Transpiling Python to Julia using PyJL

Miguel Marcelino
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

Transpilers convert source code between programming languages. With the rise of new high-level programming languages, transpilers are ideal tools to speedup the conversion of libraries written in more established languages to newer and/or less popular ones, fostering their adoption.

Julia is a recently introduced programming language that targets various application areas of the widely popular Python language. Unfortunately, it still lacks many of the high-quality libraries found in Python. To speedup the development of libraries, we propose extending the PyJL transpiler to translate Python source code into human-readable, maintainable, and high-performance Julia source code.

Despite being at an early development stage, our preliminary results show that PyJL generates human-readable code that can achieve good performance with minor changes.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; • **General and reference → Cross-computing tools and techniques**.

KEYWORDS

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

ACM Reference Format:
Miguel Marcelino and António Menezes Leitão. 2022. Transpiling Python to Julia using PyJL. In *Proceedings of the 15th European Lisp Symposium (ELS'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.6332890

1 INTRODUCTION

A generic definition of a transpiler is a tool that transforms input source code into output source code, where the input and output source code can be written in the same or in different programming languages. The term *transformation* is relatively broad, and many solutions use more specific concepts to categorize different transpilation approaches. DMS [3], a tool that focuses on the automatic management of large software solutions, uses the concept of Design Maintenance. Other tools in the area of Safety-Critical Computing [28] use the concept of Source Code Manipulation to implement fault-tolerance mechanisms. In the context of this research, we focus on the topic of Source-to-Source Translation, where transpilers translate source code from an input language to a target language.

The first transpiler was developed in 1978 to provide compatibility between an 8-bit and a 16-bit processor. It was called CONV86 [7] and was developed by Intel to translate assembly source code from the 8080/8085 to the 8086 processor. At the time, many other transpilers were developed with a similar purpose, such as TRANS86 and XLT86 [30]. Nowadays, with programmers developing software in higher-level programming languages, it makes sense to have transpilers operate at this level.

In recent years, we have seen the rise of many new high-level programming languages, such as Rust, Go, TypeScript and more. Among them is the Julia programming language, which claims to have the performance of C, the ease of use of Python, and the macro capabilities of Lisp, among others. However, Julia currently lacks the large library sets found in more established programming languages. Converting libraries from these languages to Julia would allow programmers to benefit from extended library support and from Julia's performance on modern hardware.

Manually translating large code-bases is a difficult task and requires substantial resources. For instance, the Commonwealth Bank of Australia converted its code-base from COBOL to Java, spending $750 million over five years. In this regard, manually translating Python's large library set would be an enormous challenge. Using a transpiler to convert libraries automatically would benefit programmers. However, automatic translation is a challenge for a transpiler, which we will discuss in the following section.

2 AUTOMATIC TRANSLATION

Automating the translation between source and target languages is addressed with differentiating perspectives. LinJ [18] aims at a fully automatic translation of Common Lisp to Java source code. JSweet [24] translates Java to JavaScript and preserves JavaDoc documentation in JSDoc. Other tools, such as the Fortran-Python two-way transpiler [5], intentionally require manual intervention and request the programmer to annotate the input Python source code with type hints before translating it to Fortran.

We consider that automating the translation process is relevant, as the goal is to translate libraries. Furthermore, since the aim is to generate human-readable and maintainable code, the transpiler needs to translate language syntax and semantics as programmers would, by preserving the pragmatics of Julia.

To translate language syntax and semantics, we need to consider how different constructs map to the target language. As an example, consider the following code written in Python:

```
ls1 = [1,2]
ls2 = [3,4]
ls_sum = ls1 + ls2
```


ELS'22, March 21–22, 2022, Porto, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.6332890

Despite the fact that Julia has identical syntax for assignments, arrays, and arithmetic operators, executing this example in Julia would not yield the same results. In Julia, adding `ls1` and `ls2` results in an element-wise addition, producing `[4,6]`. In Python, this results in the creation of a new list that contains the elements of both lists, i.e., `[1,2,3,4]`. A correct translation to Julia would use the syntax:

```
ls_sum = [ls1;ls2]
```

Furthermore, if the source and target languages promote different programming paradigms, the transpilation process becomes even more difficult. For example, consider translating Object-Oriented (OO) Python code to Julia, where a transpiler would need to map functionalities such as multiple inheritance, which Julia currently does not support, and handle method overrides. Python classes also implement special methods, such as `__init__` and `__str__`. Additionally, Python also allows redefining built-in operators, such as arithmetic operators, for class instances. A transpiler that aims to preserve Julia's pragmatics would need to map all these functionalities while also generating a target program that is easy to understand and modify.

