Extending PyJL – Transpiling Python Libraries to Julia

Miguel Marcelino
INESC-ID/Instituto Superior Técnico University of Lisbon, Portugal

António Menezes Leitão
INESC-ID/Instituto Superior Técnico University of Lisbon, Portugal

Abstract

Many high-level programming languages have emerged in recent years. Julia is one of these languages, claiming to offer the speed of C, the macro capabilities of Lisp, and the user-friendliness of Python. Julia's syntax is one of its major strengths, making it ideal for scientific and numerical computing. Furthermore, Julia's high-performance on modern hardware makes it an appealing alternative to Python. However, Python has a considerable advantage over Julia: its extensive library set.
There have been efforts to make Python libraries available to Julia either through Foreign Function Interfaces (FFI's), or through manual translation, but both have their tradeoffs: FFI's do not take advantage of Julia's performance, as they call Python's Virtual Machine, and manual translation is demanding and time-consuming.
To address these issues and bridge the gap between the two languages, we propose PyJL, a transpilation tool that converts Python source-code to human-readable Julia source-code. Although the development of PyJL is still at an early stage, our preliminary results reveal that the generated code follows the pragmatics of Julia and is capable of high performance.

2012 ACM Subject Classification Software and its engineering → Source code generation; General and reference → Cross-computing tools and techniques

Keywords and phrases Source-to-Source Compiler, Automatic Transpilation, Library Translation, Python, Julia

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/MiguelMarcelino/pyjl_translations`

Funding 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).

1 Introduction

Transpilation and compilation share identical mechanisms for processing source code, but they have two distinct goals: compilers convert an input language to a lower-level language, such as machine code, while transpilers translate it to another language with a similar level of abstraction.

Transpilers were initially developed to convert lower-level source code. The first transpiler, CONV86 [6], was developed in 1978 by Intel to translate assembly source code from the 8080/8085 to the 8086 processor, providing compatibility between an 8-bit and a 16-bit processor. Nowadays, as most modern programming languages are high-level languages, it makes sense to develop transpilers that operate at this level. As an example, consider Babel.js [1], a transpiler that converts newer versions of ECMAScript, such as ES6, to ES5.

Modern programming languages became more dependent on the quality and quantity of available libraries, which highly influence its adoption. One solution to this problem is to translate libraries from more established languages to newer and/or less popular ones [15]. However, manual translation requires an extensive amount of time and resources. On the other hand, using a transpiler to automate this process can significantly speed up the translation of libraries between languages.

Automatic translation is challenging, as different languages offer distinct syntactic and semantic constructs. Additionally, if transpilers work with static source code information, dynamic languages become considerably difficult to translate. All of these challenges influence the translation's quality.

There have been previous efforts regarding the transpilation of Python source code to Julia. For instance, Py2JL[13] aims at translating Python source code to human-readable Julia source code. Similarly to the work presented in this paper, it uses rule-based transpilation to generate Julia source code. However, it can only translate simple Python code excerpts. Thus, having a tool that can operate on larger code samples could benefit the community.

This paper extends our previous work on translating Python libraries to Julia [17]. PyJL is a rule-based transpilation tool that translates Python source code into human-readable Julia source code. Our previous evaluation of PyJL concluded that it is possible to translate Python source code to Julia with minimal programmer intervention. Still, the transpiler required more improvements regarding the generation of pragmatic code. Furthermore, with improved translation coverage of Python's standard library, we can now perform a more in-depth analysis of the transpilers abilities.

2 Related Work

With the rise of many new high-level programming languages, translating between them has become an increasingly important topic. Some programming languages are inclusively using transpilers to maximize compatibility with existing languages. This is the case of TypeScript [11], which transpiles to JavaScript and offers an improved syntax and additional functionalities.

The topic of library translation was also discussed when developing LinJ [15], which transpiles Common Lisp source code to Java. This work describes how the syntactic and semantic incompatibilities of these two languages make in very hard to automatically translate between them. This becomes increasingly more difficult if we want to preserve the pragmatics of the target language.

Regarding the transpilers that use Python as a source language, PyRS is a transpiler that translates Python to Rust and is also part of Py2Many [21]. It currently requires manual intervention in some cases to generate running Rust source code. The Fortran-Python transpiler [4] uses Python's type hints to translate Legacy Fortran to Python and vice versa, while producing human-readable code. It does not intend to entirely automate the translation process, instead requiring manual intervention in certain cases. There are also transpilers, such as Prometeo [23], which transpile Python source code to high-performance C source code. However, the generated code is not human-readable, an important aspect of this work.

Transpilers that translate from Julia are less common, as the language is also very recent. We previously mentioned Py2JL[13], which also transpiles Python to human-readable Julia source code. Other transpilation approaches are not concerned with human-readability.

More generic tools include TXL [5], which was designed to restructure and modify source code. Other notable tools include DMS [2], which targets automatic management and improvement of source code in large software solutions.

