TÉCNICO LISBOA

PyJL: A Source-to-Source Python Compiler to Julia

Miguel Alexandre da Costa Martins Marcelino

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof.José Faustino Fragoso Femenin dos Santos

October 2022

Agradecimentos

Agradeço...

Ao Prof. António Leitão, por me permitir fazer parte do grupo ADA de desenho algorítmico e pelos seus conselhos eruditos, que me incentivaram a superar as minhas expectativas.

Aos meus pais e ao meu irmão, que me apoiaram durante estes 5 anos de formação superior, permitindo-me alcançar os meus objetivos. Agradeço também os essenciais momentos de lazer e por me aturarem nos meus momentos menos áureos.

À minha namorada, Rafaela, que me motivou e apoiou em cada passo do meu trabalho. Agradeço também pelos passeios e pela paciência que teve comigo nestes tempos.

Ao Grupo de Arquitetura (Inês Pereira, João David, Inês Caetano, Renata Castelo Branco) pelos seus imensuráveis conselhos em apresentações e artigos científicos.

À ”Equipa Maravilha” (Vasco Castro, David Pereira, Ana Sofia Fernandes, Samuel Ferreira, Andreia Batista, Diogo Soares, João Preto), que me providenciou com momentos de procrastinação valiosos e que me apoiou durante todo o processo académico.

À Fundação para a Ciência e Tecnologia (FCT) e ao INESC-ID pelo acolhimento num estágio enriquecedor e pelo financiamento no âmbito dos contratos PTDC/ART-DAQ/31061/2017 e UIDB/50021/2020.

Por fim, ao Instituto Superior Técnico, pela valiosa experiência que me providenciou e pela sua cultura de determinação, que me fez enfrentar todos os meus desafios com rigor e exigência.

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 math-friendly syntax is one of its most prominent strengths, making it ideal for scientific and numerical computing. Furthermore, Julia's performance on modern hardware makes it an appealing alternative to Python. However, Python has a considerable advantage over Julia: its extensive library set.

Python libraries can be made available to Julia through Foreign Function Interfaces (FFI's) or manual translation. Both of these approaches have their tradeoffs: FFI's do not take advantage of Julia's performance, and manual translation is demanding and time-consuming. In this regard, transpilation is a promising option. Transpilers translate between high-level programming languages, providing an efficient alternative to manually porting software from one language to another.

To speedup the development of Julia libraries, we propose extending the Py2Many transpiler to translate Python source code into human-readable and maintainable Julia source code. Our results reveal that the generated code is capable of high performance and follows the pragmatics of Julia, allowing Julia programmers to further optimize and maintain it.

Keywords

Source-to-source Compiler, Transpiler, Library Translation, Python, Julia

Resumo

Recentemente têm emergido muitas linguagens de programação de alto nível, sendo o Julia uma destas linguagens, proclamando ter o mesmo desempenho que a linguagem C, as capacidades de processamento de `macros` da linguagem LISP e ser tão fácil de usar como a linguagem Python. Para além disto, o Julia é também uma linguagem apelativa para computação cientifica e numérica, devido à sua sintaxe intuitiva para representar operações matemáticas. Estas capacidades, com o elevado desempenho em hardware moderno, fazem do Julia uma alternativa à linguagem Python.

No entanto, o Python oferece um conjunto mais vasto de bibliotecas quando comparado com a linguagem Julia. Atualmente, um programador que necessite de usar uma biblioteca Python pode recorrer ao uso de Foreign Function Interfaces (FFIs) ou proceder à tradução manual dessa biblioteca para Julia. Ambas as alternativas têm os seus aspetos negativos. Ao usar FFIs, um programador não beneficia do elevado desempenho do Julia. No que toca a tradução manual, este é um processo demorado. Este problema pode ser mitigado com um compilador source-to-source, que traduz código entre duas linguagens de programação de alto nível, sendo uma alternativa eficaz à tradução manual entre linguagens.

Para acelerar o processo de tradução de bibliotecas, propomos estender o transpilador PyJL para traduzir código Python para código legível em Julia. Os resultados obtidos revelam que o desempenho do código gerado pelo transpilador é elevada e que o código gerado é legível, o que permite que programadores Julia possam estender ou melhorar o código.

Palavras Chave

Compilador Source-to-source, Transpilador, Tradução de bibliotecas, Python, Julia

Contributions

Some of the work presented in this thesis has also been published in two papers and a conference, namely:

- *Transpiling Python to Julia using PyJL*, published in the 15th European Lisp Symposium (ELS'22) [1]
- *Extending PyJL - Transpiling Python Libraries to Julia*, published in the Symposium on Languages, Applications and Technologies (SLATE'22) [2]
- *Extending PyJL to Translate Python Libraries to Julia* - JuliaCon 2022

Contents

1 Introduction **1**

1.1 Language Conversions . . . 2

1.1.1 Language Mismatches . . . 3

1.2 Python . . . 4

1.3 Julia . . . 4

1.4 Objectives . . . 5

2 Related Work **7**

2.1 LinJ . . . 8

2.2 Fortran-Python Transpiler . . . 8

2.3 JSweet . . . 9

2.4 Py2Many . . . 10

2.5 Clava . . . 10

2.6 Further Mentions . . . 11

2.7 Analysis . . . 11

3 Transpilation **15**

3.1 The Py2Many Transpiler . . . 16

3.1.1 Scoping Mechanism . . . 18

3.1.2 Code Annotation Mechanism . . . 19

3.1.3 Parsing Mechanism . . . 19

3.2 Python to Julia Translation . . . 20

3.2.1 Augmented Assignments . . . 20

3.2.2 Boolean Operations . . . 21

3.2.3 Loops . . . 21

3.2.4 Indexing . . . 22

3.2.5 Generator Functions . . . 24

3.2.6 Arbitrary-Precision Arithmetic . . . 24

3.2.7 Simulating Python's OO Implementation . . . 25

3.2.8 Special Methods and Attributes . . . 29
3.2.9 Scoping Rules . . . 29
3.2.10 Keyword Arguments . . . 32
3.2.11 Other Incompatibilities . . . 32
3.3 Optimizations . . . 32
3.3.1 Removing Redundant Operations . . . 33
3.3.2 Optimizing Global Variables . . . 33
3.4 Validating Translations . . . 33
3.4.1 The *unittest* Framework . . . 34
3.4.2 Parameterized Unit Tests . . . 35

4 Type Inference **37**
4.1 Py2Many's Inference Mechanism . . . 38
4.1.1 Import Analysis . . . 39
4.1.2 Static Type-Checking . . . 39
4.1.3 Limitations . . . 40
4.2 External Type Inference Mechanism . . . 41
4.2.1 Advantages . . . 42
4.2.2 Limitations . . . 43
4.3 Why two Mechanisms? . . . 43
4.4 Alternative Solutions . . . 44

5 Dependencies **45**
5.1 Python and Julia's Import Mechanisms . . . 46
5.2 Importing Local Modules . . . 47
5.2.1 Module Dependencies . . . 48
5.3 Importing Registered Modules . . . 49
5.3.1 Simulating Package Calls . . . 49
5.3.2 Using *PyCall* . . . 51
5.4 Name Aliases . . . 53
5.5 Accessing Dynamic-Link Libraries . . . 53
5.5.1 Translation Methodology . . . 53
5.5.2 Additional Functionalities . . . 54

6 Evaluation **57**
6.1 Evaluating Translation Correctness . . . 58
6.2 Performance . . . 58
6.2.1 N-Body-Problem . . . 59

6.2.2 Fasta . . . 60
6.2.3 Sieve . . . 61
6.2.4 Sieve Numpy . . . 62
6.2.5 Neural Network . . . 62
6.2.6 Binary Trees . . . 63
6.3 Library Translations . . . 64
6.3.1 The *python-reprojector* Library . . . 64
6.3.2 The *pywin32-ctypes* Library . . . 64
6.4 Evaluating Code Pragmatics . . . 65
6.5 Automatic vs Manual Translation . . . 67
6.6 Extensibility . . . 68

7 Conclusions **71**
7.1 Coverage . . . 73
7.2 Future Work . . . 74

Bibliography **77**

A Julia Classes using ObjectOriented.jl **83**

B Performance Benchmarks **85**
B.1 N-Body Julia Translation . . . 86
B.2 Fasta Julia Translation . . . 88
B.3 Binary Trees Julia Translation . . . 90
B.4 Neural Network . . . 91

List of Figures

3.1 Py2Many Architecture 17
3.2 Inference tree-walk 18
3.3 Annotation pipeline 19
3.4 Annotation Example 19
3.5 Class hierarchy 28

4.1 Pytype inference 42

5.1 Dependency Scenarios 47

6.1 N-Body Benchmark 60
6.2 Fasta Benchmark 60
6.3 Sieve Translation 62
6.4 Sieve NumPy Translation 62
6.5 Binary Trees 63
6.6 Neural Networks 63
6.7 Python Experience of Participants 65
6.8 Julia Experience of Participants 65
6.9 Code Readability and Pragmatics Evaluation 65

List of Tables

2.1 State-of-the-art Evaluation 11

6.1 Time in minutes for manual translation 68

7.1 Updated State-of-the-art Evaluation 73

Listings

3.1 Python Augmented Assignments 20
3.2 Translation to Julia 20
3.3 For-Else Python 21
3.4 For-Else Julia Translation 21
3.5 Python Combination Sort 22
3.6 Julia Combination Sort 22
3.7 Julia 1-Indexed Arrays 23
3.8 Julia Offset Arrays 23
3.9 Generator Functions 24
3.10 Julia Channels 24
3.11 Resumable Functions 24
3.12 Python Class Hierarchy 26
3.13 Julia Class Hierachy 27
3.14 Julia *Classes* Package 27
3.15 Python Multiple Inheritance 28
3.16 Julia Multiple inheritance 28
3.17 ObjectOriented External Functions 29
3.18 ObjectOriented Nested Functions 29
3.19 Dynamic Class Attributes in Julia 30
3.20 Python Mandelbrot 31
3.21 Julia Mandelbrot 31
3.22 Python Indexing 33
3.23 Julia Indexing 33
3.24 Julia Optimized 33
3.25 Python Unittest Excerpt 35
3.26 Julia Unittest Excerpt 35
3.27 Python Parameterized Test 36

3.28 Julia Parameterized Test 36

4.1 Bonacci Series Python 40
4.2 Bonacci Series Julia 40
4.3 Optional Return Types 43
4.4 Generic Type Annotations 43

5.1 Python Import 48
5.2 Julia Import 48
5.3 NumPy Array Representation 50
5.4 Translation to Julia 50
5.5 Sieve NumPy 51
5.6 Sieve Julia Translation 51
5.7 Python Module Import 51
5.8 Julia PyCall Import 51
5.9 Python reprojector excerpt 52
5.10 Julia reprojector translation 52
5.11 Levenshtein Module Python 54
5.12 Levenshtein Module Julia 54
5.13 Function Factory Python 55
5.14 Function Factory Julia 55

6.1 Combinations Function Python 59
6.2 Combinations Function Julia 59
6.3 Python Cumulative Probabilities 60
6.4 Julia Cumulative Probabilities 60
6.5 Python Sieve 61
6.6 Julia Sieve 61
6.7 Julia Offset Arrays 61
6.8 Sigmoid Functions Python 62
6.9 Sigmoid Functions Julia 62
6.10 Binomial Coefficient Python 66
6.11 Binomial Coefficient Julia 66

Acronyms

AST Abstract Syntax Tree

OO Object Oriented

PyPI Python Package Index

FFI Foreign Function Interface

MRO Method Resolution Order

IDE Integrated Development Environment

DSL Domain Specific Language

DLL Dinamic Link Library

LCG Linear Congruential Generator

GC Garbage Collection

FST Full Syntax Tree

1

Introduction

Contents

1.1 Language Conversions 2
1.2 Python 4
1.3 Julia 4
1.4 Objectives 5

In recent years, we have seen the rise of many new high-level programming languages, such as Rust, Go, TypeScript, and Julia. These programming languages offer inherent qualities that benefit programmers. However, nowadays, the success of a programming language is more dependent on the libraries it offers, which makes it critical to speedup the development of such libraries.

There are several reasons to translate a code base between two languages. On the one hand, translating between languages is a way to preserve software written in older languages while offering access to modern functionalities. On the other hand, translating libraries from more established languages to newer ones that do not yet offer equivalent functionalities speeds up language development. Therefore, converting code between languages is essential to ensure its maintainability and increase language functionalities. The following section discusses this topic and highlights the importance of automating the translation process.

1.1 Language Conversions

Converting a program written in one language to an equivalent one written in another language is a challenging process. The most important aspect of the conversion process is to keep the external behaviour of the original program [3]. Manually converting large code bases is a difficult task and requires substantial resources. Furthermore, if the code base receives constant updates, manual translation becomes almost impossible, requiring constant readjustments to the translated code. The solution is to automate this process to reduce the time and complexity of translations.