The mapping of library calls to the target language should also be considered. This can be done on a per-function basis, where functions in the input language are mapped to functions with equivalent behavior in the target language. For instance, calls to the function `np.amax` of Python's NumPy [23] library, which retrieves the maximum value of a matrix, should be translated to calls to Julia's function `maximum`.

Type information is another important aspect of transpilation. In particular, the mapping of dynamically typed languages to statically typed ones presents a challenge, which was already addressed by some proposals ([18], [31]). Furthermore, languages such as Julia may benefit from type annotations to optimize code performance.

A transpiler can translate most use-cases automatically if the input and target languages have similar levels of abstraction. However, some cases present ambiguous translation scenarios, which we will discuss in the following section.

3 DISAMBIGUATE TRANSLATIONS

In the previous section, we discussed several conflicts that a transpiler should automatically resolve. However, there are some translation scenarios where automatic translation could result in the generation of convoluted code, which would make the mapping between the input and the generated source code fuzzy.

In particular, consider transpiling source code written in a dynamically typed language, such as Python. This scenario exposes the limitations of type inference, as we are bound to the information available at compile time. As an example, consider the following Python function and its translation to Julia:

```
def sum_two(x, y):
    return x + y
```

The function `sum_two` receives two inputs, `x` and `y` that can have arbitrary types. The main problem lies in Python's `+` operator, which applies different operations depending on the runtime types of its operands, such as integer addition or string concatenations, among other possibilities.

A possible solution to disambiguate such cases is to request the programmer to annotate the Python source code using type hints to assist the translation process. In the previous case, annotating the `x` and `y` arguments using type hints would result in a more accurate translation.

Generally, it is a good practice to annotate the arguments and return types of functions, as this conveys the programmer's intentions of the source code. Furthermore, functions such as `sum_two` are too generic to be able to infer any type information. In such cases, the transpiler requires type-hints to correctly map the operations to Julia.

4 JULIA AND PYTHON

After discussing several aspects of transpilation, it is important that we introduce the two languages that will be the focus of our project.

Python was introduced more than 30 years ago. Throughout the years, its popularity has increased among the scientific community for providing an easy learning curve and an extensive library set.

The Python programming language has many alternative implementations. Two of them are Jython [12] and IronPython [13], where the first approach compiles Python source code to Java bytecode that runs on the JVM and the latter compiles Python source code to IL bytecode for the .NET platform. However, these implementations lack support for Python 3. CPython, Python's reference implementation, is written in the C programming language and has support for Python's latest version.

However, CPython suffers from slow performance on modern hardware due to Python's implicit dynamism. Programmers who require highly efficient code usually implement a prototype in Python and then convert the kernel parts to C. Furthermore, Python's high-performance libraries, such as `NumPy`[23], are also highly optimized libraries written in C that provide a speedup when compared to Python's native implementations. This is commonly referred to as the two-language problem, where the prototyping language differs from the main implementation language.

On the other hand, we have the recently introduced Julia programming language, which has been proving to be a high-performance alternative to Python, aiming at solving the two-language problem. Julia's simple syntax combined with high performance on modern hardware makes it a great alternative to Python.

Nowadays, two critical factors for the success of programming languages are the quality and quantity of available libraries. This problem was acknowledged in the context of Common Lisp [19], which, despite being a high-performance and flexible language, did not become popular due to its absence of libraries and the difficult mechanisms used to integrate them. Julia has a good library integration mechanism, which has incentivized the development of many third-party libraries. However, the available library set is still small, which is an issue we plan to address.

5 PYJL

The development of this project is based on an existing solution called Py2Many [29], which includes the PyJL transpiler. Py2Many provides a generic architecture and implements the necessary transpilation mechanisms to transpile Python to many C-like languages.

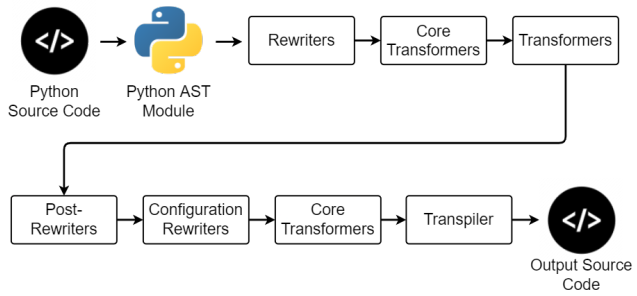

Figure 1: PyJL Architecture

PyJL builds upon that architecture and defines its transpiler implementation to translate Python to Julia. We opted to use Py2Many, since it has an active community updating it. Additionally, our preliminary analysis of the frameworks' architecture shows that it is a good starting point to support this project.