3 Automatic Translation

Automatically generating code that preserves the pragmatics of the target language is challenging, requiring the transpiler to at least use proper language constructs and suitable code formatting with appropriate use of indentation rules. In practice, the translation process

requires a careful analysis of syntactic and semantic incompatibilities that can be difficult to overcome. For instance, in the case of Python and Julia, the semantics of language constructs often require specific mappings. Python's operators must be carefully mapped to Julia, as they use overloading and apply different operations depending on their operator types.

Additionally, there is the problem of memory management, in which a transpiler must account for memory allocations and remove unused objects in memory, which may require the use of a garbage collection mechanism. This is important for transpilers such as ts2c [18], which convert JavaScript and TypeScript to human-readable C source code.

Lastly, one should also consider the difference between dynamically and statically typed languages. In a statically typed language, such as Java, all types have to be defined at compile time, while in a dynamically typed language, types are determined at runtime. Translating a dynamically typed language to a statically typed one requires a type inference mechanism. However, the reliability of static type-inference mechanisms largely depends on the type information available at compile time. Therefore, a transpiler might require restrictions regarding which types must be available at compile time. Moreover, some languages, such as Julia, also benefit from type annotations to provide better performance. That is the case of Julia's Arrays [3], where memory allocation can largely be improved if type annotations are used, allowing for contiguous element allocations and less boxing/unboxing operations.

4 Translating Python to Julia

In the previous section, we highlighted several aspects of automatic translation that make the conversion of syntax, semantics, and generation of pragmatic code more difficult. However, Python and Julia have similar levels of abstraction, which might indicate that the translation of Python libraries to Julia can occur with minimal programmer intervention. In this section, we will briefly introduce both languages.

Python was introduced in 1991, and has gained popularity in recent years due to its extensive library set and its use in Data Science. It is worth noting that there are several implementations of Python. Its reference implementation is CPython, which was created by Guido van Rossum in 1989 and is written in the C programming language. Two common alternatives to CPython are Jython [9] and IronPython [10], both developed by Jim Hugunin. The first compiles the input Python source code to JVM bytecode for the Java platform, and the latter compiles it to IL bytecode for the .NET platform. However, these implementations currently lack support for Python's latest version.[1] Furthermore, there is also PyPy [20], an implementation of Python using a Just-In-Time compiler written in RPython.

For the purpose of this research, we will use CPython, as it is the reference implementation. However, CPython suffers from slow performance due to Python's implicit dynamism. A common problem of CPython is known as the two language problem, which occurs when the prototype language differs from the main implementation language. Programmers who require high-performance typically convert the kernel parts of their programs to C, using Python only as a prototyping language.

On the other hand, Julia promises to solve the two language problem and is proving to be a high performance alternative to Python. It is one of the few languages that belongs to the Petaflop club, along with C, C# and Fortran. However, its library set is still reduced, particularly when compared to more established languages, such as Python. We plan to address this issue by speeding up the library development in Julia.

[1] As of April 2022, IronPython supports version 2.7.11 (version 3.4 is still an Alpha release) and Jython supports version 2.7.2

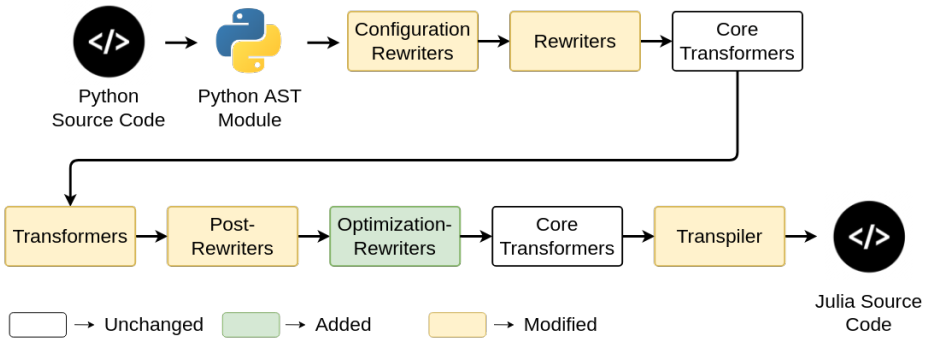

Figure 1 PyJL Architecture.

5 PyJL

PyJL[2] is part of the Py2Many [21] transpiler, which is a rule-based transpilation tool that 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. Figure 1 shows the architecture of the PyJL transpiler. The changed and added phases show up with different colours for distinction.

