Extending PyJL to Translate Python Libraries to Julia

Miguel Marcelino
INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
miguel.marcelino@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Many new high-level programming languages have emerged in recent years. Julia is one of these languages, offering the speed of C, the macro capabilities of Lisp, and the user-friendliness of Python. Regarding the latter, Julia's syntax is one of its major strengths, which makes it ideal for scientific and numerical computing. Furthermore, Julia's high performance on modern hardware makes it an appealing alternative to Python. However, its library set is still reduced when compared to Python.

To address this issue, we propose extending PyJL, a transpilation tool, to convert Python libraries into Julia and bridge the gap between the two languages. Although the development of PyJL is still at an early stage, our preliminary results reveal that the generated code is human-readable and capable of high performance.

KEYWORDS

Source-to-Source Compiler, Automatic Transpilation, Library Translation, Python, Julia

1 INTRODUCTION

Transpilers translate source code between an input and a target language. Nowadays, they are used to translate source code between high-level programming languages. As an example, consider TypeScript [2], whic is transpiled to JavaScript.

The use of transpilers as library conversion tools becomes more relevant with the rise of new programming languages, as they lack the large library sets found in more established languages and it is expensive to develop these libraries from scratch. However, automatically translating a source language to a target language requires a careful analysis of syntactic and semantic incompatibilities that can be difficult to overcome.

To address this issue, we present PyJL, a transpiler that translates Python source code to Julia that we have been developing to speedup the production of libraries to Julia. Preliminary results reveal that a significant subset of Python code can be converted into pragmatic Julia code with minimal programmer intervention. Furthermore, the produced code frequently achieves good performance with small adjustments. On the other hand, many Python functionalities are difficult to convert into semantically equivalent and pragmatic Julia code. The current limitations are described in Section 3.

2 PYJL

PyJL [3] is part of the Py2Many [4] transpiler, which is a rule-based transpilation tool. Py2Many offers a generic framework to transpile Python to many C-like programming languages, such as Rust, Go, and C++. PyJL builds upon that framework to translate Python source code to Julia. It is also a flexible translation tool, allowing

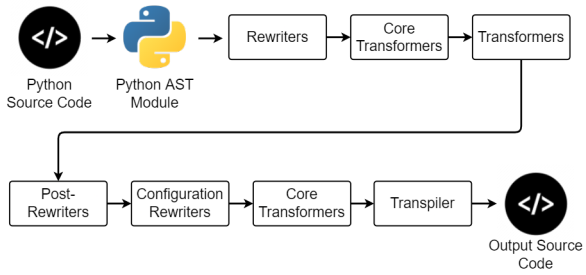

Figure 1: PyJL Architecture

the programmer to change the output of translation using external annotation files. In this section, we briefly describe the architecture of the transpiler.

As seen in Figure 1, the PyJL transpiler receives Python source code as input and first parses it with Python's `ast` module,[1] which generates an Abstract Syntax Tree (AST). It then uses several intermediate transformation phases to generate the equivalent Julia source code. A brief description of each phase follows:

(1) *Rewriters* can be both language-specific or -independent. A rewriter changes the structure of nodes in the AST to match equivalent nodes in the target language.
(2) *Core Transformers* are language-independent transformers that add relevant information for the translation process.
(3) *Transformers* are language-specific and add information to nodes. An example would be to add type annotations for type inference.
(4) *Post Rewriters* are Rewriters that have dependencies on some previous phase. Their functionality is identical to that of *Rewriters*.
(5) *Configuration Rewriters* supports configuration files in JSON and YAML format that specify AST modifications.
(6) *Transpiler* translates language syntax and semantics and converts the AST to a string representation in the target language.

Notice that the *Core Transformers* phase is executed at two stages. The first makes core information available to the *Transformers* and *Post Rewriters* stages. The second ensures that core Py2Many transformations are not overwritten, making them available in the *Transpiler* phase.

3 LIMITATIONS

One of the major difficulties in the conversion of Python source code to Julia is the lack of type information at transpilation time. As Python is a dynamically typed language, type information is

[1] Abstract Syntax Tree - Python 3.10: https://docs.python.org/3/library/ast.html (Retrieved on January 27th, 2022)