Our implementation of PyJL [22] is still in its initial development stages and far away from our goal of automating the translation of Python libraries to Julia. The following section describes the current state of PyJL. We also analyze a performance scenario and detail our future plans for this project.

5.1 PyJL Architecture

In this section, we describe the stages of the transpilation pipeline used in PyJL. PyJL uses the same architecture as the Py2Many framework and adds the necessary functionality to transpile Python source code to Julia. The language-independent stages apply transformations common to all languages and are not extended by PyJL. The current pipeline can be seen in figure 1.

The input of this pipeline is Python source code that is parsed using Python's `ast` module[1], which generates an Abstract Syntax Tree (AST). All the Phases in the Pipeline receive an AST as their input and use the visitor pattern to visit and modify nodes. We now describe each phase in the transpilation process:

(1) *Rewriters* can be both language-specific or -independent and perform modifications in select nodes of the AST. A common use case of Rewriters is to change the structure of nodes to match the target language.
(2) *Core Transformers* are language-independent transformers that modify the AST with relevant information for the translation process. The added information includes:
   (a) Variable context: Adds all the variables to the node that represents their scope.
   (b) Scope context: Adds a scope attribute to each node in the AST.
   (c) Assignment context: Adds information to node assignments. An example is to annotate nodes that are on the left-hand side of an assignment, which is useful for operations that want to verify a node's position in later phases.
   (d) List call information: Adds all list transformation operations to the scope of the variable referencing the list.
   (e) Variable Mutability: Analyzes functions to detect mutable variables.
   (f) Nesting levels: Annotates nodes with the respective nesting levels. This is important for languages sensitive to white spaces.
   (g) Annotation flags: Differentiates type annotations and nested types
(3) *Transformers* are language-specific and add complementary information to specific nodes of the AST. An example would be to add type information to nodes to help with 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* is an addition made to the Py2Many pipeline for PyJL, which supports configuration files in JSON and YAML format to modify the AST. This stage is language-specific.
(6) *Transpiler* translates language syntax and semantics and converts the AST to a string representation in the target language using the information provided by the previous phases. It is language-specific.

In the pipeline, the *Core Transformers* phase executes at two different stages. The first makes this information available for the stages that perform intermediary transformations. The second guarantees that no intermediary transformation overwrites the core functionalities of Py2Many and makes them available in the *Transpiler* phase.

After the pipeline has processed the Python source code, it generates the equivalent source code in the target language. The current implementation supports the transpilation of one or more files and performs the changes synchronously.

5.2 Code Annotations

The PyJL transpiler currently has a simple mechanism to allow programmers to specify code annotations separate from the Python source code. The goal is to support updates to the input source code while separately preserving the annotations that affect the transpilation process. This mechanism is integrated in the *Configuration Rewriters* phase of the Pipeline and reads YAML or JSON files that contain annotations, adding them to the corresponding AST nodes. This mechanism also supports the use of annotations in specific scopes, where a programmer can, for example, declare a decorator for a function within a specific class.

This is very beneficial when translating Python libraries to Julia. If a programmer annotates a Python library directly, this process will have to be repeated every time a new library version is released. By separating annotations from the source code, their application is ensured in subsequent translations.

An addition that is being considered is the integration of a Domain Specific Language (DSL) in PyJL, such as LARA [25], which was designed to be language-independent while offering more precise code annotations. Similar to the previous approach, the annotations are separate from the source code.

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

5.3 Improving Compatibility

To maximize interoperability, programming languages commonly offer Foreign Function Interfaces (FFIs) to call externally developed modules or libraries. For instance, Julia's PyCall[15] provides an FFI to interoperate with Python. Tools such as PyonR [27] use Racket's FFI to call Python functions, and benchmarks reveal that it is a high-performance alternative to mapping Python's data model in Racket.

Despite FFI's presenting an alternative to translation, we argue that the translation process brings more benefits in the case of Julia. Using an FFI results in less maintainable source code, as external calls use a different language syntax. Furthermore, many of Python's highly-optimized libraries already have dedicated alternative implementations in Julia. Translated libraries would preserve Julia's syntax and benefit from Julia's performance on modern hardware.