The transpiler first parses the input Python source code using Python's `ast` module,[3] 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. *Configuration Rewriters* support configuration files in JSON and YAML format that specify AST modifications.
2. *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.
3. *Core Transformers* are language-independent transformers that add relevant information to nodes in the AST. An example is to add scope context to the AST's nodes, allowing for node searches.
4. *Transformers* are language-specific and add information to nodes in the form of attributes. An example would be to add type annotations for type inference.
5. *Post-Rewriters* are rewriters that have dependencies on some previous phase. Their functionality is identical to the *Rewriters* phase.
6. *Optimization Rewriters* are rewriters that optimize a small set of the generated source code. This is similar to peephole optimization, commonly used in compilers.
7. *Transpiler* translates language syntax and semantics, and converts the AST to a string representation in the target language.

Notice from figure 1 that the *Core Transformers* phase executes at two stages. The first makes core information available to the *Transformers*, *Post-Rewriters* and *Optimization-Rewriters* stages. The second ensures that core Py2Many transformations are not overwritten, making them available in the *Transpiler* phase.

[2] PyJL Development Repository: `https://github.com/MiguelMarcelino/py2many` Retrieved April 24th, 2022

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

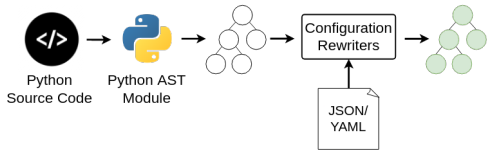

Figure 2 Annotation pipeline.

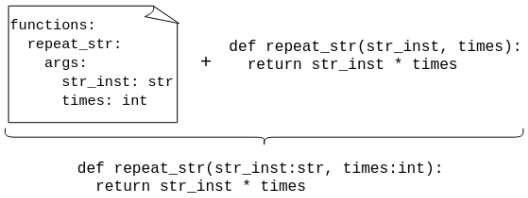

Figure 3 Annotation Example.

There are scenarios where the transpiler provides more than one translation method. In these scenarios, the programmer can decide which alternative to use by using one of three methods: (1) manually annotating the Python source code, (2) using JSON or YAML annotation files, supported by the *Configuration Rewriters* phase, or (3) using flags to make global changes to source code generation. These approaches provide the flexibility to cover a broader range of translation scenarios, allowing the programmers to selectively apply annotations if necessary.

To demonstrate the use of the annotation mechanism, we show the processing pipeline in more detail in figure 2. It parses the provided YAML/JSON files and adds the information to the AST. The current supported features are adding type hints or decorators to function definitions. An example of code annotations can be seen in figure 3. In this example, we can use the annotations provided from the YAML file and merge them with the `repeat_str` function. This is equivalent to manually annotating the code.

We will now describe the changes to the transpiler. Section 5 describes the most notable improvements. In section 6, we compare the performance of several translation scenarios and provide an optimization use-case, where we analyse the steps required by the programmer to improve the performance of the generated code. Lastly, section 7 describes some limitations of the transpiler and translation methodology applied.

5.1 Scoping Rules

The first mismatch between Python and Julia is related to the scoping rules, which define the behaviour of assigning names to values and solve possible conflict scenarios. Both Python and Julia use lexical scoping, which determines, at compile-time, the section in the source code where a name is bound to a value. This section analyses the different scoping rules and how they affect the translation process.

In Python, scopes are defined according to the *LEGB* rule [16, Ch. 16], which stands for Local, Enclosing, Global, and Built-in scopes. Local scopes define the scope of a Python function or lambda expression. Enclosing scopes define the outer scope of a nested scope. The Global scope is the top scope of a Python module. Lastly, the Built-in scope contains automatically loaded special keywords, such as built-in functions, exceptions, etc. In Python, this rule is used when searching for an unqualified name. The search for a name reference starts on the Local scope, following the *LEGB* order, and stops at the first encounter of that name.

On the other hand, in Julia, scopes can either be global or local. Furthermore, Julia's local scopes are divided into hard and soft scopes [14, Ch. 10]. To explain this concept, let us consider that a variable named `a` is defined in the global scope: if the enclosing scope is a hard scope and if there is an assignment to a new local variable `a`, then a new local variable will be created and will shadow the global variable; if all the enclosing scopes are soft scopes, the behaviour changes when used in non-interactive or interactive (REPL) contexts.

Listing 1 Python Mandelbrot.

```
def mandelbrot(limit, c) -> int:
  z = 0 + 0j
  for i in range(limit + 1):
    if abs(z) > 2:
      return i
    z = z * z + c
  return i + 1
```

Listing 2 Julia Mandelbrot.

```
function mandelbrot(limit, c)::Int
  z = 0 + 0im
  i = 0
  for _i = 0:limit
    i = _i
    if abs(z) > 2
      return i
    end
    z = z * z + c
  end
  return i + 1
end
```

In non-interactive contexts, if an assignment to a new local variable `a` occurs, it will shadow the global variable similarly to the hard scope, the only difference is that it emits a warning when shadowing occurs. In interactive contexts, the global variable is always assigned. If there is no global variable with the same name, then a new local variable will be created both in hard and soft scopes.