A particular case of automatic language conversion occurs when translating between languages with similar levels of abstraction. This process is called transpilation and is performed by a tool called a transpiler. A transpiler is a tool that takes input source code written in a programming language, called the input language, and produces output source code written in the same or in a different programming language, called the output or target language. Each transpilation tool is built with a different goal. DMS [4] is a tool that focuses on the Design Maintenance of large software solutions. Other tools in the area of Safety-Critical Computing [5] use Source Code Manipulation to implement fault-tolerance mechanisms. In the context of this research, we focus on the topic of Source-to-Source translation.

Throughout the years, transpilers have adapted to the changing programming language landscape. The first transpiler was developed in 1978 to provide compatibility between an 8-bit and a 16-bit processor. It was called CONV86 [6] 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 [7]. Nowadays, with programmers developing software in higher-level programming languages, it makes sense to have transpilers operate at this level.

Regardless of the abstraction level a transpiler is working on, the goal is to automate the transla-

tion process, requiring as little programming intervention as possible. This becomes challenging when the aim is to preserve the pragmatics of the target language, requiring the transpiler to, at least, use appropriate language constructs and suitable code formatting.

The translation process becomes even more difficult if a transpiler has to translate between dynamically and statically typed languages. In a statically typed language, such as Java, types are known at compile time, while in a dynamically typed language, type-checking occurs at runtime. Converting between these languages would likely require the use of type inference mechanisms. However, the reliability of static type inference largely depends on the type information available at compile time. Therefore, a transpiler might still impose restrictions regarding type information that must be available at compile time.

1.1.1 Language Mismatches

Syntactic differences are simple to translate from an input to an output language. However, most languages have important semantic differences between them. Furthermore, if both the input and the output languages support different constructs or paradigms, these must be simulated to achieve a similar behaviour.

As an example, consider translating a C++ program that contains a `switch` statement to Julia, which does not support it. Such cases can be simulated using other control-flow operators, such as `if` statements. Alternatively, Julia also offers third-party packages, such as the *Switch*[1] package, that can also be used to simulate this functionality.

As another example, consider translating Python's decorators to Julia. One possible solution would be to develop macros that simulate Python's decorators and offer similar functionality. 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. Such differences could further complicate the translation process.

Furthermore, the source and target languages might promote different programming paradigms, making the translation process more challenging. As an example, consider translating a program written in Java, to a language such as Rust, which does not fully support the Object Oriented (OO) paradigm. A transpiler aiming to translate between these two languages would have to simulate inheritance using Rust's language features, e.g., using the `inheritance`[2] crate to simulate Java's class inheritance features.

Despite the usefulness of language conversions, many mismatches occur when aiming for a high semantic equivalence between the input and the output language [3]. However, given the similarities

[1] Switch.jl - C-style switch statement for Julia: https://github.com/dcjones/Switch.jl (Retrieved September 8th, 2022)

[2] Rust `inheritance` crate: https://docs.rs/inheritance/0.0.1-alpha.2/inheritance/index.html (Retrieved October 25th, 2022)

of Python and Julia, it might be possible to create an automated translation tool between these two languages. The following sections discuss the differences between Python and Julia and highlight their relevance in the context of this work.

1.2 Python

The Python programming language is now more than 30 years old. Currently, it has the highest rating of 12.5% on the TIOBE index [8], which measures the popularity of programming languages. Its rapid ascent in popularity is partially motivated by its vast amount of libraries. The Python Package Index (PyPI) [9], a package repository for Python, currently registers more than 400.000 projects. Furthermore, in his Python guide to Lisp programmers, Peter Norvig commends Python for being a very pedagogical language [10], making it ideal for students learning their first programming language.

It is important to mention that Python has several alternative implementations. Two of them are Jython [11] and IronPython [12], both implemented by Jim Hugunin in 2000 and 2006, respectively. The first approach compiles Python source code to Java bytecode that runs on the JVM, thus benefiting from Java's portability and performance. The latter was written in C# and compiles Python source code to IL bytecode for the .NET platform, benefiting from the compatibility with .NET libraries. However, neither of these implementations supports Python's latest version.[3]

Regarding Python's reference version, it is called CPython, since it is written in the C programming language [13]. It was initially developed by Guido Van Rossum and is currently being maintained by the Python Software Foundation (PSF).[4] A noteworthy aspect is that 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. This is commonly referred to as the two-language problem, which refers to the general rule that an easy to use language is difficult for a computer to run and vice-versa.

1.3 Julia

The recently introduced Julia programming language has been proving to be a high-performance alternative to Python, aiming at solving the two-language problem. It is one of the four languages, along with C, C++, and Fortran that belong to the *petaFLOP Club*, achieving over 1 petaFLOP per second. Besides having good performance on modern hardware, it also offers a math-friendly syntax, which benefits numerical computing. However, Julia's popularity on TIOBE is 0.5%, which pales in comparison to

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

[4] Python Software Foundation (PSF): https://www.python.org/psf/

Python's 12.5%. Julia's general registry [14] currently holds 7400 registered packages, which is almost two orders of magnitude smaller than Python.

To benefit from external libraries written in other languages, Julia provides a Foreign Function Interface (FFI) that directly accesses libraries compiled from C or Fortran programs. Julia's FFI is also critical to support higher-level packages, such as PyCall [15], JavaCall [16] or RCall [17] that interoperate with programs written in, Python, Java, and R, respectively. When using this approach, one must consider that there is additional latency. If the language aims to provide type safety, additional type validation processes can further impact the latency of calls.

Despite the usefulness of FFI's for simple function calls, the benefits quickly diminish in more complex scenarios. As an example, dealing with compound data types might require serialization/deserialization of objects, which is cumbersome if it has to be performed manually. When dealing with callbacks, one must also guarantee that any objects passed to foreign functions are not unexpectedly freed by a Garbage Collection (GC) mechanism. This might require keeping track of object references to ensure they remain available while in use. Additionally, programs that largely depend on FFI's are less intelligible than equivalent ones that use native language features [18]. This leads to reduced code usability and maintainability.

1.4 Objectives

Having good quality libraries is a critical factor for the success of programming languages. This problem was acknowledged in the context of Common Lisp [18], where absence of libraries and the difficult mechanisms used to integrate them contributed to its decreasing popularity over the years. Given Python's large library set when compared to Julia, having a tool capable of translating a subset of Python to Julia could bring large benefits to the Julia community. On the other hand, programmers could potentially benefit from improved performance due to Julia's optimization techniques.

This brings us to our objective, which is to implement a transpiler that accepts a large subset of Python and translates it to pragmatic and human-readable Julia source code. There are already transpilers from Python to Julia, namely Py2JL [19], which converts a small subset of Python to human-readable Julia source code. However, its development has been suspended in favour of the Py2Many transpiler. Py2Many is a transpiler from Python to many C-like programming languages. Julia is amongst the supported target languages, but only a small subset of Python's features are supported. We chose Py2Many as our base implementation, as its architecture offered a good range of flexibility and included a regression test suite to ensure translation correctness.

Before starting the development of the transpiler, we established four goals that we intended to accomplish. For each goal, we devised an evaluation method to assess the transpiler's capabilities. The

following outlines our goals and evaluation methods:

1. Correctness: Translation correctness was tested by using Unit tests. The transpiler translated Python's testing methodology to an equivalent model in Julia. We evaluated correctness by extending Py2Many's unit test suite and cover a subset of CPython's unit tests.

2. Intelligibility and Pragmatics: Generating source code that respects the pragmatics of Julia is a primary concern. The generated code should also be maintainable, allowing programmers to improve it in future iterations. Code intelligibility was evaluated by experienced programmers through an anonymous survey.

3. Performance: The runtime performance of the generated code should at least match that of Python. We intend to select Python performance benchmarks and compare them to manually written Julia implementations.

4. Dependencies: The transpiler should be capable of mapping Python's imports to Julia. Furthermore, the translated Julia code should also be able to access shared libraries. The latter should be evaluated by translating a library that accesses Dynamic Link Libraries (DLLs).

Our implementation of Py2Many is publicly available [20]. This implementation is currently being integrated into the official repository. We decided to support Python 3, more specifically, version 3.9, as the majority of Python users have now transitioned from Python 2 to this new version. For Julia, we chose to use version 1.7.3.

After establishing the goals and evaluation methods, we can now describe the development of the transpiler and discuss the results obtained. We assume that the reader is familiar with the basics of Python and Julia. There will be detailed explanations for the more complex functionalities of both languages.

In Chapter 2, we will discuss alternative transpiler solutions. Chapter 3 discusses our implementation and how we mapped the most notable language mismatches. Chapter 4 discusses the inference mechanism. Chapter 5 discusses the supported dependency scenarios, including the translation of Python source code that interfaces with DLLs. Chapter 6 evaluates the results obtained from the translations. Lastly, chapter 7 discusses the future work and presents our conclusions on the topic.

2

Related Work

Contents

2.1 LinJ 8
2.2 Fortran-Python Transpiler 8
2.3 JSweet 9
2.4 Py2Many 10
2.5 Clava 10
2.6 Further Mentions 11
2.7 Analysis 11

There are many transpilation tools translating between high-level programming languages and some offer novel methods that were particularly useful in the context of this research. This section presents a selection of the current state-of-the-art transpilation tools.

2.1 LinJ

Introduced in 2007, LinJ [21] is a transpiler that generates Java code from Common Lisp code. It uses the host Common Lisp implementation to parse the code in three steps [21]:

1. The Common Lisp reader deals with *readmacro* expansions and builds a low-level AST.
2. LinJ reparses the AST using a Java-inspired grammar, producing a higher-level AST.
3. LinJ walks the high-level AST recursively to generate Java code.

During this process, the Linj translator performs several transformations to the AST. As an example, since Java does not support shadowing of local variables, variables declared in an inner scope bearing the same name as variables in an outer scope are renamed [21]. Other important changes include analysing control-flow paths to determine where to include return statements in the Java code and performing the necessary unboxing/boxing and cast operations [21].

Another aspect to consider are type annotations, that are optional in Common Lisp but mandatory in Java. This is solved through a type inference mechanism that computes a definition-use set, relating parameters with their respective uses in method bodies while finding the most specific type that satisfies them.

The goal of Linj was to generate human-readable code. To test code readability, industrial projects were translated from Common Lisp to Java and their quality was assessed by Java programmers, which also proved the effectiveness of the tool with large codebases. The end results showed that Linj was able to generate human-readable code, in some cases being interchangeable with human written code [21]. Code pragmatics is crucial for our tool, as we intend for programmers to use the generated code as if it was written by seasoned programmers. As such, the choice of what constructs to use becomes crucial, as it can severely affect the generated code's readability and pragmatics.

2.2 Fortran-Python Transpiler

The Fortran-Python transpiler [22], which was introduced in 2016, converts Legacy Fortran source code to Python source code and vice-versa, leveraging the type hints mechanism introduced in Python 3.5. This transpiler does not aim at fully automating the process of translation, intentionally requiring manual

intervention in some cases. The proposed solution supports two use cases, both leveraging just-in-time compilation [22]:

1. Migrating high-performance Fortran source code to Python.
2. Enhance the performance of existing Python source code.

The first solution starts by translating the Fortran source code to Python. Then, both solutions require the user to mark any computing kernels in the Python source code, which trigger a translation to Fortran at runtime. Calls made to Python functions identified as computing kernels are wrapped by a new interface, which transfers them to the generated Fortran functions. When their execution finishes, the return values are forwarded back to Python. This strategy allows the core of the application logic to run in Fortran, while the remaining code runs in Python [22].

Similarly to Jnil, this work presents valuable insight into the translation of a dynamically typed to a statically typed language. Furthermore, this tool imposes several restrictions during the translation process. For instance, the transpiler is unable to translate Python's dynamic behaviour, which allows variables to be assigned with values of different types. Such aspects are less relevant when translating between dynamically typed languages, but present challenges for static analysis mechanisms [23]. Therefore, some restrictions implemented in the Fortran-Python transpiler, such as requiring programmers to annotate function definitions, might be required for our work, as it might not be possible to infer a function's return type or its argument types at compile time.

2.3 JSweet

JSweet [24] transpiles Java to JavaScript using a two-step process:

1. Transpiles the Java code to TypeScript.
2. Uses *tsc* [25, Ch 10] to transpile TypeScript into JavaScript.

JSweet also translates TypeScript definition files to JSweet libraries, making them available to Java programmers. These are taken from the DefinitelyTyped[1] repository and are updated before starting the transpilation process.

The output of transpilation is the generated JavaScript source code and the source map files. The latter are auxiliary files mapping element positions between the generated JavaScript code and the Java code. This is important for debugging, as JavaScript is compacted into a much shorter file, losing its original structure.