Regarding the translation process, we found some cases that offer no direct mapping from Python to Julia. An example is the translation of Python's generator functions, which return a lazy iterator that implements the producer/consumer pattern. The producer generates a new value whenever `yield` is called and saves its execution state. When the consumer requests a value, the generator resumes its execution from the saved state. To demonstrate this use-case, we present an implementation of the Fibonacci sequence that returns an infinite iterator:

```
def fib():
    a = 0
    b = 1
    while True:
        yield a
        a, b = b, a + b
```

The producer/consumer pattern can be implemented in Julia using channels. The producer uses the `put!` function to add values to the channel while the consumer uses the `take!` function to retrieve them. We include a possible implementation below:

```
function fib()
    Channel() do ch
        a = 0
        b = 1
        while true
            put!(ch, a)
            a, b = b, a + b
        end
    end
end
```

Despite the syntactic similarities, there is an important difference. Even with the use of unbuffered Channels, the execution will only block at the first call to `put!`, allowing side effects in the producer to be executed before the consumer requests the first value.

A possible alternative that preserves Python's behavior is to use the third-party package `ResumableFunctions` [17]. This package defines a `@resumable` macro that is used to simulate the behavior of generator functions in Julia. A `@yield` macro is used to replace Python's `yield` keyword. Similar to Python, this implementation uses a Finite State Machine to save the execution state and resume it in subsequent calls. An equivalent implementation of the Fibonacci sequence using this package is the following:

```
@resumable function fib()
    a = 0
    b = 1
    while true
        @yield a
        a, b = b, a + b
    end
end
```

Besides preserving Python's behavior, this approach also has the benefit of mapping more directly to its equivalent Python implementation, resulting in improved readability.

Another approach we found that helps the translation process is to use Julia's macro capabilities to map Python functionalities. We are experimenting with the development of a `dataclass` macro, that currently offers preliminary support for Python's `dataclass` decorator in Julia. However, we need to account for the fact that Julia's macros are expanded at macro-expansion time, which happens at compile time, while Python decorators operate dynamically at runtime. We are assessing the limitations of such implementations.

We considered both of these approaches in the development of our transpiler and found measurable improvements. Whenever possible, the transpiler should default to using Julia's native constructs. However, we recognize that translating Python's behavior may require the addition of new functionalities in Julia or the use of third-party packages. When the transpiler requires the use of these mechanisms to ensure correctness, it should always inform the programmer by producing a corresponding log message.

5.4 Object Mapping

Python allows programmers to use functional programming. However, it also supports the use of the OO Paradigm. Julia, on the other hand, is a mostly functional programming language, that does not fully support the OO paradigm. For the development of PyJL, we considered mapping Python's classes to Julia using native Julia constructs.

Translating Python's class model to Julia represents a tradeoff. A positive aspect is that we preserve the intended behavior of Python programs in Julia. However, this could potentially introduce a high overhead in computations, due to the added indirection of having class representations. We believe that a correct approach that also offers a choice to programmers is to support two alternative aproaches:

(1) The first uses Julia constructs to create a class hierarchy mechanism
(2) The second relies on the use of a third party package called `Classes` [26].

The first approach converts classes into mutable structs, which have the corresponding fields of the class. It also creates abstract types for each class that are extended by each corresponding struct. The methods of each class are translated with a `self` field as their first parameter, which extends the abstract type mapped to the corresponding Python class.

The second approach uses the aforementioned `Classes` package. This package contains the `@class` macro, which defines a hierarchy

of abstract types and creates the necessary functions for each type, including a constructor function. This is a method of automating the previously introduced solution, with the benefit of having a simpler syntax.

To visualize both mechanisms, we provide a simple class inheritance example written in Python:

```python
class Person:
    def __init__(self, name:str):
        self.name = name

    def get_id(self) -> str:
        return self.name

class Student(Person):
    def __init__(self, name:str, student_num:int):
        super().__init__(name)
        self.student_num = student_num

    def get_id(self) -> str:
        return f"{self.student_num} - {self.name}"
```

In this example, we define the `Person` and the `Student` classes. `Student` extends the `Person` class and adds the `student_num` field and a new definition of the `get_id` function. A possible translation to Julia using the first approach is the following:

```julia
abstract type AbstractPerson end
abstract type AbstractStudent <: AbstractPerson end

mutable struct Person <: AbstractPerson
    name::String
end
function get_id(self::AbstractPerson)
    return self.name
end

mutable struct Student <: AbstractStudent
    name::String
    student_num::Int
end
function get_id(self::AbstractStudent)
    return "$(self.student_num) - $(self.name)"
end
```

As was previously mentioned, this involves the creation of one abstract type for each Python class. In this case we have both the `AbstractPerson` and `AbstractStudent` abstract types which are inherited by the `Person` and `Student` structs respectively. The functions include a `self` parameter, which has the type of the corresponding abstract type. The argument types allow Julia's dispatch mechanism to select the correct function when performing calls.