Global scopes include `module`s and `baremodule`s. Local soft scopes include `struct`s, `for`, `while` and `try`, while local hard scopes include `macro`s, `function`s, `do` `block`s, `let` `block`s, `comprehension`s, and `generator`s. Furthermore, Julia's constructs are only allowed in certain scopes. These scoping rules impose some limitations in the translation process.

For instance, one of Julia's scoping restrictions is that `struct`s can only be defined in the global scope. As an example, the transpiler generates `struct`s to translate Python classes. However, classes in Python can be defined in local scopes. Automatically changing the scope of classes could potentially result in name clashes. In addition, this might not match the programmer's intent of the code. Therefore, we include a `remove_nested` Python decorator that the programmer can use to annotate the classes that should be moved to the global scope.

This problem also affects the `resumable` macro, which is used by the transpiler to simulate Python's generators in Julia. This macro defines a Finite State Machine to simulate Python's generator functions. To save the machine's state, it creates a `struct`, restricting its use to the global scope. To account for these cases, we have added an optional argument field `remove_nested` to the `resumable` decorator, which has a similar functionality as the previously defined decorator.

Another mismatch occurs with control flow operators. Despite their syntactic similarities in Python and Julia, they have considerable differences in their scoping rules. As an example, translating Python's loops to Julia could potentially result in errors if loop target variables are used outside its body. To detect these cases, the transpiler analyses the enclosing scope to find any assignments that have the same variable name as any of the loop target variables. This enforces the concept of for-loop scopes during transpilation and can then be used in two different ways:

1. Have the transpiler emit a warning message when loop target variables are used outside the loop's scope.
2. Create a new variable in the enclosing scope and update it in every iteration of the loop.

To demonstrate a use-case where this occurs, consider the `mandelbrot` function shown in listing 1 that tests if a complex number c belongs to the Mandelbrot set by computing the number of iterations required (up to a given `limit`) to get a value greater than 2.

Listing 3 Python Augmented Assignments.

```
1 x = [1,2]
2 y = x
3 x += [3,4]
4 x[1:2] *= 2
5 y[1:2] += [1]
```

Listing 4 Augmented Assignment Expansion.

```
x += [3, 4]
     ↓
x = x + [3, 4]
     ↓
x = append!(x, [3, 4])
```

Notice how the loop variable `i` is used outside the scope of the loop. This is valid in Python, as the loop does not define its own scope, but cannot be translated directly to Julia. By using code analysis and applying the second translation method, the transpiler generates the code shown in listing 2, which now works in Julia with the intended result.

5.2 Augmented Assignments

Augmented assignments combine binary operations with assignment statements. These are supported by both Python and Julia, but given Python's use of operator overloading, not all augmented assignments can be translated directly to Julia.

As an example, consider the code excerpt in listing 3. The first two assignments are equivalent in both languages. However, executing the third statement in Julia does not yield the same results, as it performs an element-wise addition instead of the concatenation that is done in Python. To solve this problem, the transpiler expands augmented assignments into assignments and binary operations and then transpiles the result.

The conversion of this statement can be seen in listing 4. Notice how the function `append!` was used, which performs an in-place concatenation of elements to the list `x`, matching the behaviour of Python's augmented assignments [22, Ch. 7].

Another mismatch occurs when translating *slices* from Python to Julia. Lines 4 and 5 of listing 3 represent that scenario. To translate these augmented assignments into Julia, one can use the `splice!` function, which replaces or inserts new elements in a given list. The translated Julia source code for the previous example is the following:

```
splice!(x, 2:2, repeat(x[2:2], 2))
splice!(y, 3:2, [1])
```

Notice that the second call to `splice!` uses the form `n:n-1` for the second argument, which inserts a new element in the list [14, Ch. 42].

5.3 Subscripts

In Python, a subscript is used to represent indexing and slicing on sequences and key lookups on mapping types. Translating subscripts to Julia becomes a challenge when trying to generate pragmatic code. Furthermore, there are several cases that require special handling, which will be discussed in this section.

Indexing is used to look up a particular position in a sequence, i.e., tuples, lists, strings, etc. The main difference between indexing in Python and Julia, is that Python uses 0-based indexing while Julia uses 1-based indexing. Indexing can be performed using integer literals or generic expressions. Integer literals can be incremented to match Julia's 1-based indexing, but non-literal expressions require adding the literal 1, which can reduce the readability of the code. However, because most indexing is performed in loops, we can optimize the entire scenario instead.