[1] DefinitelyTyped repository: `https://github.com/DefinitelyTyped/DefinitelyTyped` (Online, Retrieved 26th July, 2022)

Besides bridging the gap between Java's Integrated Development Environments (IDEs) and modern web development, JSweet allows access to JavaScript libraries, as they provide important functionalities required by programmers. This emphasizes the idea that libraries are a crucial aspect of languages. Being able to interface with existing libraries is the main focus of JSweet, which also translates to our work, with regard to accessing Python's libraries from Julia.

2.4 Py2Many

The Py2Many transpilation framework is a tool written in Python that is aimed at translating Python source code into C-like programming languages. Its development began with a transpiler that translates Python to C++, but was then expanded to include several other programming languages, such as Rust, Dart, Go, Kotlin, Nim, V, and Julia, with different degrees of correctness and completeness.

Each language is supported through an extension. PyRS is currently the most developed transpilation extension, which transpiles Python to Rust. It is still an experimental solution that, in some cases, requires manual intervention to generate running Rust source code. PyRS was used in an approach [26] that explores Rust as an intermediate source code step to compile Python to a low-level target language. The runtime performance of the generated Rust source code was superior to Python's while using less memory.

For our research, we were particularly interested in PyJL, which is an extension to Py2Many's framework that translates Python source code to Julia. Unfortunately, PyJL was only capable of transpiling simple Python examples, failing to translate non-trivial cases. Additionally, crucial functionality was missing, such as the ability to properly translate Python's classes to Julia.

2.5 Clava

Clava is a source-to-source compiler whose development started in 2016 [27] and allows developers to perform software refactoring. It uses the LARA Domain Specific Language (DSL) to specify strategies for efficient code generation. Clava's source code processing has the following stages [27]:

1. A C/C++ Frontend parses the input source code, which is comprised of two components:
   (a) The *Clava AST Dumper* uses Clang[2] to create a dump containing syntactic and semantic information from the input source code.
   (b) The *Clava AST Loader* parses the dump and builds an instance of a *Clava AST*.

[2]Clang compiler for the C language family `https://clang.llvm.org/` (Retrieved October 16th, 2022)

	Input Language	Target Language	Intelligible	Coverage	Upkeep	Performance
Linj	Common Lisp	Java	✓✓✓	✓✓	✓	$5-6\times$
Fortran-Python	Python	Fortran	✓✓	✓✓	✓	$6-10\times$
JSweet	Java	JavaScript	✓✓	✓✓	✓✓✓	✗
Py2Many/PyJL	Python	Julia	✓	✓	✓✓✓	✗

Table 2.1: State-of-the-art Evaluation

2. The *LARA engine* interprets and parses LARA strategies that contain instructions to generate the modified program.

3. The *Clava Weaver Engine* maintains an updated representation of the source code by communicating with the two previous components. It receives the AST from the frontend and applies the strategies from the *LARA engine* to the AST.

What is particularly interesting in this research is the concept of strategies, which are used to perform source code modifications [27]. The ability to modify source code through external specification files is crucial when developing a transpiler capable of translating libraries, since it allows programmers to separate annotations from the source code and future-proof translations.

2.6 Further Mentions

In this section, we briefly mention other noteworthy projects that have been developed in this area: Babel [28] transpiles Smalltalk into CLOS and is focused on automatic translation; and *prometeo* [29] transpiles Python source code to high-performance C source code, using libraries such as BLASFEO[3] to speedup linear algebra calculations.

2.7 Analysis

In this section, we briefly differentiate each of the approaches we have previously discussed and highlight some features that are important in the context of this research. Table 2.1 compares the presented transpilers in four main categories: (1) Intelligibility, which covers the ability for humans to read and understand the generated source code, (2) Coverage, which evaluates the subset supported by each transpiler, (3) Upkeep, which evaluates how well maintained the transpilers are, and (4) Performance, which evaluates the performance difference between the input and the generated source code. The comparison has a rating scheme, where ✓✓✓ is the highest rating and ✓ is the lowest. An invalid category for a given tool is represented with ✗.

[3]Basic Linear Algebra Subroutines for Embedded Optimization (BLASFEO): https://github.com/giaf/blasfeo (Retrieved July 16th 2022)

For this evaluation process, we will not consider Clava, as it is a software refactoring tool. The most relevant functionalities for our project are Clava's processing pipeline and annotation mechanism, which will be discussed in section 3.1.2.

Regarding code intelligibility, JSweet presents examples on its website that are human-readable.[4] The authors of the Fortran-Python transpiler also ensure that code is maintainable [22], although we could not evaluate this with larger examples since they were not available. LinJ goes to great lengths to ensure that the generated code is human-readable and maintainable, even ensuring the appropriate translation of source code comments [21]. PyJL was only able to generate pragmatic source code with very simple examples. During our evaluation, we observed that the code generated by PyJL was not pragmatic, even in simple translation scenarios. For instance, when translating Python's 0-indexed lists to Julia's 1-indexed arrays, the transpiler simply added the literal 1 to each indexing operation, which created redundant operations if the indexing values are literals. Furthermore, this also reduced the readability of the generated code.

Another aspect to consider is coverage. The LinJ, Fortran-Python and JSweet transpilers target a considerable subset of their input language. On the other hand, PyJL covers only a very small subset of Python's constructs and built-in functions, which limits its applicability scenarios.

Furthermore, transpilers also need to be updated to support new versions of their input and output languages. Both Clava[5] and JSweet[6] show recent activity in their official repositories. As Py2Many transpiles to many C-Like programming languages, it should be harder to maintain. Still, it shows recent changes for most of the supported target languages.

When comparing the runtime performance of the generated code, we were not able to find any performance results for the JSweet transpiler, although its intent was never to produce high-performance code but to provide interoperability between two platforms. The source code generated by Linj is, on average, $5 - 6\times$ faster than Common Lisp. This includes the results of the Boyer benchmark, commonly used to compare different Lisp implementations, and a cellular automaton benchmark, where LinJ was only outperformed by C and a manually optimized version of a Java program [21]. The code generated by the Fortran-Python transpiler for the Miranda IO benchmarking application is around $6\times$ faster than Python [22]. When using the DGEMM library, used to perform matrix calculations, the generated code is around $10\times$ faster [22]. Regarding PyJL, we found that the Julia source code produced had errors in many cases, which prevented us from translating benchmarks and provide meaningful performance measurements. The performance of PyJL will be revisited after applying our improvements.

Besides these three main aspects, other individual aspects of each transpiler were considered important in the context of this research. The need for type hints, demonstrated by the Fortran-Python

[4] JSweet Website: https://www.JSweet.org/ (Retrieved July 26th, 2022)
[5] Clava Repository: https://github.com/specs-feup/clava
[6] JSweet repository: https://github.com/cincheo/JSweet (Retrieved September 8th, 2022)

transpiler, might also be necessary for our work, as they largely benefit static analysis mechanisms. Furthermore, as Julia's type system benefits from type annotations [30], using Python's type hints could have a large impact on the performance of the generated code. Additionally, the importance of choosing the right constructs, demonstrated by the Linj transpiler, can have a large impact on the maintainability and pragmatics of the generated source code, an aspect that is crucial for our project. Lastly, having the ability to support external annotations is crucial, as separating annotations from the input source code ensures that they can be used for subsequent translations. This was inspired by the Clava software refactoring tool, which supports external code annotations to generate new versions of the input source code [27]. These aspects have influenced the development of PyJL and have also resulted in changes to Py2Many's architecture, which will be discussed in the following chapter.

3

Transpilation

Contents

3.1 The Py2Many Transpiler 16

3.2 Python to Julia Translation 20

3.3 Optimizations 32

3.4 Validating Translations 33

As discussed in Chapter 1, there are several mismatches that make automatic translation between languages a challenging process. However, as Python and Julia offer similar functionalities, automated translation becomes a feasible approach. This chapter presents our implementation of Py2Many and the PyJL extension, which we developed to translate Python to human-readable and pragmatic Julia source code. Furthermore, it discusses the most notable scenarios encountered during the translation process.

3.1 The Py2Many Transpiler

The Py2Many [31] transpiler transpiles Python to many C-like programming languages. It provides a modular framework, where the support for each language is added as an extension. As an example, there is the PyRS extension for Rust, the PyCPP extension for C++ etc. Most notably for this research, we have the PyJL extension, which adds functionalities to translate Python to Julia.

We chose Py2Many as our starting point, as it offers a flexible architecture and shows recent development. Its architecture was adapted to accommodate essential features required by our solution. Figure 3.1 shows the updated architecture. Py2Many receives Python source code as input and uses Python's `ast`[1] module to parse the source code and build an Abstract Syntax Tree (AST). The AST is modified using intermediate phases that handle language incompatibilities. Lastly, a code generator parses the AST to produce the output source code in the chosen language. A brief description of each transpilation phase follows:

1. *Configuration Rewriters* is a language-independent phase that supports configuration files in JSON and YAML format. These can contain type-annotations and specific decorators, that allow programmers to choose a preferred translation method.

2. *Rewriters* can be language-specific or -independent. They modify or add new nodes to the AST to make it compatible with the target language. As an example, consider a rewriter that translates Python's classes to Julia.

3. *Core Transformers* are language-independent transformers that modify the AST by adding relevant information for the translation process. The added information includes:

   (a) Variable Context: Adds an attribute to each scope, containing a dictionary of locally defined variables.

   (b) Scope Context: Adds a `scopes` attribute to each node in the AST, which includes a list of all its parent scopes.

   (c) Assignment Context: Distinguishes if a node is on the left-hand side of an assignment, i.e., represents the target variable of an assignment.

[1] Abstract Syntax Tree - Python 3.10: https://docs.python.org/3/library/ast.html (Retrieved on July 12th, 2022)

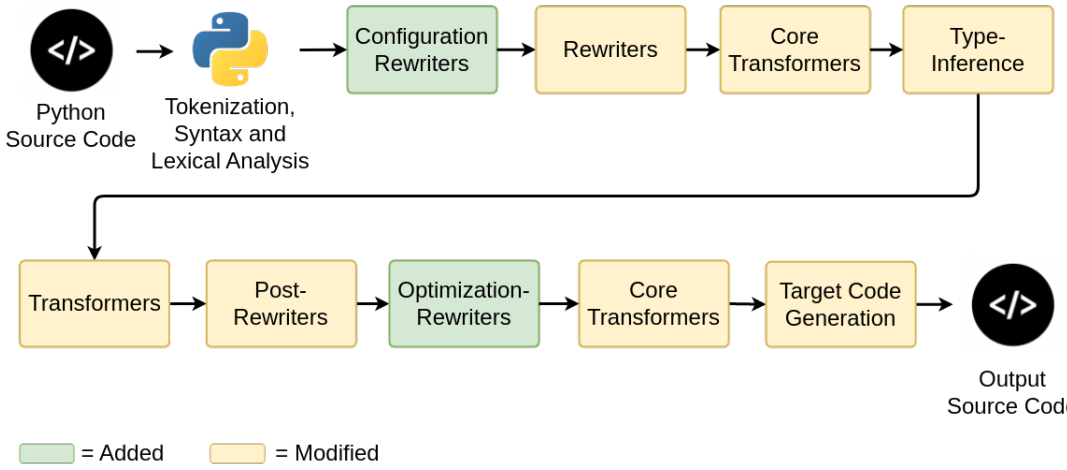


Figure 3.1: Py2Many Architecture

(d) List call information: Adds all list transformation operations to the scope of the variable referencing the list.

(e) Variable Mutability: Adds a list of all mutable variables to function scopes.

(f) Nesting levels: Annotates nodes with the respective nesting levels. This is important for languages sensitive to white space.

(g) Annotation flags: Adds flags to annotations and differentiates type annotations from container types.

(h) Import Parsing: Adds an attribute to each scope, containing a list of local imports.

4. *Type inference* executes Py2Many's type inference mechanism. Transpilers can also extend this mechanism to add more specific inference rules.

5. *Transformers* add complementary information to specific nodes of the AST. As an example, consider performing an analysis step that detects whether using broadcasting is necessary in Julia.

6. *Post Rewriters* are functionally identical to *Rewriters*, but have dependencies on previous intermediary phases, such as the *Inference* phase.

7. *Optimization Rewriters* optimize the syntax and performance of the generated code.

8. *Target Code Generation* is a language-specific phase that translates language syntax and semantics, converting the AST to a string representation in the target language.

In the pipeline, the *Core Transformers* phase executes at two different stages. The first makes necessary information available for the intermediary transformation phases, while the second guarantees that core changes are applied to any newly introduced nodes.