The second approach uses `Classes.jl` to generate Julia source code that maps much more directly to Python. This package also uses abstract types to define its hierarchy, which are generated when using the `@class` macro. In the previous example, we chose the names of the abstract types to match the names of the generated abstract types used in the `Classes` package, to allow for an easier evaluation of the generated code. The following represents an equivalent translation using this package:

```julia
using Classes

@class mutable Person begin
    name::String
end
function get_id(self::AbstractPerson)
    return self.name
end

@class mutable Student <: Person begin
    student_num::Int
end
function get_id(self::AbstractStudent)
    return "$(self.student_num) - $(self.name)"
end
```

This approach offers a more direct mapping between the Python and the Julia source code. However, it still discloses some parts of the underlying Julia mechanism. For instance, notice how both `get_id` functions extend the types `AbstractPerson` and `AbstractStudent` to work in a class hierarchy. Still, it hides the creation of the abstract types and the creation of the structs to hold object fields, further blurring the lines between Python and Julia.

To choose between these two implementations, a programmer could use the provided annotation mechanism. The first approach would be the default implementation, as it does not require the use of a third-party library.

One important aspect that is not covered by both of these approaches is multiple inheritance. This would require implementing the C3 [2] algorithm in Julia to support Python's Method Resolution Order (MRO). For the first release of PyJL we are focused on supporting single inheritance, which already covers a large subset of Python implementations.

We also intend to cover the implementation of Python's special methods mentioned in section 2, such as `__init__` and `__str__`. These add necessary functionalities commonly used in Python. A possible solution is to extend the `Classes` package and create a new `PyClass` package that implements this functionality.

Furthermore, we also intend to map Python's default class field values. The current solution is to integrate the `Parameters` package [33], which defines a new constructor for each struct that includes default field values.

5.5 Mapping Dynamic Behavior

The mapping of operators from Python to Julia is frequently dependent on the types of its arguments. However, since Python is a dynamically typed language, this type information is only known at runtime.

A possible solution is to create new functions in Julia to simulate the behavior of Python operators. For instance, in the case of the `sum_two` function from section 3, we could map Python's `+` operator to a new `py_add` function in Julia. A similar approach was implemented in PyonR [27] to translate Python to Racket. Although this is a valid approach, the generated code would not preserve the pragmatics of Julia, which negatively affects maintainability.

An alternative solution is to use a type inference mechanism to help identify, at transpilation time, the most appropriate Julia

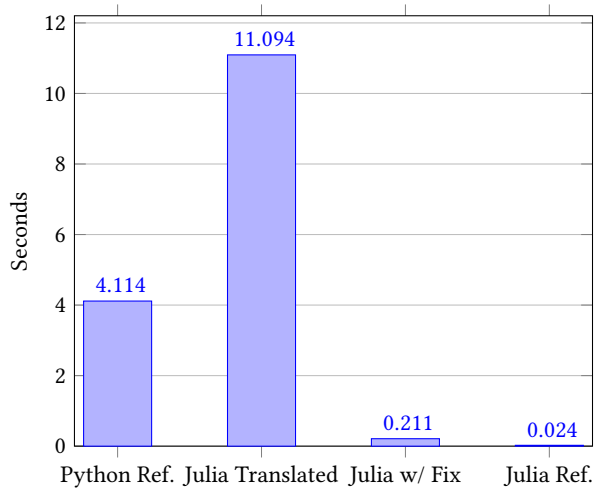

Figure 2: Performance of N-Body Implementations

operations. As these mechanisms are limited by the information available at compile time, it might be necessary to judiciously add type hints to the Python source code. This is also the case of MyPy [32], which requires type hints in function definitions to check for the soundness of the code.

A possible alternative is TYPPETE [11], which is an inference mechanism for Python based on the Z3 theorem prover [8] that uses a MaxSMT solver to define type constraints.

The approach that is currently being studied is the integration of the Pytype[10] type-inference mechanism. It generates separate `.pyi` stub files that contain type annotations. Optionally, we can merge these annotations with the original source code using the provided `merge-py` tool. This integrates newly added type information into the AST and makes the types available for the translation process.