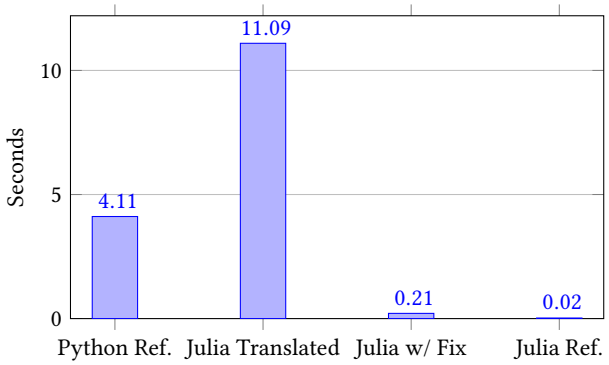

Figure 2: N-Body Implementations

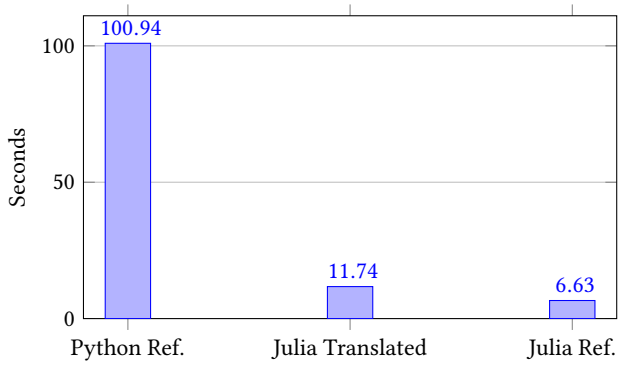

Figure 3: Binary Trees Implementations

only available at runtime. Currently, PyJL requires type hints in function arguments and return types, which is still susceptible to the problem of type soundness, as Python does not check for type hint correctness. To solve this issue, we are currently integrating pytype [1] in PyJL to perform type verification.

PyJL also maps a subset of Python's Object-Oriented features. To map Python's classes, PyJL currently supports two alternatives: (1) Mapping classes to structs and functions in Julia and creating a hierarchy with abstract types, (2) Using the `Classes` package, which offers a comparable syntax to Python. In both aproaches, class constructors and also the `__init__` method are mapped to Julia constructors. More improvements are required to support Python's special methods, such as `__repr__`, which could be mapped to a Base.show() in Julia. In addition, there are also specific methods, such as `__add__`, which are not supported.

Furthermore, the translation of Python's `multiprocessing` library is also a subject of future study. This would require the use of Julia's `Distributed` library and proper handling of namespaces when using multiple processors.

4 EVALUATION

To assess the capabilities of the PyJL transpiler, we use Py2Many's test suite together with a subset of Python's formal test suite. The outputs of the tests are compared with expected test results to detect any errors.

Regarding the performance of the translated code, we used the transpiler to convert commonly used benchmarks, of which we present two. The first is an implementation of the N-Body problem, which predicts the gravitational interactions of planets in the solar system. The second tests Garbage Collection performance by allocating short-lived trees and iterating through them recursively. The translation results are publicly available.[2]

The N-Body benchmarks in Figure 2, show the translated Julia source code being almost 3 times slower than the Python reference implementation, which is unexpected given Julia's high performance. Analyzing the generated code revealed that the slowdown was caused by insufficient type information. However, by annotating just one line of code, we managed a speedup of 52.6× when compared to the initial translation result, which makes the translated Julia code 19.5× faster than the reference Python implementation. Julia's reference implementation is an order of magnitude faster, but it is highly optimized and benefits from Julia's performance characteristics

Regarding the second benchmark, its results can be seen in Figure 3. In this case, translating Python's reference implementation yields good results without any user changes, resulting in an 8.6× faster execution time. Both the Python Reference version and the generated Julia code use a similar amount of memory, measured at 270MiB and 250MiB respectively. The reference Julia version runs almost twice as fast as the translated version, although it also uses about twice as much memory.

[2]PyJL Benchmarks Repository: https://github.com/MiguelMarcelino/pyjl_benchmarks

5 CONCLUSIONS

This work aims to automate the translation of Python libraries to Julia, with the aim of increasing Julia's available library set. The generated code should respect the pragmatics of Julia, allowing it to be further maintained by Julia programmers.

As demonstrated by our preliminary evaluation, the PyJL transpiler requires little programmer intervention to generate high performance Julia source code.

The development of the PyJL transpiler is still at an early stage, although preliminary results are favorable, both in terms of code intelligibility and the performance obtained with few, if any, changes to the generated source code.

REFERENCES

[1] Google. Pytype: A static type analyzer for python code, March 2015. [Online. Retrieved February 25th, 2022 from: https://github.com/google/pytype].
[2] P. Japikse, K. Grossnicklaus, and B. Dewey. *Introduction to TypeScript*. Apress, 2017. Chapter 7.
[3] M. Marcelino and A. Menezes Leitão. Pyjl implementation, 2021. Retrieved April 8th, 2022 from: https://github.com/MiguelMarcelino/py2many.
[4] A. Sharma, L. Martinelli, J. Konchunas, and J. Vandenberg. Py2many: Python to many clike languages transpiler, 2015. Retrieved November 18th, 2021 from: https://github.com/adsharma/py2many.