Listing 5 Julia Combination Sort.

```
def comb_sort(
    seq: List[int]) -> List[int]:
  gap = len(seq)
  swap = True
  while gap > 1 or swap:
    gap = max(1, floor(gap / 1.25))
    swap = False
    for i in range(len(seq) - gap):
      if seq[i] > seq[i + gap]:
        seq[i], seq[i + gap] = \
          seq[i + gap], seq[i]
        swap = True
  return seq
```

Listing 6 Julia Combination Sort.

```
function comb_sort(
    seq::Vector{Int})::Vector{Int}
  gap = length(seq)
  swap = true
  while gap > 1 || swap
    gap = max(1, floor(Int, gap / 1.25))
    swap = false
    for i = 0:length(seq) - gap - 1
      if seq[i + 1] > seq[i + gap + 1]
        seq[i + 1], seq[i + gap + 1] =
          (seq[i + gap + 1], seq[i + 1])
        swap = true
      end
    end
  end
  return seq
end
```

For instance, consider the implementation of the combination sort algorithm in listing 5. The simplest translation, as shown in listing 6, is to preserve the ranges and adjust the indexing operation. As an alternative, we provide two additional optimization methods that the programmer can use:

1. Determine if operations in loops are only performed on sequences, and increment loop ranges.
2. Use the OffsetArrays package [12] to define custom index ranges for sequences.

The first approach requires analysing the source code to verify if the loop ranges can be optimized. The transpiler validates the applicability of this optimization by analysing if all nodes in the loop's body using its target variables are Subscript nodes. To demonstrate this translation method, we used the transpiler to translate the Python combination sort implementation by changing the loop ranges instead of the indexes. The transpilation result, shown in listing 7, respects the pragmatics of Julia, being much closer to what a Julia programmer would write.

The second translation method, which uses OffsetArrays, allows defining arrays with the same index ranges as Python. The code generated by the transpiler is available in listing 8. The call to `OffsetArray` creates a wrapper around the array `seq` and decreases its indexing value by 1, which is equivalent to performing 0-based indexing. In this case, the transpiler used a *let*-block, to restrict the wrapping of the input vector `seq` to the block's scope.

Both alternatives have tradeoffs. The first is arguably more pragmatic in this particular example, but changes the algorithm's implementation to use 1-based indexing, which might not be desired for all scenarios. The second preserves the program's original indexing, but can be a less pragmatical solution in some cases. The programmer can select his preferred method, as demonstrated in the beginning of section 5.

Another scenario is the use of subscripts with slices. In Julia, slices are called `UnitRange`s and are very similar to those of Python. One notable exception is that in Python, slices have an inclusive beginning and exclusive end, whereas in Julia, both are inclusive, requiring the transpiler to adjust the ranges. Furthermore, Python allows slices that do not explicitly define a beginning or end index, whereas Julia's `UnitRange`s require all values to be specified. An example of this can be found below, where the Python code excerpt on the left can be translated to the corresponding Julia code on the right:

Listing 7 Julia Optimized Indexing.

```
function comb_sort(
    seq::Vector{Int})::Vector{Int}
  gap = length(seq)
  swap = true
  while gap > 1 || swap
    gap = max(1, floor(Int, gap / 1.25))
    swap = false
    for i = 1:length(seq) - gap
      if seq[i] > seq[i + gap]
        seq[i], seq[i + gap] =
          (seq[i + gap], seq[i])
        swap = true
      end
      end
  end
  return seq
end
```

Listing 8 Julia Offset Arrays.

```
function comb_sort(
    seq::Vector{Int64})::Vector{Int64}
  let seq = OffsetArray(seq, -1)
    gap = length(seq)
    swap = true
    while gap > 1 || swap
      gap = max(1,floor(Int, gap / 1.25)
          )
      swap = false
      for i = 0:length(seq) - gap - 1
        if seq[i] > seq[i + gap]
          seq[i], seq[i + gap] =
            (seq[i + gap], seq[i])
          swap = true
        end
      end
    end
  end
  return seq
end
```

```
l = [1,2,3,4]
a = l[1:]
b = l[:-1]
```

```
l = [1,2,3,4]
a = l[2:end]
b = l[begin:end-1]
```

Notice that the last line covers the translation of negative indexing. Currently, PyJL only supports negative indexing if the index is a literal.

Lastly, subscripts can be used for key lookups, where a common scenario is a dictionary lookup. In practice, distinguishing subscripts that use indexing from ones that use keys depends on the container's type. The transpiler currently relies on the type-inference mechanism to infer the container types and transpile the corresponding subscripts to Julia. The inference mechanism is described in section 5.5.

5.4 Functions

Both in Python and Julia, functions are first-class values. Most notably, they can be returned, passed as arguments, and assigned to variables. Some functions defined in Python can be mapped to equivalent functions in Julia. However, even if their behaviour is equivalent, they may differ in terms of the argument order or even argument count. For instance, consider the Python code excerpt on the left and its corresponding translation to Julia found on the right:

```
write = sys.stdout.buffer.write
write(b"test")
```

```
write = x -> Base.write(stdout, x)
write(b"test")
```

If no argument is provided, the Python built-in function `write` will output the content to `stdout`. However, in Julia, the equivalent function requires an explicit parameter indicating where to write the contents to. The transpiler translates such cases using lambda expressions to automatically input the default parameters.

