HW6: Language Bindings for TensorFlow

Quentin Truong - University of California, Los Angeles

Abstract

Python is a highly effective prototyping language due to its ease of use, readability, wide library support, and flexibility; however, Python is rarely as performant as other languages. With respect to a server proxy herd for machine learning applications, we compare Python, Java, OCaml, and Crystal on the basis of ease of use, flexibility, generality, performance, reliability, support for event-driven servers, and support for TensorFlow bindings.

1. Introduction

1.1. Problem Statement

A server proxy herd for machine learning applications has been built using Python, Python's asyncio, and Python bindings for TensorFlow. After benchmarking, it is discovered that the Python sections of the program are the slowest. To increase performance, the program will be rewritten using either Java, OCaml, or Crystal.

Tradeoffs between ease of use, flexibility, generality, performance, reliability, support for event-driven servers, and support for TensorFlow bindings will determine which language is chosen.

2. Comparing Python, Java, OCaml, and Crystal

2.1. Table

Refer to Table 1 in appendix for comparisons between Python, Java, OCaml, and Crystal.

2.2. Ease of Use

Programming language should be easy-to-use. Oftentimes, this is a question of syntax. Since most programmers intend to solve problems, boilerplate code is simply overhead for the programmer. Another important consideration for easy-to-use languages is memory management. Memory management should be handled automatically, removing the need for explicit allocation and freeing of memory. Lastly, the language's type checking should help programmers write bug-free code while not hindering expressiveness.

We begin with considering the syntax of Python, Java, OCaml, and Crystal. Interestingly, Python determines code blocks according to whitespace. This removes the need to add beginning and ending braces, resulting in more concise and clear code. Moreover, enforcing whitespace enables new developers to quickly understand the structure and flow of the program. Other languages, such as Java, rely on braces and allowing the developer more freedom in how to space their code [1]. OCaml, in fact, has no rules regarding whitespace. While this flexibility enables the programmer complete freedom over spacing, it removes the sort of uniform structure across programs that Python has [2]. Crystal, deriving from Ruby, puts heavy emphasis on readability. While Crystal still occasionally uses braces, the result remains clear and concise.

Memory management is important, as few programmers set out wanting to manipulate memory – most programmers want to manipulate data structures. Moreover, most modern garbage collectors are often sufficiently intelligent that the performance side effects are negligible for most non-critical applications. Automatic garbage collectors frequently result in many fewer memory-related issues, as compared to explicit memory allocation and freeing. This is because memory allocation and freeing is rarely an easy task when it comes to complex objects. Python, Java, OCaml, and Crystal all implement an automatic garbage collector using various algorithms [3], [4], [5], [6].

Type checking ensures that functions only operate on types as defined. For instance, without type checking, functions may have unexpected behavior, if they expect an integer and receive a Boolean. Python uses dynamic and strong type checking, meaning that types are resolved only at runtime, but are strictly enforced [7]. Strong type checking prevents many common programming mistakes, such as passing an integer when the function expects a string. However, since Python uses dynamic type checking, the programmer will only realize type errors at runtime rather than compile time. Other languages, such as Java, OCaml, and Crystal employ static and strong type checking, meaning that type errors are caught at compile time and types are enforced strictly [8], [9], [10]. Notably, Python's type checking enables duck typing, allowing programmers to reuse objects without the extensive boilerplate interface code that many languages, like Java, require. Duck typing is a powerful language feature which enables object reuse in a vast variety of contexts. Another powerful tool is type inference, which allows programmers to leave off type-annotations whenever the compile is able to determine the type according to the variable's value or relationship to other variables. Both OCaml and Crystal support type inference, resulting in more concise code and reducing development time for programmers [9], [10].

2.3. Flexibility

Programming languages should allow the programmer to express ideas in a concise and natural manner. Python, for instance, is a multi-paradigm language which allows the programmer to choose between imperative, object-oriented, and functional programming tools. This flexibility of tools enables programmers to solve a wide range of questions in styles well suited to the nature of the question. Like Python, Java supports imperative, object-oriented, and functional programming tools. However, because Python and Java were not originally designed for functional programming, their integration for functional programming is limited to functools and functional interfaces, respectively [11], [12]. OCaml, on the other hand, was designed for functional and object-oriented programming and supports both through the language itself and most built-in API's.