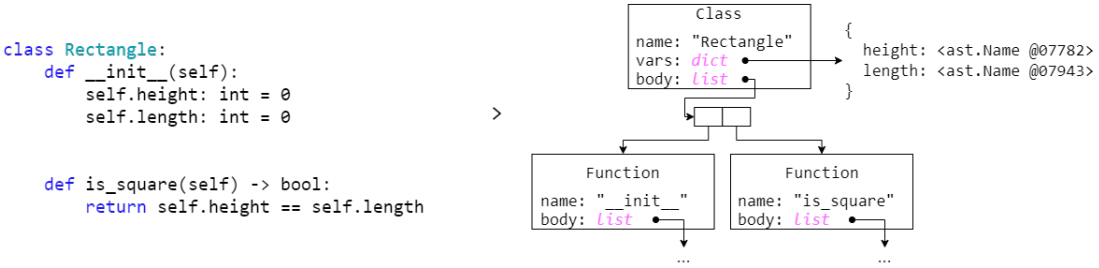


Figure 3.2: Inference tree-walk

Before focusing on the features of PyJL and the translation from Python to Julia, it is important to discuss some noteworthy features included in Py2Many. Some of these were improved or added to Py2Many to complement the translation process. We will briefly describe them in the following sections.

3.1.1 Scoping Mechanism

Py2Many includes some functionalities that aid the translation process. To facilitate tree walks, the *Scope Context* phase adds a `scopes` attribute to each node in the AST, which contains a reference to a node's parent scopes. To cover the scoping rules of all supported languages, Py2Many defines an extended list of constructs that introduce scope blocks, which include the `module`, `function`, `class`, `for`, `while`, `if`, `with`, and `try` blocks. Each of these scope blocks holds a dictionary of all the locally defined variables, which is added by the *Variable Context* phase to speedup variable type lookups. Control flow statements and expressions have a separate dictionary for each execution branch, which ensures that variables with the same name defined in separate branches do not override each other.

Furthermore, Py2Many also includes a `find` function to search for variables by their names. Given the enclosing scope of the current node, it finds the first instance matching that variable's name, by walking the scopes in reverse order. Starting the search from the enclosing scope is an important aspect, as it ensures that Py2Many always finds the appropriate definition of the variable, even when variable shadowing occurs. The `find` function starts by searching the variable dictionaries of each scope, which largely speeds up searches. If the variable is not found in the dictionaries, it searches the body of each scope.

However, Py2Many's mechanism was not complete, as it did not allow class methods to access information about class variables defined in constructors. To demonstrate this use-case, let us consider the example shown in figure 3.2. Notice that the `Rectangle` class defines the `height` and `width` parameters in the `__init__` method. With Py2Many's mechanism, it was impossible to access the information of the `height` and `width` variables by searching the enclosing scopes of the `is_square` method. With our contribution, the current mechanism gathers all class variable assignments in the corresponding class nodes, shown on the right of figure 3.2, which ensures they are accessible by all class methods.

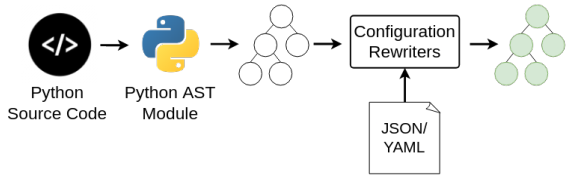


Figure 3.3: Annotation pipeline

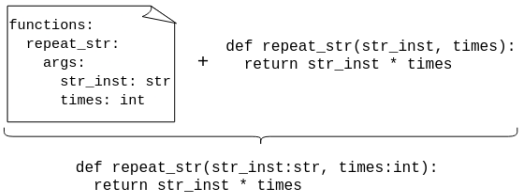


Figure 3.4: Annotation Example

3.1.2 Code Annotation Mechanism

There are scenarios where more than one translation method is supported. Therefore, we added support for three complementary approaches through which programmers can choose their preferred translation method: (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. The first approach offers a simple mechanism for programmers to test the different translation methods. The second and third approaches separate annotations from the input source code, which ensures that annotations can be used for subsequent translations. This is especially useful if the code-base one is trying to translate is updated.

To demonstrate the use of the *Configuration Rewriters* phase, figure 3.3 contains a high-level overview of the processing pipeline. It parses the provided YAML/JSON files and adds the information to the AST. The current supported features are adding decorators to functions and classes, and adding type-hints to function definitions. An example of code annotations can be seen in figure 3.4. In this example, the user supplies a YAML file containing the type annotations that should be considered for the parameters of the `repeat_str` function. These annotations are subsequently merged with the source code before proceeding to the next transpilation phases.

3.1.3 Parsing Mechanism

As previously mentioned, Py2Many uses Python's `ast` module, which syntactically analyses and tokenizes the input source code and generates an AST. However, there are some limitations with the current parsing mechanism. The most notable missing feature is the translation of source code comments, which are not included in the AST. This limitation could be avoided by switching to a Full Syntax Tree (FST). There is a Python library called *baron*,[2] which produces an FST from Python source code. However, at the time of writing, there is no guaranteed future support. Future releases of Py2Many should support the parsing of comments, as this is crucial for code documentation.

Another notable aspect is that Python's `ast` module might not preserve the data representation of the Python source code. This is the case when parsing binary, octal, and hexadecimal literals, which

[2] *baron*: Full Syntax Tree (FST) library for Python: https://github.com/PyCQA/baron

Listing 3.1: 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 3.2: Translation to Julia

```
1  x = [1,2]
2  y = x
3  append!(x, [3,4])
4  splice!(x, 2:2, repeat(x[2:2], 2))
5  splice!(y, 3:2, [1])
```

lose their original representation. As an example, the Python binary literal `0b10110` will be converted to the number `22`. In such cases, PyJL supports custom type aliases, which can be used by programmers to identify literal types. Binary literals can be annotated with `BLiteral`, octal literals with `OLiteral` and hexadecimal literals with `HLiteral`. These type aliases inform PyJL that the values must be converted to their original representation.

After discussing the noteworthy functionalities of the current pipeline and detailing our contributions, we now discuss the translation process of Python to Julia. The following section's will focus on newly implemented features for PyJL and discuss the most notable translation aspects.

3.2 Python to Julia Translation

When translating Python to Julia, a transpiler has to deal with the syntactic and the semantic differences between the two languages. The following sections discuss newly implemented features that focus on translating the most notable Python functionalities to Julia. These were implemented for PyJL, if not specifically noted otherwise.

3.2.1 Augmented Assignments

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

As an example, consider the Python code excerpt and its translation to Julia shown in listings 3.1 and 3.2 respectively. Notice that the Python augmented assignment on line 3 changes when translated to Julia. In Python, the addition operator concatenates the list x with the list `[3,4]`, producing the result `[1,2,3,4]`, while in Julia, it performs an element-wise addition of both arrays, yielding the result `[4,6]`. As augmented assignments performed on mutable objects use in-place operations in Python, PyJL uses Julia's `append!` function, which performs an in-place concatenation of elements to the list `x`.

Translating *slice* operations is another aspect to consider. Lines 4 and 5 of listing 3.2 represent that scenario. PyJL uses the `splice!` function, which replaces or inserts new elements in a list. Notice that the second call to `splice!` uses the form `n:n-1`, which inserts a new element in the list [30, Ch. 42].

Listing 3.3: For-Else Python

```python
def find_factors(n):
  for i in range(2, n):
    for j in range(2, i):
      if i % j == 0:
        print(i, 'equals', j, '*', i/j)
        break
    else:
      print(i, 'is a prime number')
```

Listing 3.4: For-Else Julia Translation

```julia
function find_factors(n)
  for i in 2:n-1
    has_break = false
    for j = 2:i-1
      if (i % j) == 0
        println("$(i) equals $(j) * $(i / j)")
        has_break = true
        break
      end
    end
    if !has_break
      println("$(i) is a prime number")
    end
  end
end
```

3.2.2 Boolean Operations

Most boolean expressions are equivalent in both Python and Julia. However, when used in control flow statements or in the context of boolean operations, some Python values are interpreted as false. These include `False`, `None`, the numeric value zero, empty strings, and empty containers. In contrast, Julia's control flow statements and boolean operations only accept expressions that evaluate to the values `true` or `false`.

To solve this mismatch, PyJL starts by analysing the return type of expressions used in control flow statements and boolean operations. If it detects that the result of such expressions is not a boolean value, it creates a comparison between the corresponding types that Python interprets as false. As, this relies on the inference mechanism to analyse the types of expressions, there might be cases where the types are not available at compile time. Therefore, as a fallback when types are not inferrable, PyJL creates an expression that performs these checks at runtime. Despite not being the most pragmatical solution, it ensures that the result is equivalent on both languages. The only functionality that is currently unsupported is calling Python's `__bool__` method for user-defined objects.

3.2.3 Loops

Python offers two looping constructs, namely `for` and `while`. The most notable aspect is that both of these constructs support an `else` clause, which executes when the loop's condition becomes false. The only exception is if the loop iteration halts through a `break` statement or if an exception is raised.

As none of the target languages in Py2Many offer support for this feature, we implemented a generic translation approach that maps `for-else` and `while-else` expressions. As a possible translation scenario, consider the `find_factors` function from listing 3.3, which outputs a list of factors and prime numbers up to `n`.

The generated Julia source code can be seen in listing 3.4. Notice that a new assignment to a

Listing 3.5: Python Combination Sort

```python
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 3.6: Julia Combination Sort

```julia
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 in 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
```

variable called `has_break` was created, which initially sets it to `false`. It only gets set to `true` when a break statement is executed. After the loop, a conditional expression verifies if the value of this variable is false and only executes the if-statement's block if this condition is verified.

3.2.4 Indexing

In Python and Julia, a subscript is used to perform indexing and slicing on sequences and key lookups on mapping types. The translation of indexing becomes particularly challenging when trying to generate pragmatic code. This section analyses PyJL's indexing translation mechanisms.

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-indexed lists while Julia uses 1-indexed arrays. Indexing can be performed using integer literals or generic expressions. Integer literals can be easily incremented to match Julia's 1-indexed arrays, but non-literal expressions require creating a new binary expression to increment its value, which can reduce the readability of the code generated by PyJL. Nevertheless, because most indexing is performed in loops, we can optimize the entire scenario instead.

For instance, consider the Python implementation of the combination sort algorithm from listing 3.5. The simplest translation, shown in listing 3.6, is to preserve the ranges and adjust the indexing operations. However, this generates redundant binary operations that decrease the code's readability. Therefore, we provide two optimization methods that the programmer can use to improve the quality of the generated code:

1. Determine if loop variables are only used for indexing, and increment loop ranges.

2. Use the *OffsetArrays* package [32] to define custom index ranges for sequences.

Listing 3.7: Julia 1-Indexed Arrays

```
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 in 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 3.8: Julia Offset Arrays

```
function comb_sort(
    seq::Vector{Int})::Vector{Int}
  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 in 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
```

For the first approach, PyJL uses a heuristic to verify if the loops variables are only used for indexing. This ensures that there are no side-effects and that changing the loop's ranges does not change the external behaviour of the generated code. We transpiled the Python combination sort implementation using this translation method. The transpilation result, shown in listing 3.7, respects the pragmatics of Julia and better resembles what a Julia programmer would write.

The second translation method uses *OffsetArrays* to create arrays with the same index ranges as Python. The code generated by PyJL can be seen in listing 3.8. 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. Notice that the array `seq` is modified and returned by the function. Therefore, PyJL used a *let*-block to restrict the wrapping of the input vector `seq` to the block's scope and encapsulate the use of *OffsetArrays* within the function.

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. The second preserves the program's original indexing, but also changes its layout. The applicability of these implementations is, therefore, dependent on each translation scenario, which is why we opted to allow programmers to select their preferred code generation method.

A key factor to consider is that Python allows programmers to use negative indexing to index lists in reverse order. If indexing is performed outside loops, PyJL only supports negative indexing if the indexing value is a literal. However, if indexing is performed within a loop and uses the loop's variables, then this will depend on the ability to statically determine the range of the loop. Currently, PyJL can determine if indexing is performed with negative values when loops use Python's `range` function.

Listing 3.9: Generator Functions

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

Listing 3.10: Julia Channels

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

Listing 3.11: Resumable Functions

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

3.2.5 Generator Functions

Python's generator functions return a lazy iterator and implement 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. In Python, this is implemented using a Finite State Machine. To demonstrate the translation of generator functions, we present an implementation of the Fibonacci sequence that returns an infinite iterator, which can be seen in listing 3.9

To transpile generator functions to Julia, PyJL offers two alternatives. One alternative is to use channels, which also implement the producer/consumer pattern. 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 in listing 3.10. 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.

Another alternative that preserves Python's behaviour is the third-party package `ResumableFunctions` [33]. This package defines a `resumable` macro, which simulates generator functions in Julia. A `yield` macro is used to replace Python's `yield` keyword. Similarly 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 can be seen in listing 3.11. Besides preserving Python's behaviour, this approach also maps more directly to its equivalent Python implementation. Furthermore, it also achieves much higher performance when compared to channels. Section 6.2.2 compares these two alternatives in greater detail. Nonetheless, PyJL offers the possibility to use both methods, as each has their applicability scenarios.