Notice how in the translated code above, the assignment to variable `write` overwrites any future calls to Julia's built-in `write` function within the scope where it is defined. To prevent this issue, every time an assignment name clashes with one of Julia's built-in functions,

we translate function references using Julia's `module.name` notation. This can be seen in the example above, where `sys.stdout.buffer.write` is translated to `Base.write` in Julia, mitigating the name clashes.

5.5 Type Inference

Having a type-inference mechanism is crucial to transpile Python source code to Julia, as the translation outcome might depend on the type information available. Py2Many already offers a type-inference mechanism, which we extended with new inference rules. This section briefly describes the current inference mechanism.

The inference mechanism in Py2Many is implemented using a define-use set. This mechanism recursively walks the AST and aggregates type information from node assignments for each scope. The defined scopes are more extensive than Python's scopes, to cover the scoping rules of all supported languages. The current scopes include `modules`, `functions`, `classes`, `for` and `while` loops, `if`-statements, and `with`-statements.

We extended the current inference mechanism with a more extensive rule-set. The changes include adding more information to binary operations, as their translation to Julia is largely dependent on the types of their operands. This involves adding type annotations to the left and right operands, to support more complex operations.

Similarly to MyPy, PyJL also requires programmers to annotate function definitions. PyJL uses these definitions to mitigate type instability and to translate operations that depend on the operand types.

Lastly, PyJL strictly enforces static typing using Python's type hints. Therefore, any user-annotated variables are only allowed to have one type within their scope. As an example, consider the following assignment operations in Python:

```
l: List[str] = ["a", "c", "g", "t"]
...
l = "acgt"
```

In this case, the transpiler will reject the second assignment to variable `l`, as the type of its value does not match the previous type annotation. Alternatively, if the value of the second assignment is another variable, the transpiler will search for the variable's type. If the type was provided by a type hint and does not match the previous annotation, the transpiler will reject the assignment.

6 Performance

To evaluate the performance of the generated code, we chose three benchmarks: (1) the binary trees benchmark, which tests the garbage collection mechanisms of each language by allocating short-lived trees and traversing them, (2) the sieve of Eratosthenes, which evaluates the performance of Julia's iterators and compares the performance of the generated code to a high performance NumPy implementation, and (3) the fasta benchmark, which generates random DNA sequences using a Linear Congruential Generator (LCG), testing the performance of IO operations and generator functions.

For all benchmarks, we have chosen a reference version, representing the best-case single-threaded implementation. Julia's reference version for the fasta benchmark was simplified by removing the use of threads. We measured the results using the `bencher` benchmarking tool [8]. The results were measured on a machine with an Intel(R) Core(TM) i7 4790K @4.4GHz

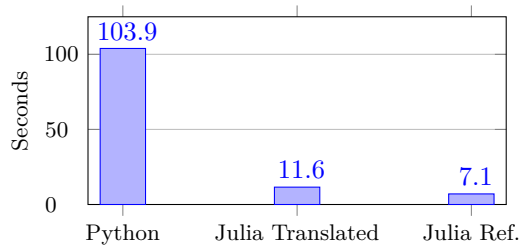

Figure 4 Binary Trees Benchmark.

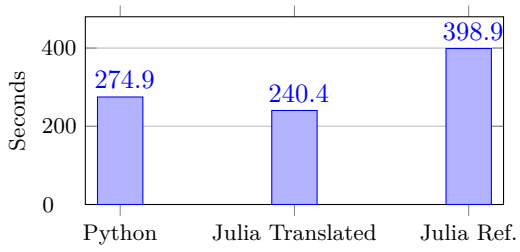

Figure 5 Binary Trees Memory Allocation.

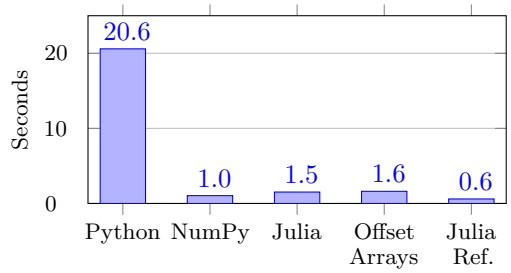

Figure 6 Sieve Benchmark.

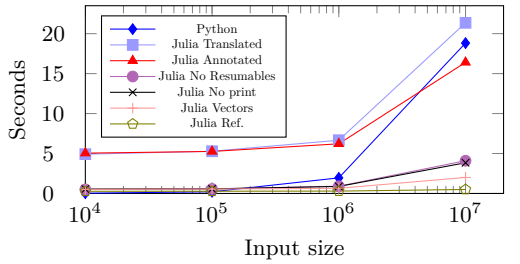

Figure 7 Fasta Benchmark.

with 16GB of RAM under Linux. We used an input of 21 for the binary trees benchmark, and 100,000,000 for the sieve benchmark. The fasta benchmark was tested with varying input sizes. The results of the translation are publicly available.[4]

From figure 4, we observe that translating Python's implementation of the binary trees benchmark to Julia yields a 9× faster execution time. Both the Python reference version and the generated Julia code use a similar amount of memory, as can be observed in figure 5, measured at 275MB and 240MB respectively. The reference Julia version manages a 1.64× faster execution time compared to the translated version, although it also uses 1.66× more memory.