5.6 Performance

To evaluate the current capabilities of the PyJL transpiler, an implementation of the N-Body Problem was translated to Julia. The chosen implementation predicts the gravitational interactions of planets in the solar system, and has both a Python [6] and a Julia [9] reference version. The results of the translation are publicly available[2]. The benchmarks were executed on a machine with an Intel Core i7 4790K with 16GB of RAM under Windows 10. We compared the implementations with an input value of 500000 and chose an average of 10 runs for each test. Also, the obtained output generated by the program was verified to be identical in Julia and in Python.

The performance results shown in figure 2 reveal that the initial translation is not as high-performing as Python and is orders of magnitude slower than the reference Julia implementation.

After analyzing the generated source code, we discovered that the slowdown was caused by insufficient type information. The Python function that resulted in the generation of generic Julia source code is shown below.

```
def combinations(l):
    result = []
    for x in range(len(l) - 1):
        ls = l[x+1:]
        for y in ls:
            result.append((l[x],y))
    return result
```

This function receives a list as its argument and generates combinations using the elements of that list. It then adds those combinations to a new list, which is returned by the function. The translation performed by the transpiler was the following:

```
function combinations(l)::Vector
    result = []
    for x in (0:length(l)-1-1)
        ls = l[(x+1+1):end]
        for y in ls
            push!(result, (l[x+1], y))
        end
    end
    return result
end
```

Note that the generated Julia function returns a generic array. In this case, we cannot infer the type of the returned array, as it is impossible to guarantee that the input list will always have the same type. This forces the translation to use generic containers that have considerable overheads when compared to type-specific ones. The result produced by the combinations function is not type stable, which impacts the performance of the generated source code.

After manually modifying one line of code by specifying the necessary type information, we obtained a speedup of 52.6×, making the translated Julia code 19.5× faster than the original Python code. This result can be achieved in one of two ways. We can either annotate the `result` array with its corresponding type:

```
result::Vector{Tuple{Tuple{Vector{Float64},
                           Vector{Float64},
                           Float64},
                     Tuple{Vector{Float64},
                           Vector{Float64},
                           Float64}}} = []
```

or convert the `result` array, changing the last line of the function to the following:

```
return typeof(result[1])[result...]
```

Regarding the reference Julia implementation, it is relevant to mention that it is highly optimized and takes advantage of Julia's performance characteristics.

5.7 Code Maintainability

The current status of PyJL does not allow us to make many claims on code maintainability. The generated code for the N-Body problem preserves Python's code structure and pragmatics. However, this example maps almost directly to Julia, only requiring minor syntax changes. More complex Python examples that use native Python

[2]Transpiled N-Body Problem: https://github.com/MiguelMarcelino/translated_n_body_problem

constructs with no direct translation to Julia or use Python's classes are required to analyze code readability.

We are also evaluating how the use of third party packages affects the readability of the generated source code. So far, they have shown measurable improvements and offer a better mapping between Python and Julia source code.

A proper evaluation would require user tests to determine if the generated code is intelligible by programmers. We are considering this evaluation method to assert that code generated by the PyJL transpiler is similar to human-written code.

6 EVALUATION

In the context of program translation, it is important to assess the limitations of transpilers, which will be covered in this section.

Throughout this work, we have acknowledged that there are translation cases that reach the limitations of type inference. We have previously shown two examples, the `sum_two` function in section 3 and the `combinations` function in section 5.6, where the lack of type information results in the generation of generic code. Attempting to infer types in these situations might be possible but only if bounded to a given scope, which does not guarantee overall correctness.

Another problem that we encountered was related to the reliability of type information, where some Python programs include type hints that do not match the correct attribute or variable types. One could use Pytype [10], the proposed inference tool, to perform these checks or even enforce the type annotations provided by programmers.

Regarding the mapping of Python's classes to Julia, it is important to note that translating Python's OO behavior to Julia will always inherent Julia's mechanisms. We are still relying on multiple dispatch to relate methods to object types, which implies that there is weaker coupling to objects in Julia. In the case of the translations shown in section 5.4, the Julia methods are only bound to the self argument that represents the equivalent Python Class.

7 FUTURE WORK

The PyJL transpiler is still a work in progress and far from our goal of converting Python libraries to Julia. In this section, we discuss the plans for the transpiler.

Regarding the mapping of Python's dynamic behavior to Julia, the integration of Pytype[10] should make the transpiler less dependent on type hints and help evaluate their soundness. Nonetheless, it is expected that type hints will still be necessary in function definitions due to their ambiguity.