3.2.6 Arbitrary-Precision Arithmetic

Another mismatch occurs with the precision used for arithmetic operations. A programming language can either use arbitrary-precision arithmetic, where the digits of precision are only limited by the available

memory in an operating system, or fixed-precision arithmetic, which only allows up to a fixed amount of digits. The latter has the benefit of providing faster performance, as arithmetic operations have native hardware support.

In Python, arbitrary-precision arithmetic is used on all operations. Julia, on the other hand, always uses fixed-precision arithmetic, but offers specific types to support arbitrary precision floating point numbers and integers. PyJL provides a mechanism that converts all operations to arbitrary precision. It attempts to analyse variable types and wraps any assignments to integers and floating-point values using Julia's `BigInt` and `BigFloat` types. A programmer can supply a flag to choose this translation approach.

As an alternative, PyJL supports the type aliases `BigInt` and `BigFloat`, which allow programmers to manually annotate variables that require the use of arbitrary-precision arithmetic. These variables will be wrapped into calls to Julia's `BigInt` and `BigFloat` types, respectively.

3.2.7 Simulating Python's OO Implementation

Python is an imperative OO language. Julia, on the other hand, is mostly a functional programming language that does not fully support the OO paradigm. If a transpiler from Python to Julia has to translate Python's classes, it must simulate OO mechanisms, such as inheritance. This section discusses PyJL's translation process of Python's OO implementation.

Before discussing our translation approach, one must consider the difference between inheritance, subtyping and composition. Inheritance refers to the reuse of functionality, subtyping models associations between data types by following the concept of substitutability, and composition refers to defining an object as the sum of its parts. Python supports all three concepts through its classes. Furthermore, it also supports multiple inheritance, where one class can extend multiple parent classes. On the other hand, Julia only supports subtyping and composition. Subtyping is supported through Julia's abstract types that can have at most one supertype. A Julia function that is written to work for a supertype will also work for any subtypes, following the concept of substitutability.

Another important difference occurs in the dispatch mechanisms of Python and Julia. Whereas Python uses single dispatch, which selects a method given the type of its first argument, Julia uses multiple dispatch, which selects the appropriate method according to the types of all its arguments. One can use multiple dispatch in Python through the `multimethod`[3] module, but this is not officially supported by the language.

When translating Python's classes, we separated single and multiple inheritance, as the translation approaches to Julia are inherently different. We start by discussing single inheritance and, to that end, we have created a class inheritance example, which can be seen in listing 3.12. It defines three classes,

[3] `multimethod` module: https://pypi.org/project/multimethod/0.5/ (Retrieved July 25th, 2022)

Listing 3.12: Python Class Hierarchy

```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_number: int, domain: str = "school.student.pt"):
        self.name = name
        self.student_number = student_number
        self.domain = domain

    def get_id(self):
        return f"{self.name} - {self.student_number}"

class Worker(Person):
    def __init__(self, name: str, company_name: str, hours_per_week: int):
        self.name = name
        self.company_name = company_name
        self.hours_per_week = hours_per_week
```

namely the `Person`, `Student` and `Worker` classes. Both `Student` and `Worker` are subclasses of the `Person` class, where `Student` adds the `student_num` field and a new definition of the `get_id` method, and `Worker` adds a `company_name` and an `hours_per_week` argument. We considered two different translation approaches for classes that use single inheritance:

1. Using Julia's native constructs to create a class hierarchy.
2. Using a third-party package called *Classes* [34]

In both of these approaches, PyJL must translate Python's `__init__` method, which is called after an object has been created to initialize its fields. To map this method, PyJL uses Julia's constructors, which have a similar purpose. The remaining aspects change depending on the chosen translation approach.

When using the first approach, PyJL generates the code seen in listing 3.13. For each class, it creates a mutable struct to hold the class fields and an abstract type that can be used for subtyping. Notice that the `self` parameter of each function extends the abstract type mapped to the corresponding struct, allowing each function to be used by any subtypes in the hierarchy.

The second approach uses the aforementioned *Classes* package. This package contains a `class` macro, which defines a hierarchy of abstract types and creates a constructor function for each type. The main advantage is that structs annotated with the `class` macro do not have to repeat the fields of its supertypes, which removes duplicate code. In the previous example, we chose the names of the abstract types to match the abstract type names used by the *Classes* package. Listing 3.14 shows an equivalent translation using this package. Notice that PyJL no longer repeats the fields of its parent types when declaring an object. For instance, the `Student` struct only defines the fields `student_number` and `domain`, and omits the `name` field, as it is defined in the parent type.

Listing 3.13: Julia Class Hierachy

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

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

mutable struct Student <: AbstractStudent
  name::String
  student_number::Int
  domain::String

  Student(name::String,
          student_number::Int,
          domain::String = "school.student.pt") =
    begin
      new(name, student_number, domain)
    end
end
function get_id(self::AbstractStudent)
  return "$(self.name) - $(self.student_number)"
end

mutable struct Worker <: AbstractWorker
  name::String
  company_name::String
  hours_per_week::Int
end
```

Listing 3.14: Julia *Classes* Package

```
using Classes

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

@class mutable Student <: Person begin
  student_number::Int
  domain::String
  Student(name::String,
          student_number::Int64,
          domain::String = "school.student.pt") =
        new(name, student_number, domain)
end
function get_id(self::AbstractStudent)
  return "$(self.name) -
    $(self.student_number)"
end

@class mutable Worker <: Person begin
  company_name::String
  hours_per_week::Int
end
```

Despite the *Classes* package offering some advantages to the first method, it still discloses some parts of the underlying Julia implementation. For instance, notice how both `get_id` functions extend the types `AbstractPerson` and `AbstractStudent` to work in a class hierarchy. Still, this implementation hides the creation of the abstract types and structs that hold object fields.

The examples above covered single inheritance. However, if we want to fully map Python's classes to Julia, we also need to handle multiple inheritance. Let us extend our previous example by introducing a new `StudentWorker` class, seen in listing 3.15. With this new class, the hierarchy becomes an instance of the diamond problem, where the class `Person` is the parent of both the `Student` and `Worker` classes, and the class `StudentWorker` is a subclass of `Student` and `Worker`. We include a visual representation of the class hierarchy in figure 3.5.

To support multiple inheritance, PyJL uses a package called *ObjectOriented* that has similar semantics to Python. The code segment in listing 3.16 shows the translation of the `StudentWorker` class when using the *ObjectOriented* package. The remaining code is available in Appendix A. Notice that this translation uses two macros. The `oodef` macro is used to annotate structs and support Python-like features, such as allowing structs to have their own methods, while the `mk` macro is used to initialize

Listing 3.15: Python Multiple Inheritance

```python
class StudentWorker(Student, Worker):
  def __init__(self,
        name: str,
        student_number: int,
        domain: str,
        company_name: str,
        hours_per_week: int,
        schedule_conflicts:bool):
    Student.__init__(self, name,
      student_number, domain)
    Worker.__init__(self, name,
      company_name, hours_per_week)
    self.schedule_conflicts = schedule_conflicts
```

Listing 3.16: Julia Multiple inheritance

```julia
@oodef mutable struct StudentWorker <:
    {Student, Worker}
  schedule_conflicts::Bool

  function new(
    name::String,
    student_number::Int,
    domain::String,
    company_name::String,
    hours_per_week::Int,
    schedule_conflicts::Bool,
  )
    @mk begin
      Student(name, student_number, domain)
      Worker(name, company_name, hours_per_week)
      schedule_conflicts = schedule_conflicts
    end
  end
end
```

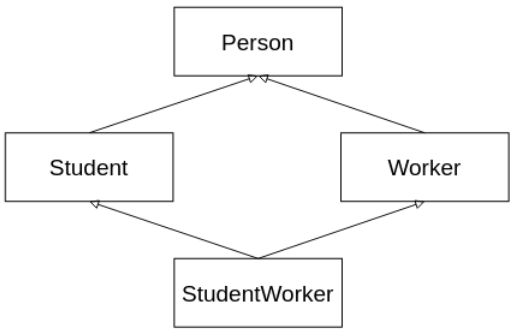


Figure 3.5: Class hierarchy

struct values. Besides creating the class hierarchy, this package also implements the C3 linearization for Method Resolution Order (MRO) [35], which, combined with the ability to overload constructors and methods, and use Python-style properties, offers the current closest mapping of Python's classes to Julia.

An important aspect of this package is that it has two different mechanisms for relating functions to structs: (1) It provides a `like` macro, which performs dispatching based on a structs type, allowing any of its subtypes to use the functions, and (2) allows `oodef` structs to have their own functions similarly to how Python's classes can have their own methods. PyJL currently has support for both translation methods. We will compare these two mechanisms by translating the Person class.

In listing 3.17, we used the first translation method, where the functions are defined outside the body of the struct. The `like` macro allows any subtype of `Person` to extend the `get_id` function. Alternatively, one can use the second code generation method, which can be seen in listing 3.18. Notice how the `get_id` function is now defined inside the struct's body. Using this method no longer requires the `like` macro. As nested functions are considered attributes of the class, they are accessed identically to how

Listing 3.17: ObjectOriented External Functions

```
@oodef mutable struct Person
  name::String

  function new(name::String)
    @mk begin
      name = name
    end
  end
end

function get_id(self::@like(Person))::String
  return self.name
end
```

Listing 3.18: ObjectOriented Nested Functions

```
@oodef mutable struct Person
  name::String

  function new(name::String)
    @mk begin
      name = name
    end
  end

  function get_id(self)::String
    return self.name
  end
end
```

class methods are in Python, that is, akin to accessing object attributes.

3.2.8 Special Methods and Attributes

A Python class can define special methods, which are called implicitly by Python when performing certain operations on objects. These can be extended by any Python class to define custom behaviour, and are identified by names that start and end with double underscores.

Regarding the translation of Python's special methods, we have already discussed the mapping of Python's `__init__` special method, which initializes an object's fields. Furthermore, PyJL also supports Python's `dataclass` special methods. As some methods require objects to be compared for equality, PyJL uses a `__key` function that returns a tuple containing an object's field values. This is used in methods, such as `__eq__` or `__lt__`, to compare struct instances.

Besides Python's special methods, one must also consider Python's special class attributes. Python objects store their attributes in a dictionary, which can dynamically change throughout the execution of the program. Every class defines a `__dict__` attribute, which exposes its attributes. As Julia does not allow structs to change their fields, PyJL extends them with a dictionary, allowing properties to be added at runtime. A heuristic is used to detect this dynamic behaviour. For our example, we will consider the `Person` class that was defined in section 3.2.7. We excluded the `get_id` method, as it was redundant in this scenario. If PyJL detects any calls adding attributes to classes, it generates the code seen in listing 3.19. It adds a new `__dict__` attribute, which is used to add new attributes to the struct at runtime. It also defines a `Base.getproperty` function, which is implicitly called when retrieving a struct's properties. This approach separates any properties added at runtime from properties that are statically available.

3.2.9 Scoping Rules

Another mismatch is related to scoping rules, which define the behaviour of assigning names to values and solve possible conflicts. Both Python and Julia use lexical scoping, which determines, at compile-

Listing 3.19: Dynamic Class Attributes in Julia

```
mutable struct Person
    name::String
    __dict__::Dict{Symbol,Any}
    Person(name::String, __dict__::Dict{Symbol,Any} = Dict{Symbol,Any}()) =
        new(name, __dict__)
end

function Base.getproperty(x::Person, property::Symbol)
    __dict__ = getfield(x, :__dict__)
    if haskey(__dict__, property)
        return __dict__[property]
    end
    return getfield(x, property)
end
```

time, the section in the source code where a name is bound to a value. However, they differ in the scoping rules they apply.

In Python, scopes are defined according to the *LEGB* rule [36, 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 name bindings, 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 [30, Ch. 10]. To explain this concept, we now consider that a variable named `a` is defined in the global scope. If a variable assignment to `a` occurs in a local hard scope, then a new local variable will be created and will shadow the global variable. If a variable assignment to `a` occurs in a local soft scope and all its enclosing scopes are soft scopes, the behaviour changes when used in non-interactive or interactive (REPL) contexts. In non-interactive contexts, the new assignment shadows 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 assigned. Constructs that introduce global scopes include `module`s and `baremodule`s. Local soft scopes are created by `struct`, `for`, `while` and `try` blocks, while local hard scopes are created by `macro`, `function`, `do`, `let`, `comprehension`, and `generator` blocks.

To demonstrate the scoping mismatches between Python and Julia, let us consider an example that uses control flow operators. As previously mentioned, both `for` and `while` constructs introduce new scopes in Julia, but not in Python. Therefore, translating Python's loops to Julia could potentially result in errors if loop target variables are used outside its body. To detect these cases, PyJL analyses the enclosing scope using a heuristic to find any assignments that have the same variable name as any

Listing 3.20: 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 3.21: 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
```

of the loop target variables. If this condition is verified and a `fix_scope_bounds` flag is supplied by a programmer, PyJL creates a new variable in the enclosing scope and updates it in every iteration of the loop. The flag is used to allow programmers to control the code generation method, avoiding unexpected code changes. Alternatively, programmers can supply a `scope_warning` flag that instructs PyJL to emit warning messages when variables defined in Julia's local scopes are used in enclosing scopes. The messages include Python's module names and the corresponding line numbers, which is useful if programmers prefer to manually optimize the code.