Regarding the sieve benchmark, figure 6 shows the obtained results, where the generated Julia source code is 13.5× faster than the Python implementation. We also translated this implementation using OffsetArrays, mentioned in section 5.3, where the results only differ by 6% from the translated code. Nonetheless, Python also allows programmers to use NumPy [19], a scientific computing library implemented in C, to speedup code execution. The NumPy implementation is 1.46× faster than the generated code. Julia's reference implementation still manages a faster execution time, but it uses sophisticated techniques, such as loop unrolling, and exploits cache allocation.

Lastly, the results of the fasta benchmark can be seen in figure 7. As the Python implementation uses generator functions, we have chosen to translate this benchmark using the Resumables package, which simulates the use of Python's generator functions in Julia. Similarly to Python, it uses a finite state machine to save the generator's state. However, despite its similarities, the translated version is slightly slower across all input ranges. This is especially noticeable with smaller inputs from 10^4 to 10^6. We now describe possible steps a programmer can follow to improve the performance of the translated code.

[4] PyJL Benchmarks Repository: `https://github.com/MiguelMarcelino/pyjl_benchmarks` (Retrieved June 6th, 2022)

To discover what is causing the slowdown, the first step is to look for unannotated code. After a closer inspection of the fasta benchmark, we determined that there was one instance where this occurred. The translated Julia function can be found below, for which we added the type annotations manually:

```
1  function makeCumulative(table)
2      P::Vector{Float64} = []
3      C::Vector{String} = []
4      prob = 0.0
5      for (char, p) in table
6          prob += p
7          P = append!(P, [prob])
8          C = append!(C, [char])
9      end
10     return (P, C)
11 end
```

This function calculates cumulative probabilities from the predicted probabilities of choosing each nucleotide in a DNA sequence, represented by the `table` argument. A list `P` is used to store the cumulative probabilities, and a list `C` is used to store the nucleotides. Both lists are translated into generic arrays, which have large overheads in Julia, resulting in less efficient code due to excessive boxing and unboxing. The annotations on lines 2 and 3 solve this problem. As seen in figure 7, the changes are more noticeable with larger inputs of 10^7, where the execution time is 1.3× faster than Python.

Next, we measured the overhead of using resumables. In this case, as it is only required to save a seed value, we can use a memory reference holding that value, similar to what the Julia reference version does. Removing the use of resumables resulted in much faster execution times across all input ranges. This lead us to conclude that the slowdown observed with the smaller inputs was likely caused by the initial operations required to setup the finite state machine.

Another aspect that can be improved is IO performance, as the fasta benchmark outputs large nucleotide sequences to the standard output. Python's `print` function was translated by the transpiler to Julia's equivalent `println` function, which has notable overheads when called repeatedly, as it does not buffer the data, issuing more calls to operating system functions that require expensive context-switching operations. This can be improved by using Julia's `write` function, which, similarly to Python's `println` function, buffers the data and reduces the number of operating system calls. This improvement is more noticeable with the larger input size of 10^7, resulting in a 1.06× faster execution time. Despite being a small improvement, this becomes more noticeable with input values larger than 10^7.

Finally, we can use Julia's more efficient data types and convert Python's strings into vectors holding `UInt8` values. This not only makes the implementation more efficient, but also reduces the amount of memory required to allocate the nucleotide sequences. These changes are more noticeable with input sizes of 10^6 and 10^7, resulting in a further 1.36× and 1.91× improvement, respectively.

Julia's reference implementation is still faster than our optimized fasta implementation, as it stores and retrieves the random nucleotide sequences in a more efficient way. Instead of the `makeCumulative` function used by Python, Julia builds a lookup table that holds all the nucleotides in their respective positions, given the calculated cumulative probabilities. It then relies on indexing to retrieve the nucleotides. In contrast, Python uses the `bisect_right` function to locate the appropriate nucleotides, which has more overheads and results in slower execution times when translated to Julia.

7 Future Work

The changes made to PyJL bring it closer to our goal of automatically translating Python libraries to Julia. However, there are still limitations, which we discuss in this section.

The transpiler currently analyses modules independently, without considering module-level dependencies. Py2Many already uses a topological sort algorithm to sort all modules according to their import dependencies, but still requires a more sophisticated mechanism to analyse the program's context as a whole.