2.4. Generality

Generality of a language is important, as recalling quirky, special cases does little more than deter programmers from producing working code. Python, for instance, is quite regular in its API function names. For example, to cast any type to integer, float, boolean or char, we may simply call int(), float(), bool(), or chr() [13]. This contrasts with OCaml, where casting a float to integer, int_of_float(), and string to integer, int_of_string(), are different methods; albeit, OCaml's naming convention is fairly straightforward [14]. Java is not so straightforward. In Java, casting a float to integer uses int(), whereas casting a string to integer uses Integer.parseInt() or Integer.valueOf(), depending if you want a primitive integer or object Integer. This complexity burdens the programmer with needing to recall unnatural function calls. Python's predictability across types and functions allows the programmer to focus on writing code, rather than recalling function names.

Although some of Java's functions names are irregular, many of Java's built-in packages have very regular naming conventions. For instance, consider the java.util.concurrent.atomic package and its associated classes; the library implements a number of classes with names such as AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, and so on [15]. Java's built-in packages are known to follow naming conventions based upon compounding words together. Although this leads to longer names, the names are clear and regular.

Crystal derives from Ruby, a language whose philosophy is to focus on the human, rather than the machine [16]. Interestingly, Ruby is often said to follow the Principle of Least Astonishment, where the language should behave in the manner which minimizes confusion for the programmer [16]. Oftentimes, this means behaving in quite regular, general ways. As such, Crystal, which shares much of its syntax with Ruby, is general and regular.

2.5. Performance

Python is a high-level, interpreted programming language, and CPython is the most common implementation of Python. CPython uses an intermediate bytecode for the interpreter, leading to lower performance since the source code is not compiled to machine code [17]. Notably, performance has never been a goal for Python or CPython, affecting design decisions across Python /CPython including the use of a global interpreter lock, automatic garbage collector through reference counting, and allowing monkey patching [18].

Java, although it also uses bytecode, uses a just-in-time compiler. This allows frequently-run code to be compiled to native machine code, greatly increasing performance [19]. OCaml, is known to be a very fast language [20]. Crystal was designed to be "fast as C", and is indeed quite performant according to a variety of benchmarks [21].

2.6. Reliability

Python, Java, OCaml, and Crystal all thoroughly support exception handling through various means. Notably, Python, Java, and OCaml are each backed by more than twenty years of development, whereas Crystal was only officially released in 2014. However, despite the language maturity difference, Crystal is thoroughly documented and quickly growing into a production-ready programming language [22].

Reliability and its relationship to memory management and type errors has been addressed in section 2.2.

2.7. Event-driven Server

Python supports event-driven servers through its asyncio package. Connections are easily set up through high-level functions provided by the asynchronous package [23]. Java supports non-blocking I/O through its java.nio [24]. Also, Deft, an event-driven server for Java has been created and accepted as an Apache project [25]. OCaml has built-in support for the Async library, which, although is not a true event loop server, may accomplish similar tasks [26]. Lastly, Crystal fully supports event loops using fibers, which are Crystal's version of very lightweight threads [27]. Crystal's event loop is capable of managing asynchronous tasks and communications. Crystal's runtime scheduler manages when fibers run; fibers may also explicitly yield [27]. Crystal also supports servers built using fibers.

2.8. TensorFlow Bindings

Python bindings for TensorFlow are officially supported [28]. This allows our application to fully integrate with TensorFlow. Java has incomplete, but still usable bindings [29]. OCaml has fewer bindings available than Java, however is still sufficient for basic tasks [30]. Crystal, as a very new language, has very few bindings for TensorFlow. Crystal's bindings for TensorFlow are early-stage [31].

References

[1] Java Code Conventions. 12 September 1997. <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>.

[2] OCaml for the Skeptical. 17 June 2006. <https://www2.lib.uchicago.edu/keith/ocaml-class/prelim.html>.

[3] Reference Counts. 22 June 2001. <https://docs.python.org/2.0/ext/refcounts.html >.

[4] Memory Management. 31 Decemeber 2017. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management>.

[5] Memory Management by Object CAML. <https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora087.html>.

