is an `Animal`, then it cannot be a `MemberOfTheHouse`, as the relationship between the two possible parents is not hierarchical.

To solve this, a new method appeared called object composition. In this paradigm, the programmer can state that a `Cat has AnimalBehaviour` and also `Cat has MemberOfTheHousePrivileges` and the `Cat` class will inherit fields and methods from both parent classes. Composition has benefits and drawbacks but it is generally regarded as a better language design than inheritance, for example in the classic book *Design Patterns* [3].

Here is an example of composition in action. It makes use of the type intersection construct that Pony supports. The class `Bob` will have access to both `name()` and `hair()`.

```
trait Named
  fun name(): String => "Bob"

trait Bald
  fun hair(): Bool => false

class Bob is (Named & Bald)
```

3.2.2 Traits and Interfaces

Pony implements two types of sub-typing, nominal sub-typing thorugh *Traits* and structural sub-typying throught *Interfaces*.

Nominal sub-typing is the usual Java-like implementation of categorical types through interfaces (which in pony are called traits, Pony's interfaces are a slightly different object).

Structural sub-typing on the other hand is very interesting. A Pony programmer can define an *Interface* and all the objects that follow that interface's contract will get that interface assigned as a parent type at compilation. The interesting part is that the programmer does not need to indicate which objects follow the *Interface*, this matching is done completely by the compiler. Structural Type Systems are not an innovation created by Pony but are an extremely useful feature that I had never heard of before. To maintain the safety of modules, interfaces can not ask for private fields of methods in their contract.

Here is an example of structural sub-typing in action. We create a `Compactable` interface and the compiler will automatically assign it to any type that has a `size()` and `compact()` methods, for example to `Array, Sets, Maps`. Thanks to this, we can then call `Compactor.try_compacting(...)` with any of such objects.

```
interface Compactable
  fun ref compact()
  fun size(): USize

class Compactor
  """
  Compacts data structures when their size crosses a threshold
  """
  let _threshold: USize

  new create(threshold: USize) =>
    _threshold = threshold

  fun ref try_compacting(thing: Compactable) =>
    if thing.size() > _threshold then
      thing.compact()
    end
```

4 Execution System

Pony is compiled *Ahead-of-Time* directly into an executable. There is no Virtual Machine nor a Bytecode. Eventhough this means that compilation can be less performant that in a *Just-in-Time* compiler, but Pony needs AOT to statically enforce its capabilities security.

Pony gives support for a native Foreign Function Interface to *C*. Nonetheless, Pony can not enforce capability safety in a program that calls a *C* library using the FFI. All the Pony guarantees do not stand when the FFI API is used.

5 Miscellaneous

During this bibliographical research, I have found a small amount of curious properties that I think are worth sharing.

- In Pony, the assignment expression `a = b` returns the old value of *a*. This is called a destructive read and it has some cool properties. For example, a swap operator can be implemented as `a=b=a`, instead of the usual

  ```
  var atemp = a
  a = b
  b = atemp
  ```

  This small language feature is used extensively in Actor-based message handling.

- Pony does not accept any shadowing of variables between scopes. Shadowing bugs are caught at compile time. It also does not support Global Variables.

- Pony forces the use parenthesis in ambiguous mathematical expressions like `1+2*3`. The development team think that people often forget the priorities between algebraic operations (specially with weirder operators, like logical shifts). Hence Pony makes no assumption about the mathematical priorities between operands and flags any ambiguous expression as an error at compile time. In my experience, operation priority mishaps are a fairly usual bug and I think this is a good general solution to the problem.

- Large arithmetic computations is a usecase where correctness and efficiency collide. A language that always checks for correctness at runtime (overflows, divisions by 0) will loose efficiency and, in the contrary case, correctness might be compromised. To solve this, Pony gives 2 sets of arithmetical operators, safe and slow (`+, -, *`) and unsafe and fast (∼ `+,` ∼ `-,` ∼ `*`).

- Variable names can end in a `'` to indicate very similar variables. For example, this is used in constructors. I had never seen this notation in a programming language before.

6 History

Pony was born in 2014 out of the necessity of a compiler-enforced Actor paradigm language for large concurrent systems [1].

In the early 2000, Pony designer Sylvan Clebsch, worked at a flight simulator company, where he implemented an actor-based concurrency system for the simulator's physics engine. Some years later, working in finance, he had to implement a similar system for a different purpose and he was surprised at how the same errors seemed to pop-up.