An example that shows the scoping mismatches between Python and Julia is the `mandelbrot` function shown in listing 3.20 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. 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 `fix_scope_bounds` flag, PyJL generates the code shown in listing 3.21, which now produces the expected result in Julia.

Another mismatch occurs when using nested constructs, as Julia imposes scoping restrictions. For instance, one of Julia's scoping restrictions is that `struct`s can only be defined in the global scope. One instance where PyJL uses structs is when translating classes from Python. However, while Python's classes can be defined in local scopes, Julia's structs can only be defined in the global scope. Automatically changing the scope of structs could potentially result in name clashes. In addition, this might not match the programmer's intent. Therefore, PyJL supports 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 PyJL to simulate Python's generators in Julia. This macro defines a Finite State Machine to simulate Python's generator functions and creates a `struct` to save its state, restricting its use to the global scope. To account for such cases, we added support for an optional argument field `remove_nested` in the `resumable` decorator, which instructs PyJL to move the resumable function to the global scope.

3.2.10 Keyword Arguments

Another aspect of translation is the mapping of Python's keyword arguments to Julia. Whereas Python allows the use of keyword arguments for all parameters in a function, Julia explicitly requires programmers to distinguish keyword arguments in function definitions. A transpiler mapping Python's functions to Julia must simulate this functionality.

It is important to distinguish two different translation scenarios. The first is regarding keyword-based class constructors. As Python's classes are translated to Julia's structs, which was detailed in section 3.2.7, PyJL can use Julia's *kwdef* macro. This macro analyses which struct parameters have default values and creates a keyword-based constructor for each type. A programmer can annotate a Python class with the `parameterized` macro, which is subsequently translated by PyJL using the `kwdef` macro.

Unfortunately, a limiting factor of *kwdef*, is that it can only be used to annotate structs. Therefore, PyJL provides a `parameterized_func` macro to annotate Python functions. For all annotated functions, PyJL creates separate Julia methods that each support a keyword argument combination. This approach extends the previous functionalities to all translated functions, simulating the behaviour of Python.

3.2.11 Other Incompatibilities

Besides the above-mentioned incompatibilities, several others were translated to Julia. This is the case of Python's context managers. PyJL currently supports Python's `with` statement, which is translated using Julia's `do-block` syntax. Both simplify resource management without requiring a programmer to explicitly close resources. A common use case is file operations, where files are automatically closed when leaving the scope of the opening statement. Alternatively, Python also provides the `contextlib.contextmanager` decorator, which defines factories that create context managers. This was mapped using Julia's `DataTypesBasic` package, which offers similar functionality.

Python's `del` statement, which is used to delete objects, was also mapped to Julia, but only for a small subset of operations. For instance, deleting an element of a dictionary is equivalent to calling the function `delete!` in Julia. However, calling `del` with a variable name in Python will remove the binding between the variable's name and its value, which is not supported in Julia.

3.3 Optimizations

After dealing with the language dissimilarities, we turn to code optimizations. The end goal is to translate language syntax and semantics as programmers would, which is challenging when attempting to perform it automatically. Furthermore, the translated code should achieve similar runtime performance to the original Python source code. In this section, we briefly describe some optimizations performed by PyJL.

Listing 3.22: Python Indexing

```
def newman_conway(n: int):
  f = [0, 1, 1]
  for i in range(3, n + 1):
    r = f[f[i-1]] + \
      f[i-f[i-1]]
    f.append(r)
  return r
```

Listing 3.23: Julia Indexing

```
function newman_conway(n::Int)
  f = [0, 1, 1]
  for i in 3:n+1-1
    r = f[f[i-1+1]+1] +
      f[i-f[i-1+1]+1]
    push!(f, r)
  end
  return r
end
```

Listing 3.24: Julia Optimized

```
function newman_conway(n::Int)
  f = [0, 1, 1]
  for i in 3:n
    r = f[f[i]+1] +
      f[i-f[i]+1]
    push!(f, r)
  end
  return r
end
```

3.3.1 Removing Redundant Operations

The first important aspect is to remove any redundant code that is produced by the `Rewriters` and `Post Rewriters` phases. In listing 3.22, we have an implementation of the Newman-Conway sequence written in Python. If we attempt to translate this implementation to Julia without any code optimizations, we get the result shown in listing 3.23. Notice that simply adding the literal 1 to every indexing operation would result in the generation of redundant operations. PyJL's *Optimization Rewriters* phase removes the redundant code generated by the intermediate phases to optimize code generation. This optimization can be observed in listing 3.24, which produces code that is much closer to what a Julia programmer would write.

3.3.2 Optimizing Global Variables

Global variables in Julia have significant overheads. As their values and corresponding types may change, Julia cannot optimize memory allocations. Code that relies heavily on global variables is almost guaranteed to run slower. One workaround is to pass global variables as function arguments, which results in considerable speedups due to Julia's function-level optimizations. However, this is not a solution for a transpiler, as adding new arguments to functions can be regarded as an unwanted behaviour.

As an alternative, PyJL uses Julia's `const` keyword when global variables hold constant values. To choose this translation approach, the programmer can supply a `use_global_constants` flag. PyJL also uses a heuristic to verify if the variable is not redefined throughout the program and will only apply the optimization if this condition is verified.

3.4 Validating Translations

Validating the generated code is crucial to determine that its external behaviour is identical to that of the input source code. Unit testing is commonly used to verify the correctness of source code and to guarantee that it has the expected behaviour. Testing provides a method to assert relative correctness, where the correctness of software is tested relative to its specification. Therefore, testing identifies

the most valuable behaviours set by programmers, providing a method to test corner cases and detect possible errors. This is a crucial aspect for unit test translation, as translated tests provide a method to assert that the behaviour of the code has not changed after the translation process.

Fortunately, the majority of Python's tests use assertion methods that check for errors and report any test failures. Identifying and translating unit-tests is therefore simple and allows for the verification of translated code. Furthermore, some Python test frameworks also allow parameterizing test functions, which executes tests with different input values. It would be beneficial to support this approach, as it promotes test reuse. This section discusses our translation approaches.

3.4.1 The *unittest* Framework

The *unittest* [37] framework is the most commonly used framework for unit testing in Python. It follows an OO approach and supports 4 concepts for test creation [37]:

1. *Test fixtures* perform the necessary set-up and clean-up of resources and execute before and after running one or more unit tests, respectively.
2. *Test cases* are individual units of testing, which are represented as Python classes that extend the `unittest.TestCase` base class.
3. *Test suites* aggregate multiple test cases or test suites.
4. *Test runners* manage the execution of unit-tests and output the final test results.

Besides the above terms, Python also differentiates test modules from the remaining Python modules. Modules that contain unit tests are commonly called test scripts and can contain multiple test cases.

As Py2Many translates Python to many C-like programming languages and does not currently support the translation of unit-tests, it would be beneficial to create a generic translation mechanism that supports all the output languages. Our generic translation approach translates a subset of the *unittest* framework and is compatible with all the supported languages, further contributing to Py2Many's development. The only aspect that is language-dependent is the mapping of Python's assertion functions, such as `assertIs` or `assertEqual`, which have to be translated into corresponding functions or checks for each target language. In the case of Julia, PyJL uses the `test` macro from Julia's `Test` module, which tests if a given expression evaluates to `true`.

To demonstrate our translation approach, we chose an excerpt from the test script `test_augassign`, which is from CPython's test suite. Listings 3.25 and 3.26 show the Python and the generated Julia source code, respectively. The `AugAssignTest` test case contains several test methods, which use the prefix `test`. This prefix is used by the `test runner` to identify test methods. In this case, we have the

Listing 3.25: Python Unittest Excerpt

```python
class AugAssignTest(unittest.TestCase):
  def testSequences(self):
    x = [1, 2]
    x += [3, 4]
    x *= 2
    self.assertEqual(x, [1, 2, 3, 4, 1, 2, 3, 4])
    x = [1, 2, 3]
    y = x
    x[1:2] *= 2
    y[1:2] += [1]
    self.assertEqual(x, [1, 2, 1, 2, 3])
    self.assertTrue(x is y)

if __name__ == "__main__":
  unittest.main()
```

Listing 3.26: Julia Unittest Excerpt

```julia
abstract type AbstractAugAssignTest end
mutable struct AugAssignTest <:
  AbstractAugAssignTest end

function testSequences(self::AbstractAugAssignTest)
    x = [1, 2]
    append!(x, [3, 4])
    append!(x, repeat(x, 1))
    @test (x == [1, 2, 3, 4, 1, 2, 3, 4])
    x = [1, 2, 3]
    y = x
    splice!(x, 2:2, repeat(x[2:2], 2))
    splice!(y, 3:2, [1])
    @test (x == [1, 2, 1, 2, 3])
    @test x === y
end

if abspath(PROGRAM_FILE) == @__FILE__
  aug_assign_test = AugAssignTest()
  testSequences(aug_assign_test)
end
```

`test_sequences` method, which tests augmented assignment operations with container types. Notice that the Python module calls `unittest.main`, which runs the tests contained in the current test script. Py2Many replaces this call by creating a new instance of each test case, represented as Julia structs, and calling its respective test methods, simulating the use of a *test runner*.

Furthermore, our translation approach also supports the translation of test case fixtures, which set-up test cases and clean-up the resources when these finish. In Python, these are represented by the methods `setUp` and `tearDown`, which are implicitly called by the *unittest* module prior and after running all the test methods, respectively. In such cases, Py2Many will explicitly call the functions before and after running all the tests to ensure that resources are properly handled.

3.4.2 Parameterized Unit Tests

As an additional functionality, we also support parameterized unit tests, as these largely benefit the re-use of test methods. Python has support for this functionality thought the `pytest` testing framework. This contribution was made specifically for PyJL and is not a generic translation approach, as was the case of the *unittest* module. The main reason for this decision, is that Julia has the *ParameterTests* package that offers a much more pragmatic solution to the translation of parameterized tests.

To demonstrate the translation process, we selected a simple test case, which can be seen in listing 3.27. Listing 3.28 shows the code generated by PyJL. Notice that the `test_palindrome_detector` function was replaced for a new parameterized test created using the `paramtest` macro. Internally, it uses Julia's `testset` macro to group a set of tests and output a test summary showing all the successful and failed tests. The test statistics are created by analysing the results of the individual unit tests. The

Listing 3.27: Python Parameterized Test

```
1  import pytest
2
3  def palindrome_detector(s: str):
4      s = s.lower().replace(' ', '')
5      return s == s[::-1]
6
7  @pytest.mark.parametrize("input,expected", [
8      ("madam", True),
9      ("false", False)])
10 def test_palindrome_detector(input, expected):
11     assert palindrome_detector(input) == expected
```

Listing 3.28: Julia Parameterized Test

```
1  using ParameterTests
2  using Test
3
4  function palindrome_detector(s::String)::Bool
5      s = replace(lowercase(s), " " => "")
6      return s == s[end:-1:begin]
7  end
8
9  @paramtest "test_palindrome_detector" begin
10     @given (input, expected) ∈ [
11         ("madam", true),
12         ("false", false)
13     ]
14     @test(palindrome_detector(input) == expected)
15 end
```

assertion on line 11 in listing 3.27 was also replaced by a call to Julia's `test` macro (line 14 of listing 3.28). Whereas Python's assertions are used for test statistics, this is not the case in Julia, requiring the use of the `test` macro. This ensures that the behaviour is similar to Python.

4

Type Inference

Contents

4.1 Py2Many's Inference Mechanism 38
4.2 External Type Inference Mechanism 41
4.3 Why two Mechanisms? 43
4.4 Alternative Solutions 44

Dynamically typed programming languages, such as Python, have become increasingly popular in the past years, as they are easy to use and allow for fast development. However, with that ease of use, several problems emerged, such as the difficulty to perform early error checking or to use static analysis techniques [23].