The transpiler should also use Julia's functionalities to enhance the generated Julia source code. We have previously mentioned the creation of macros, which would result in the generation of more maintainable code. However, since macros are executed at compile-time, and due to Python's dynamism, this might only be achievable in some cases.

The performance of the generated source code is another facet of the translation process that can be optimized. Performing code optimizations is a topic which is more targeted at software restructuring tools, usually employed in software maintenance. These have the ability to change the structure of a given program without modifying its behavior [1]. The generated source code would probably benefit mostly from perfective maintenance, an approach which focuses on improving program performance or maintainability [14].

A transpiler developed for code translation can have mechanisms that account for code restructuring. The programmer could use the annotation mechanism discussed in section 5.2 to annotate code segments to restructure. The transpiler could then apply the intended code transformations during the translation process.

The restructuring process could result in improved code performance. High performance in Julia is achievable through proper techniques. We present some that were considered:

- *Separating Kernel functions*: Separate source code into different functions to allow the compiler to generate type-specific code [4].
- *Devectorizing expressions*: In Julia, loops are very well optimized, making them as fast as loops written in C. We are currently analyzing the `Devectorize` [20] package used to devectorize expressions in favor of using loops.
- *Improving Cache hit rate*: Optimize the transpiler to rewrite loops over matrices in column major order to achieve even higher performance [4].

8 RELATED WORK

In the area of source-to-source translation, many transpilers have already used Python as their source language. PyonR [27] explores the use of two complementary solutions to use Python functionalities in Racket: (1) using an FFI to call Python's C functions, (2) translating Python's data model to Racket and use the Racket execution environment. In terms of performance, calls made to the FFI ended up taking a very similar time when compared to the calls made by Python to its C API. The implemented data model also managed good performance, sometimes outperforming the equivalent CPython implementations.

Additionally, some approaches that target the improvement of Python's performance. One of these [21] explores the use of Rust as an intermediary, high-performance, and high-level language to represent Python source code. The PyRS [16] transpiler, which is now also part of Py2Many, is used to translate Python to Rust. It is an experimental transpiler that requires manual intervention in some cases to generate running Rust source code. The Rust intermediary code can then be translated to a lower-level optimized target. The performance evaluation of the transpiled code shows that the transpiled Rust source code achieves better performance while using less memory than Python.

The two-way Fortran-Python transpiler [5] also aims at improving Python's performance by using Fortran as a high-performance target language. It offers two solutions, where the programmer can either transpile Fortran code to Python and improve its performance or improve the performance of an already existing Python program. The programmer annotates the kernel functions in Python with a decorator, which are translated back to Fortran at runtime to benefit from a high-performance execution environment. The performance results are similar to those obtained when manually translating Python to Fortran.

The context of library translation has also been discussed in the development of Jnil [19], that transpiles Java to Common Lisp. This approach also studied the challenges of preserving language pragmatics in automatic translation, an important aspect for the development of PyJL.

9 CONCLUSION

Throughout the years, transpilers have evolved to generate source code that is not only human-readable, but also hard to distinguish from human-written programs, which has allowed transpilers to become alternatives to manual translation.

This work extends the PyJL transpiler to convert Python libraries to human-readable and modifiable Julia source code. The process of automating the translation represents a challenge, but it can be achieved with high-levels of reliability if enough information is provided in the Python source code. The generated code should also respect the pragmatics of Julia.

We expect that PyJL further decreases the library gap between Python and Julia, speeding up library development. The conversion subset should cover widely used Python features, such as the aforementioned `yield` construct or the `dataclass` decorator. Python's Data Model should also be mapped to an equivalent Julia Data Model. This includes mapping Python's classes to Julia, with support for single inheritance.

Furthermore, this research also aims at improving the performance of the translated libraries. Preliminary results show that Julia's compilation strategy can lead to huge performance increases when some type hints are judiciously added to the generated code.

ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) (references PTDC/ART-DAQ/31061/2017 and UIDB/50021/2020).

REFERENCES

[1] R.S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989. doi: 10.1109/5.24146.

[2] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 69–82, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 089791788X. doi: 10.1145/236337.236343. URL https://doi.org/10.1145/236337.236343.

[3] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Edinburgh International Conference Centre, Scotland, UK, 2004. Semantic Designs.

[4] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al. Julia Language Documentation - Performance Tips, 2014. Chapter 3, p.279-299.

[5] Mateusz Bysiek, Aleksandr Drozd, and Satoshi Matsuoka. Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 9–18, Salt Lake City, UT, USA, 2016. IEEE. doi: 10.1109/PyHPC.2016.006.