[6] Wajnerman, Juan. Garbage Collector. 5 December 2013. <https://crystal-lang.org/2013/12/05/garbage-collector.html>.

[7] Why is Python a dynamic language and also a strongly typed language. 24 February 2012. <https://wiki.python.org/moin/Why_is_Python_a_dynamic_language_and_also_a_strongly_typed_language>.

[8] Dynamic typing vs. static typing. March 2015. <https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html>.

[9] Strong Static Type Checking with Inference. 17 June 2006. <https://www2.lib.uchicago.edu/keith/ocaml-class/static.html>.

[10] Type Inference. <https://crystal-lang.org/docs/syntax_and_semantics/type_inference.html>.

[11] functools. <https://docs.python.org/3/library/functools.html>.

[12] Subramaniam, Venkat. Functional Interfaces. 8 September 2017. <https://www.ibm.com/developerworks/library/j-java8idioms7/index.html>.

[13] Built-in Functions. <https://docs.python.org/3/library/functions.html>.

[14] Pervasives. <https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html>.

[15] Package java.util.concurrent.atomic. <https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/atomic/package-summary.html>.

[16] Ruby 1.9. 22 February 2008. <https://www.youtube.com/watch?v=oEkJvvGEtB4>.

[17] Disassembler for Python bytecode. 6 June 2018. <https://docs.python.org/3/library/dis.html>.

[18] Notes on Python and CPython performance, 2017. 2017. <http://faster-cpython.readthedocs.io/notes_2017.html>.

[19] JIT Compiler Overview. <https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.zos.70.doc/diag/understanding/jit_overview.html>.

[20] Performance and Profiling. <https://ocaml.org/learn/tutorials/performance_and_profiling.html - Speed>.

[21] Dogruyol, Serdar. An Introduction to Crystal: Fast as C, Slick as Ruby. 5 May 2017. <https://blog.codeship.com/an-introduction-to-crystal-fast-as-c-slick-as-ruby/>.

[22] Crystal Programming Language. <https://crystal-lang.org/docs/>.

[23] Asynchronous I/O, event loop, coroutines, and tasks. <https://docs.python.org/3/library/asyncio-protocol.html>.

[24] Package java.nio. <https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html>.

[25] Synodinos, Dio. Asynchronous, Event-Driven Web Servers for the JVM: Deft and Loft. 24 Januay 2011. <https://www.infoq.com/articles/deft-loft>.

[26] Hickey, Jason, Madhavapeddy, Anil, Minsky, Yaron. 2018. Chapter 18. Concurrent Programming with Async. <https://realworldocaml.org/v1/en/html/concurrent-programming-with-async.html>.

[27] Concurrency. <https://crystal-lang.org/docs/guides/concurrency.html>.

[28] TensorFlow in other languages. <https://www.tensorflow.org/extend/language_bindings>.

[29] TensorFlow for Java. <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/java>.

[30] OCaml bindings for TensorFlow. <https://github.com/LaurentMazare/tensorflow-ocaml/>.

[31] Crystal bindings for TensorFlow. <https://github.com/fazibear/tensorflow.cr>.

Appendix

	Python	Java	OCaml	Crystal
Memory Management	Automatic Garbage Collector using Reference Counting [3]	Automatic Garbage Collector using Mark-and-Sweep with Generations [4]	Automatic Garbage Collector using variant of Mark-and-Sweep with Generations [5]	Automatic Garbage Collector (Boehm-Demers-Weiser Conservative) [6]
Type Checking	Dynamic, Strong [7]	Static, Strong [8]	Static, Strong, Inferred [9]	Static, Strong, Inferred [10]
Exception Handling	try, except, else, raise	try, catch, finally, throw	Error, exception, raise	Exception, begin, rescue, else, ensure, end, raise
Event-driven Server	asyncio, aiohttp [23]	java.nio [24], Deft [25]	cohttp.async, async.std [26]	Fibers [27]
Supporting Tools	json, urlparse	org.json, Uri.Builder	yojson, uri	json, uri
TensorFlow Bindings	Officially supported [28]	Incomplete, but usable [29]	Some bindings available [30]	Early-stage [31]

Table 1: Comparison between Python, Java, OCaml, and Crystal