One particular case of static analysis is static type inference, which relies on static information to infer types at compile time. Static type inference mechanisms can infer the types of expressions, but only with sufficient static constraints. For instance, by analysing the expression `x = 2`, a static type inference mechanism can infer that `x` has the type `int`. However, as Python is a dynamically typed language, inferring the type of most expressions is difficult, as these do not have predetermined types. This is opposed to Hindley-Milner languages, such as Haskell, where the types of expressions can be deduced with little or no annotations. As an example, consider the Python expression `x = y + z`. In this scenario, it is impossible to statically infer the type of `x` if the types of `y` and `z` are unknown, as Python uses overloading and determines the appropriate operation depending on the runtime types of operands. Furthermore, expressions requiring dynamic evaluation, such as Python's `eval`, are, in general, not type-inferrable.

As an attempt to solve these problems, Python introduced PEP484 [38], which added optional type hints. Despite the importance of type hints for static analysis, many Python code bases do not use them [39]. Given the complexity and size of modern code bases, requiring programmers to manually add type hints before transpilation could take an enormous amount of time. As such, it would be beneficial to have a type inference mechanism that requires little programmer input.

Py2Many already offers a basic type inference mechanism for Python. However, this mechanism did not provide sufficiently precise types, even after our extensive revisions and corrections. Therefore, we integrated an external type inference mechanism to increase the available type information at transpilation time and reduce the time required to manually annotate the Python source code. This chapter describes the current solution and discusses its limitations.

4.1 Py2Many's Inference Mechanism

The inference mechanism in Py2Many is implemented using a definition-use chain. This mechanism recursively walks the AST and aggregates type information from node assignments for each scope. It works with the mechanism defined in section 3.1.1, by using the variable dictionaries to search for any variable types.

We extended Py2Many's inference rules to cover a broader subset of Python. The new inference rules cover `for` and `while` nodes, generator expressions, and support type propagation for different execution branches. Furthermore, we added inference support for list and dictionary comprehensions,

as well as binary operations. The latter required extensive changes, most notably, differentiating the left and right operands to support Python's operator overloading.

As Python modules typically have many dependencies on external modules or libraries, it is important to determine the return types of imported functions. The following section discusses this topic.

4.1.1 Import Analysis

Type inference in Python is particularly challenging, given Python's heavy dependence on external APIs. Most static type inference mechanisms rely on analysing a program's data flow, starting from values for which types are known at compile time. The lack of type information for imported functions can result in an incomplete data-flow analysis. Furthermore, it is critical to analyse the data-flow dependencies between Python modules.

Therefore, we consider two different import scenarios:

1. Importing from Python's standard library or from distributed packages listed on PyPI [9].

2. Importing local Python modules.

To solve the first scenario, we mapped a subset Python's built-in functions to their corresponding return types. It is important to note that the return type of some functions is dependent on the argument types. This is the case of the built-in `max` and `min` functions, which return the maximum and the minimum value of iterables, respectively. In such cases, the inference mechanism analyses the argument types to determine the corresponding return type of the function.

Regarding imports from local modules, Py2Many merges the type information from imported functions with the importing module. As an example, if Py2Many detects that a module `A` imports a function from module `B`, then it searches module `B` for that function and adds an `annotation` attribute that matches the one in module `A`. We completed this mechanism by adding new rules for Python's `import-from` statement. This method also requires all the modules to be sorted according to their import dependencies, as the type information from imported modules has to be available to any modules importing them. This will be discussed in more detail in section 5.2.1.

4.1.2 Static Type-Checking

We extended Py2Many's inference mechanism to create static type constraints from the provided type hints. The current mechanism creates bindings between variables and their provided type hints. Any future assignments within the variable's scope will be checked for incompatible value types. To illustrate this mechanism, consider the following assignment operations in Python:

Listing 4.1: Bonacci Series Python

```python
def bonacciseries(n: int, m: int):
    a = [0] * m
    a[n - 1] = 1
    for i in range(n, m):
        for j in range(i - n, i):
            a[i] = a[i] + a[j]
    return a
```

Listing 4.2: Bonacci Series Julia

```julia
function bonacciseries(n::Int, m::Int)::Vector
    a = fill(0, m)
    a[n] = 1
    for i in n:m-1
        for j = i-n:i-1
            a[i+1] = a[i+1] + a[j+1]
        end
    end
    return a
end
```

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

In this case, the mechanism rejects the second assignment to variable `l`, as the type of the value being assigned to it does not match its previous type annotation. Alternatively, consider a scenario where the value of the second assignment is another variable. In this case, the inference mechanism attempts to search for the variable's type. If the type does not match the previous annotation, the inference mechanism rejects the assignment.

4.1.3 Limitations

A crucial aspect when translating Python to Julia, is that the translation of operators is influenced by the types of their operands. To further demonstrate this scenario, let us consider the Python code excerpt on the left and its corresponding translation to Julia on the right, which both concatenate two lists:

```
a = [1,2]
b = [3,4]
ab = a + b
```

```
a = [1,2]
b = [3,4]
ab = [a;b]
```

Notice that the last statement is different in both languages. While the addition of lists in Python corresponds to their concatenation, in Julia it corresponds to the element-wise addition of list elements. The translation of the addition operator is therefore dependent on the types of the variables `a` and `b`.

The fact that operator translations are dependent on the operand types implies that the transpiler must be able to infer the types of operands. This was possible in the example above, as one can statically infer the types of the variables `a` and `b`. However, this is not the case for all scenarios. Let us consider the Python Bonacci Series implementation from listing 4.1. Notice how the first assignment to variable `a` uses the multiplication operation between a list and an integer. In Python, this creates a new list by repeating the elements of the original list `m` times. Without any type annotations, the inference mechanism is not able to determine the type of variable `m`. However, if the function is annotated, the

transpiler can translate this operation as seen in listing 4.2, which uses Julia's `fill` function to create an array of size `m` with every location set to the repeated element.

Due to Python's dynamic behaviour, Py2Many requires programmers to annotate function definitions and can only generate correct source code if the appropriate type-hints are provided. This is a downside of the current mechanism. Depending on the size of the code base one is trying to translate, the required time to perform these annotations can quickly escalate, making transpilation less appealing. Therefore, it would be beneficial to integrate an external type inference mechanism to increase the available type information at transpilation time.

4.2 External Type Inference Mechanism

To complement Py2Many's type inference mechanism and reduce the time required to annotate the Python source code, we integrated an external type inference mechanism. Py2Many provides the option of using the TYPPETE [40] type inference mechanism, which is based on the Z3 theorem prover [41] that uses a MaxSMT solver to solve type constraints. However, the most recent version only supports up to Python 3.6, which limits any future releases of Py2Many.

We also found an inference mechanism that uses probabilistic type inference [42]. This extracts naming conventions from a code base and generates a set of constraints from variable names, attribute accesses, and the data flow of the program. It subsequently uses the generated constraints to create a probabilistic inference network and resolves the network to get probabilities of individual types for each variable. Despite offering a promising solution, the implementation is only a prototype, which offers no guarantees on future support.

The most promising static type inference tools were pyright [43] and pytype [44]. The first was developed by Microsoft, whereas the latter was developed by Google. In contrast to the MyPy [45] static type checker, which uses a gradual typing approach, both of these inference tools provide a lenient approach to type inference, allowing operations that do not contradict annotations. As an example, consider the following Python expressions:

```
l = ["1"]
l.append(2)
```

Whereas the second line would result in an error when using MyPy, both pyright and pytype allow the expression, as it is correct at runtime.

A benefit of both tools is that they can export the inferred type information to Python stub files, separating them from the Python source code. Programmers can also supply external stub files containing annotations, which largely speeds up the inference process. Furthermore, both pyright and pytype use

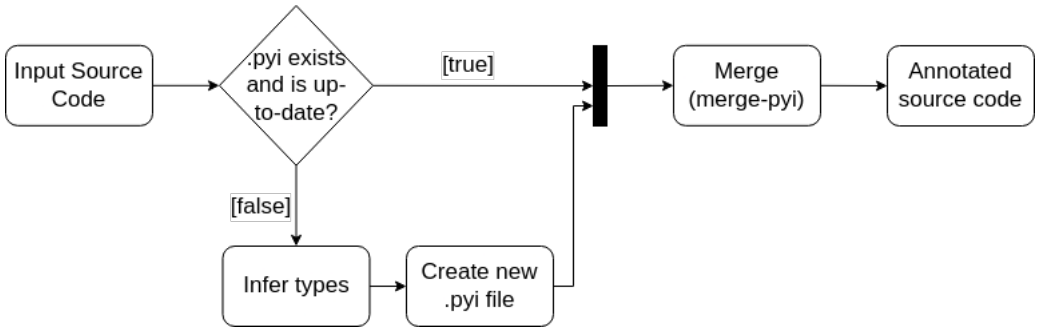


Figure 4.1: Pytype inference

typeshed,[1] which provides library stubs for Python containing type annotations. These cover Python's standard library and a small subset of Python's registered libraries listed on PyPI [9]. This resolves the problem of incomplete data-flow analysis by providing a large data-set of annotated functions.

We decided to use pytype, as it supports useful tools for our project. This is the case of the `merge-py` tool that merges the annotations in the stub files with the original source code. Since pytype's annotations are exported as stub files, they only include annotations for functions and global variables. Nonetheless, these annotations contribute to minimize the work required to manually annotate function definitions.

The mechanism created to integrate pytype's annotations can be seen in figure 4.1. The first time a Python module is transpiled, the transpiler calls pytype, which creates a stub file containing the inferred type declarations. Subsequently, the transpiler uses pytype's `merge-py` tool that merges the stub file with the Python module and creates an annotated version of the module, which will be used for the remaining translation process. Separately, a *log* file is created, which contains the full path of each transpiled Python module, along with a hash of its contents. In subsequent translations, the transpiler checks the log file and compares the saved hash to the newly generated hash of the module being transpiled. If the hashes differ, the transpiler calls pytype's inference mechanism to re-generate the stub files, and updates the log file with the new hash. Otherwise, the existing stub file is used.

4.2.1 Advantages

The integration of the pytype type-inference mechanism improves the type information available at compile time. In particular, pytype's control-flow analysis allows us to identify optional return types. As an example, consider the function `read_file_contents` that retrieves the contents of a file given its path. The function first verifies if the given path references a file. If it does, it returns the contents of the file. Otherwise, it returns the value `None`. As the return type is the union of the types `str` and `None`, pytype

[1] typeshed: Collection of library stubs for Python, with static types: https://github.com/python/typeshed (Retrieved August 3rd, 2022)

Listing 4.3: Optional Return Types

```
def read_file_contents(path: str)
    -> Optional[str]:
  if not isfile(path):
    return None
  res = []
  with open(path, "r") as file:
    res.append(file.readline())
  return "\n".join(res)

def parse_file(path: str):
  if file_contents := read_file_contents(path):
    # Parse file contents
  else:
    raise Exception("The file was not found")
```

Listing 4.4: Generic Type Annotations

```
_T0 = TypeVar('_T0')

def bubble_sort(seq: _T0) -> _T0:
  l = len(seq)
  for _ in range(l):
    for n in range(1, l):
      if seq[n] < seq[n - 1]:
        seq[n - 1], seq[n] =
          seq[n], seq[n - 1]
  return seq
```

annotates it as `Optional[str]`.

Using the added information from pytype, Py2Many's inference mechanism propagates the appropriate variable types for each execution branch. Listing 4.3 includes an example that shows this scenario. The function `parse_file` contains a conditional expression that verifies the presence of the file contents. Remember from section 3.2.2 that the value `None` is interpreted as false in the context of boolean expressions. Therefore, the information provided by pytype allows Py2Many's inference mechanism to propagate the type of the variable `file_contents` as a `string` for the first branch.

4.2.2 Limitations

However, there are limitations to what a static inference mechanism can do. Although pytype does not explicitly require type annotations, we found cases where the static type information is not enough to produce precise annotations.

As an example, consider the implementation of bubble sort from listing 4.4, which was annotated by pytype. The function `bubble_sort` receives a list `seq` and sorts it in-place. As pytype has no information on the list's type, it creates an alias for an unbound type `_T0`, which can be anything. If the transpilation requires more precise types, these must be judiciously added by programmers to the Python source code.

4.3 Why two Mechanisms?

After introducing the pytype type inference mechanism, it seems redundant to also use Py2Many's inference mechanism. However, pytype exports the inferred type annotations in the form of Python stub files, which only contain the annotations for function definitions and global variables. Py2Many propagates these annotations throughout the remaining nodes in the AST. Most notably, Py2Many adds

an `annotation` attribute to each AST node, containing the inferred type information. The type information added by Py2Many is crucial for static analysis and translating Python's dynamic features.