Another aspect that has to be addressed is the handling of exceptions. As an example, consider Python's `ValueError` exception, which is raised when an argument has the correct type but an incorrect value. Two Python calls that throw this exception are `math.sqrt(-1)` and `float("-")`. In Julia, the first call throws a `DomainError` exception, but the second call throws an `ArgumentError` exception. Therefore, an exception in Python does not have a deterministic corresponding exception in Julia. Furthermore, some Python exceptions, such as the `ZeroDivisionError` that is raised in Python when the quotient of a division operation is zero, are not considered as exceptions in Julia. Julia instead returns `Inf`, which represents infinity. Given that the transpiler performs static analysis, it will not be possible to detect such situations in advance, and it is not pragmatic to generate code that does that at runtime. Such cases remain a topic for future work.

Regarding Py2Many's type-inference mechanism, it is currently rather conservative when annotating generic containers, which largely impact the performance of the generated Julia source code. This is due to its limitations regarding intra-procedural analysis, which considers the whole program's context. An external type inference mechanism supporting intra-procedural analysis, such as pytype [7], could potentially increase the available type information at transpilation time.

Lastly, the readability of the generated code is an aspect that requires a more extensive evaluation. We are currently in the process of preparing user tests, but these still require a thorough analysis and remain a topic for future studying.

8 Conclusions

This work aims to automate the translation of Python libraries to Julia to increase Julia's library set. The generated code should conform to Julia's pragmatics, allowing Julia programmers to further maintain it.

In this paper, we presented recent additions to our previous work on translating Python libraries to human-readable and modifiable Julia source code. In particular, we further automated the translation of source code by covering a more extensive subset of Python. The readability and pragmatics of the generated code were also improved using code analysis, making it harder to distinguish from human-written code.

Automatically converting Python source code is challenging due to the semantic differences between Python and Julia. Python's dynamic typing further makes this a difficult process, as types are not available at compile time. The improved inference mechanism now covers a broader range of scenarios, but this is still a limitation when automatically converting Python source code. Requiring type-hints in function definitions mitigates most of these limitations.

As demonstrated by our performance results, the PyJL transpiler requires little programmer intervention to generate high-performance Julia source code. Furthermore, as the generated source code is human-readable, programmers can further optimize and maintain it.

References

1 Babel. Babel, September 2012. Retrieved April 7th, 2022 from: https://github.com/babel/babel.

2 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.

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

4 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, 2016. doi:10.1109/PyHPC.2016.006.

5 James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology, 44(13)*, pages 827–837, 2018. doi:10.1016/s0950-5849(02)00104-0.

6 Intel Corporation. MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users, 1979.

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

8 Isaac Gouy. The Computer Language Benchmarks Game, 2007. Retrieved April 16th, 2022 from: https://salsa.debian.org/benchmarksgame-team/benchmarksgame.

9 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.

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

11 Philip Japikse, Kevin Grossnicklaus, and Ben Dewey. *Introduction to TypeScript*. Apress, 2017. Chapter 7. doi:10.1007/978-1-4842-2478-6.

12 JuliaArrays. Offsetarrays.jl, January 2014. Retrieved April 11th, 2022 from: https://github.com/JuliaArrays/OffsetArrays.jl.

13 JuliaCN. Py2Jl, 2018. [Online. Retrieved 19th April, 2022 from: https://github.com/JuliaCN/Py2Jl.jl].

14 JuliaLang. The julia language - 1.7.3, May 2022.

15 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. doi:10.1145/1622123.1622147.

16 Mark Lutz. *Learning Python - 3rd Edition*. O'Reilly Media, Inc., 2007.

17 Miguel Marcelino and António Menezes Leitão. Transpiling Python to Julia using PyJL. In *Proceedings of the 2022 European Lisp Conference*. Zenodo, March 2022. doi:10.5281/zenodo.6332890.

18 Andrei Markeev. ts2c: JavaScript/TypeScript to C Transpiler, 2016. [Online. Retrieved March 30th, 2022 from: https://github.com/andrei-markeev/ts2c].

19 Travis Oliphant. NumPy, 2009. [Online. Retrieved May 30th, 2022 from: https://numpy.org].

20 Armin Rigo. Pypy, 2007. [Online. Retrieved April 17th, 2022 from: https://www.pypy.org/].

21 Arun Sharma, Lukas Martinelli, Julian Konchunas, and John Vandenberg. Py2many: Python to many CLike languages transpiler, 2015. Retrieved April 22nd, 2022 from: https://github.com/adsharma/py2many.

22 Guido van Rossum and Python Development Team. *The Python Language Reference - Release 3.9.7*, August 2021. [Online. Retrieved April 9th, 2022 from: https://docs.python.org/release/3.9.7/].

23 Andrea Zanelli, Tommaso Sartor, Peter Bowyer, and Dominik Moritz. Prometeo - An experimental Python-to-C transpiler, 2017. [Online. Retrieved 19th October, 2022 from: https://github.com/zanellia/prometeo].