[6] Kevin Carson, Fredrik Johansson, Tupteq, and Daniel Nanz. N-Body Problem, Python Implementation, Mar 2021. Retrieved January 4th, 2022 from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/nbody-python3-1.html.

[7] Intel Corporation. Mcs-86 assembly language converter operating instructions for isis-ii users, 1979.

[8] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[9] Andrei Fomiga, Stefan Karpinski, Viral B. Shah, Jeff Bezanson, smallnamespaces, Adam Beckmeyer, and Vincent Yu. N-body problem, julia implementation, Apr 2021. Retrieved January 4th, 2022 from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/nbody-julia-8.html.

[10] Google. Pytype: A static type analyzer for python code, March 2015. [Online. Retrieved February 25th, 2022 from: https://github.com/google/pytype].

[11] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsmt-based type inference for python 3, 2018.

[12] Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th international Python conference*, volume 9, pages 2–18, Reston, VA, 1997. Citeseer.

[13] Jim Hugunin. IronPython Home Page, 2013. [Online. Retrieved January 6th, 2022 from: https://ironpython.net/].

[14] ISO/IEC/IEEE. International Standard for Software Engineering - Software Life Cycle Processes - Maintenance, 2006.

[15] JuliaPy. PyCall - Calling Python functions from the Julia language, February 2013. [Online. Retrieved February 25th, 2022 from: https://github.com/JuliaPy/PyCall.jl].

[16] Julian Konchunas. PyRS - A Python to Rust Transpiler, 2015. [Online. Retrieved January 18th, 2022 from: https://github.com/konchunas/pyrs].

[17] Ben Lauwens. Resumablefunctions.jl, August 2017. Retrieved on January 29th, 2022 from: https://github.com/BenLauwens/ResumableFunctions.jl.

[18] Antonio Menezes Leitao. Migration of common lisp programs to the java platform -the linj approach. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 243–251, Amsterdam, Netherlands, 2007. IEEE. doi: 10.1109/CSMR.2007.34.

[19] António Menezes Leitão. The next 700 programming libraries. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936189. doi: 10.1145/1622123.1622147.

[20] Dahua Lin. Devectorize.jl, 2015. Retrieved January 2nd, 2022 from: https://github.com/lindahua/Devectorize.jl.

[21] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. Transpiling Python to Rust for Optimized Performance. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 127–138, Cham, 2020. Springer International Publishing. ISBN 978-3-030-60939-9.

[22] Miguel Marcelino and António Menezes Leitão. Pyjl implementation, 2021. Retrieved March 5th, 2022 from: https://github.com/MiguelMarcelino/py2many.

[23] Travis Oliphant. NumPy, 2009. [Online. Retrieved November 18th, 2021 from: https://numpy.org/].

[24] Renaud Pawlak. Jsweet: insights on motivations and design a transpiler from java to javascript, 2015.

[25] Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. LARA as a Language-Independent Aspect-Oriented Programming Approach. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1623–1630, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344869. doi: 10.1145/3019612.3019749. URL https://doi.org/10.1145/3019612.3019749.

[26] Richard Plevin. Classes.jl: A simple, Julian approach to inheritance of structure and methods, November 2021. Retrieved November 22nd, 2021 from: https://github.com/rjplevin/Classes.jl.

[27] Pedro Henriques Palma Ramos and António Menezes Leitão. PyonR: A Python Implementation for Racket. Master's thesis, Instituto Superior Técnico, 2014.

[28] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, Florence, Italy, 2001. IEEE. doi: 10.1109/SCAM.2001.972664.

[29] Arun Sharma, Lukas Martinelli, Julian Konchunas, and John Vandenberg. Py2many: Python to many clike languages transpiler, 2015. Retrieved November 18th, 2021 from: https://github.com/adsharma/py2many.

[30] Roger Taylor and Phil Lemmons. Upward migration part 1: Translators using translation programs to move cp/m-86 programs to cp/m and ms-dos, 1982.

[31] Tijs van der Storm. Nomen: A Dynamically Typed OO Programming Language, Transpiled to Java, 2016.

[32] Guido van Rossum, Jukka Lehtosalo, Ivan Levkivskyi, and Michael J. Sullivan. Mypy, 2014. [Online. Retrieved February 29th, 2022 from: http://mypy-lang.org/].

[33] Mauro Werder. Parameters.jl, 2015. [Online. Retrieved February 30th, 2022 from: https://github.com/mauro3/Parameters.jl].