He decided to implemented a C/C++ library to handle actor-based concurrency. The library was very well received, showing a need for these systems in the industry. Nonetheless, programmers usually struggled enforcing the one-pointer-at-a-time invariant in which the Actor Model's message communication is based.

The ideas of using the Type System to enforce Actor model's promises was developed during Sylvan's PhD at the Imperial College of London, under mentorship of Sophia Drossopoulou. Little after, they started a company named Causality to develop their idea. The company ended up going bankrupt but the open source community still maintains and develops the language to this day.

7 Applications

Pony's theoretical application is to build large scale, highly concurrent and performing systems. In reality, Pony is still a somewhat experimental language. It has never reached a stable version, which makes it hard for a mainstream project to be developed in it. At the moment, the project is purely voluntary work and does not have any corporation backing, so development is very slow.

Nonetheless, there are a few large projects developed in Pony, made by members of the language development community.

- Jylis [5]. A distributed in-memory database for Conflict-free Replicated Data Types

- Wallaroo [8]. Distributed Stream Processor written in Pony

8 My experience learning Pony

Pony is well documented [7], the tutorials are very well written and behaviour is well defined.

Nonetheless, hearing about its history and reading the community forums, its hard not to realize that Pony is slowly loosing users and lacks the support to be a general-use production-safe programming language.

I feel like learning Pony has been very interesting to hear about what a series of very good language engineers opine about programming language design. I have learned the benefits and drawbacks of some modern design patters like Class Composition, Structural Subtyping and Action model. I am sure this will be useful the next time I have to implement a parallel programming pipeline. Nonetheless, in my opinion, the language itself is not ready for general use and might only be used effectively by open source language developers that can understand and fix the language problems on the go.

9 Bibliographical Analysis

This section explains the references compiled for the production of this paper.

- Pony's tutorial and documentation [7]. The tutorial series and the language design documentation are extremely well written and maintained. Its the source of truth for the implementation of Pony and it has been written accessibly for new users.
- Wikipedia entries of common Programming Language designs [9]. In Computer Science and in Programming Language Design, the Wikipedia entries are well sourced and maintained. I have used it to learn about type-safety, memory-safety, null-safety.

- *Design Patterns* book [3]. p. 30, 44. I have used this book to learn about classical Design Pattern, specifically about the Inheritance vs. Composition debate.
- Engineer's Blogs [1], [4]. This serve to learn about the human component of the software product, in this case, Ponylang. They are primary sources and can be biased. We need to understand the information in this sources as the voice of a opinionated engineer defending their own project and be careful with the biases it might impose on our view.
- Pony's Community Forum at Zulip [6]. Some of the coding examples in this document have been inspired by examples explained in forum posts. In general is a good technical source, maintained by a group of open source language developers. In contrast with other programming forums, Ponylang's seems very beginner friendly.

References

[1] Sylvan Clebsch. *An early history of Pony*. Last accessed 7 May 2022. 2017. URL: `https://www.ponylang.io/blog/2017/05/an-early-history-of-pony/`.

[2] Sylvan Clebsch and Sophia Drossopoulou. *Fully Concurrent Garbage Collection of Actors on Many-Core Machines*. Last accessed 7 May 2022. 2013. URL: `https://www.ponylang.io/media/papers/opsla237-clebsch.pdf`.

[3] Ralph Johnson Erich Gamma Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[4] Richard Gabriel. *The Rise of Worse is Better*. Last accessed 7 May 2022. 1992. URL: `https://www.jwz.org/doc/worse-is-better.html`.

[5] *Jylis Landing Page*. Last accessed 7 May 2022. URL: `https://jemc.github.io/jylis/`.

[6] *Ponylang Community Forum*. Last accessed 7 May 2022. URL: `https://ponylang.zulipchat.com/#`.

[7] Pony Language Team. *The Pony Philosophy*. Last accessed 7 May 2022. URL: `https://tutorial.ponylang.io/index.html`.

[8] *Wallaroo Github Repository*. Last accessed 7 May 2022. URL: `https://github.com/WallarooLabs/wally`.

[9] *Wikipedia Type System*. Last accessed 7 May 2022. URL: `https://en.wikipedia.org/wiki/Type_system`.