There is a pytype tool called `annotate-ast`, which could avoid this two-part solution and allow us to solely use pytype. This tool parses the Python source code and creates an annotated version of Python's AST. Annotations are added to the AST's nodes using a `resolved_annotation` attribute, which contains the inferred type information. However, the development of this tool is still in-progress and the support is very preliminary. The integration of such a tool would benefit Py2Many. To avoid breaking the current implementation, one could create a wrapper around the returned annotated AST, changing the `resolved_annotation` attribute to the current `annotation` attribute. We plan to integrate this tool in future releases.

4.4 Alternative Solutions

Since Python is a dynamically typed language, one could generate code in Julia that determines, at runtime, the correct operations to apply. For instance, we could create a new `py_add` function in Julia to map Python's addition operator, which would determine the appropriate operation to apply depending on the runtime types of its arguments. One could create several Julia methods for each supported type combination, which would benefit from Julia's multiple dispatch. Although this is a valid approach, using it extensively would generate convoluted code and negatively affect readability, pragmatics and performance. Despite being a tradeoff, requiring function annotations not only ensures that the generated source code is correct, but also allows the generation of pragmatic source code.

5

Dependencies

Contents

5.1 Python and Julia's Import Mechanisms 46
5.2 Importing Local Modules 47
5.3 Importing Registered Modules 49
5.4 Name Aliases 53
5.5 Accessing Dynamic-Link Libraries 53

A predominant aspect when translating Python to Julia is proper handling of module dependencies. Python provides a sophisticated import machinery to allow a module to access functionalities defined in external modules. This must be analysed in two different ways. The first is performing a local search for a module that belongs to the same library or package, where PyJL must ensure that translated modules can interact with each other in the same way as Python modules do. The second is importing a module from Python's standard library or a registered package. In such cases, PyJL maximizes the use of existing Julia packages when these offer the same functionalities as Python libraries. If no equivalent Julia packages are available, the generated Julia source code must use the FFI to access existing Python libraries.

As an additional functionality, we also considered the translation of Python modules that access DLLs. We chose to translate a subset of Python's `ctypes` [46] module. To evaluate the chosen translation method, we translated a Python library that uses ctypes to access shared libraries.

Figure 5.1 demonstrates the four scenarios we intend to support in PyJL. The first scenario will be discussed in section 5.2, the second and third scenarios will be discussed in section 5.3, and the fourth scenario will be discussed in section 5.5.

5.1 Python and Julia's Import Mechanisms

Before discussing the mapping of Python's imports to Julia, it is important to distinguish both import mechanisms. Despite Python and Julia offering mechanisms that search and import functionalities from external source files, they differ substantially.

An important difference is how both languages define modules and packages. Whereas Python calls each source file a module, which has its own namespace, Julia's modules are created separately from files, where each module can spread over several files and each file may contain several modules. Similarly to Python, each Julia module has its own namespace. Regarding packages, these are defined in Python as modules that contain other submodules or sub-packages [47]. Furthermore, Python's packages are divided into two categories. Regular packages contain a Python module called `__init__.py` that is executed implicitly when importing a regular package. A package without the `__init__.py` file is called a namespace package, which can be split across multiple directories. PyJL currently supports Python's regular packages. On the other hand, in Julia, packages are defined as modules that wrap a collection of submodules, i.e., they are modules that import submodules and make them accessible as a single unit. This differs from Python, as Python's packages can have a similar structure to a file system, where each package can be represented as a folder.

To access the contents of other modules or packages in Python, one can use the `import` keyword or the `import-from` statement. Both search for a module or package name and bind that result to a name

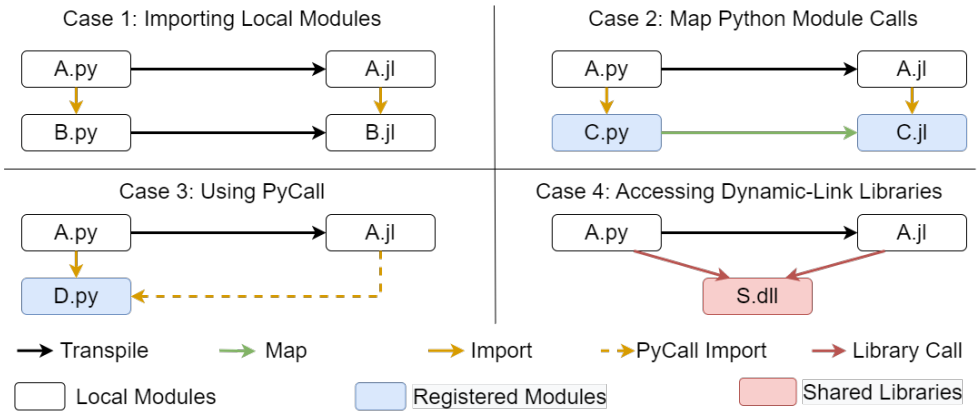


Figure 5.1: Dependency Scenarios

in the local scope of the importing module [48, Ch 5]. On the other hand, Julia offers two mechanisms to import code from external source files. One can use inclusion through the `include` keyword, which evaluates the contents of a source file in the global scope of the module importing it [30, Ch 16]. The second method is used for loading packages and offers two different syntaxes [30, Ch 16]:

1. The `import` keyword operates on a single name at a time. It does not allow one to perform module searches, but allows extending functions with new methods, if necessary.

2. The `using` keyword can operate on multiple modules, but does not allow extending imported functions.

The main difference between code inclusion and package loading, is that the latter mechanism is used to search packages in project environments listed in Julia's `LOAD_PATH`. A project environment is created using a project file and an optional manifest file. The project file contains project-specific information, such as its name, identifier, and external package dependencies. This is used to build a roots map, which maps dependency names to their unique identifiers. The manifest file is used to build a dependency graph of the project. Both ensure that import statements can find the correct module or package, avoiding potential name clashes. Furthermore, project environments define a specific folder structure, which differs from Python's package structure.

After describing the differences of both mechanisms, we now focus on mapping Python's imports to Julia. The following sections discuss our translation approach.

5.2 Importing Local Modules

To access local modules, we considered both package loading and code inclusion. Package loading would require creating a project environment to automatically add Julia source file paths to the

Listing 5.1: Python Import

```python
# module1.py
from module2 import fib
fib(10)
```

```python
# module2.py
def fib(i: int):
    if i == 0 or i == 1:
        return 1
    return fib(i - 1) + fib(i - 2)
```

Listing 5.2: Julia Import

```julia
module module1
using FromFile: @from
@from "module2.jl" using module2: fib
fib(10)
end
```

```julia
module module2
function fib(i::Int)::Int
    if i == 0 || i == 1
        return 1
    end
    return fib(i - 1) + fib(i - 2)
end
end
```

`LOAD_PATH`. Furthermore, Python's packages are not equivalent to Julia's projects. On the other hand, code inclusion introduces code duplication, as inclusions re-evaluate the contents of source files in the scope of the importing file. Therefore, we used the *FromFile* package, which evaluates files in total isolation. Any time a file is loaded, a new binding is created in the current context. Upon loading a file that has already been loaded in the current context, a binding to the previously loaded file is returned, which avoids code duplication. Listings 5.1 and 5.2 show an example using Python's import statement and its translation to Julia using the *FromFile* package, respectively.

As Python modules define their own namespace, PyJL wraps the contents of each Julia source file using the `module` keyword, which avoids any potential name clashes. However, as creating modules for each source file is not mandatory, the transpiler offers a flag that removes this encapsulation. As a safeguard, it includes a heuristic that detects potential module name clashes, in which case it uses Julia's modules independently of the flag's value. This is particularly useful if the intent is to transpile individual modules with no dependencies between them.

PyJL uses the base package's file system directory to trace the paths to all local modules. Modules are imported using a relative path from the importing to the imported module. An important aspect is that Python's imports commonly use fully qualified names to identify the imported modules, which include any parent packages. In such cases, one can transpile the Python base package and all its subpackages, which allows PyJL to trace the module paths from the root package's file system directory. Alternatively, if the goal is to transpile a subset of a package, PyJL offers an `import_basedir` flag to manually set the base directory.

5.2.1 Module Dependencies

When translating multiple modules, one must also account for the dependencies between them, as these can affect the translation outcome. When importing variables or functions from external modules,

a transpiler should be capable of propagating the associated type information. As this is a language-independent functionality, it is implemented for all transpilers in Py2Many.

As Python's import system does not allow circular imports, we can use a sorting algorithm to sort all the modules before analysing and transpiling them. There is already an implementation in Py2Many that extracts the imports from modules and uses a topological sort algorithm to sort them according to their import dependencies. Any modules that have dependencies on other modules will be analysed last, allowing type information to be propagated across modules.

A noteworthy aspect is that this mechanism was incomplete. Most notably, Py2Many was not able to analyse subpackage dependencies. Furthermore, package-relative imports were not supported, which resulted in the lack of type propagation throughout modules. Solving this problem required extending the mechanism to support Python's `import-from` statements. Additionally, Py2Many did not propagate types for `__init__.py` modules, which are implicitly executed when importing packages. By fixing these incompatibilities, we completed the mechanism.

5.3 Importing Registered Modules

After analysing the dependencies between local modules, we now focus on imports to Python's built-in library or to distributed packages listed on PyPI [9]. We considered two alternative approaches, which are represented as cases 2 and 3 of figure 5.1, respectively:

1. Translating Python's module and library calls into calls to Julia packages that offer identical functionalities.

2. Using Julia's *PyCall* [15] package to call Python libraries.

PyJL uses the first approach whenever Julia offers libraries that have the same external behaviour as Python libraries. This is an important aspect to reduce any dependencies from Python and benefit from Julia's native functionalities. Nonetheless, some Python functionalities might need to be simulated to achieve a similar behaviour in Julia. The following section discusses this aspect in greater detail.

5.3.1 Simulating Package Calls

An FFI allows a program written in one language to access functions from foreign languages, providing a mechanism to reuse existing functionalities. A transpiler translating Python to Julia could use this mechanism to call Python's existing libraries. However, this approach has several drawbacks. A notable one is handling GC incompatibilities, which can lead to unpredictable program behaviour. Furthermore, handling complex types tends to be rather cumbersome and usually requires explicit type conversions

Listing 5.3: NumPy Array Representation

```
1  a1 = np.array([1,2,3,4])
2  a2 = np.array([1,2,3,4])
3  np.dot(a1, a2)
4
5  a3 = np.array([[1,2], [3,4]])
6  a4 = np.array([[1,2], [3,4]])
7  np.dot(a3, a4)
8
9  a3 * a4
```

Listing 5.4: Translation to Julia

```
1  a1 = [1,2,3,4]
2  a2 = [1,2,3,4]
3  a ⋅ b
4
5  a3 = [1 2;3 4]
6  a4 = [1 2;3 4]
7  a3 * a4
8
9  a3 .* a4
```

in the source code, which can reduce its readability. To minimize these problems, PyJL attempts to use existing Julia packages to simulate Python's library calls.

The best-case scenario is when Julia already offers equivalent packages for Python's libraries or modules. This is the case of Python's `json` module, which can be mapped to Julia's `JSON` package. An equally good scenario occurs with Python library calls that can be translated to Julia to achieve a similar behaviour as Python. This is the case with Python's linear algebra or mathematical libraries, for which Julia offers equivalent functionalities. To test the feasibility of such translations, we chose to translate a subset of the high-performance numeric library NumPy.

An important aspect when translating NumPy calls is to consider the internal representation of data. Despite NumPy's arrays being stored contiguously in memory, some NumPy functions create views over the returned arrays, which affects indexing. Furthermore, one has to account for the fact that some NumPy functions have a different behavior depending on their argument types. To demonstrate these two mismatches, let us consider a simple example in NumPy from listing 5.3 and its corresponding translation to Julia, which can be seen in listing 5.4. When using one-dimensional arrays, NumPy's `dot` function is translated to the dot ($\cdot$) function in Julia. However, when used with multidimensional arrays, NumPy's `dot` function performs matrix multiplication, which must be translated using Julia's multiplication operation. A more interesting scenario occurs when using the standard multiplication operator with two matrices. To preserve the semantics, this operation is translated to Julia using broadcasting, which applies operations element-wise on array-like objects.

To further demonstrate the mapping of NumPy calls to Julia, we translated an implementation of the Sieve of Eratosthenes shown in listing 5.5. The generated Julia source code can be seen in listing 5.6. The first NumPy calls are trivial to translate to Julia. NumPy's `ones` and `sqrt` functions get translated to Julia's `trues` and `sqrt` functions. The vectorized assignment on line 6, seen in listing 5.5, must use broadcasting in Julia. Similarly to Python, it sets all the elements in the slice to false. Lastly, the function `flatnonzero` is translated by PyJL into a list comprehension in Julia, that uses the function `enumerate` to iterate through the array. Notice that PyJL decreases the return values by 1 to match Python's 0-indexed lists. We also validated the pragmatics of the generated code through user tests. The suggestions allowed us to make several improvements to our NumPy translations. The results will be discussed in

Listing 5.5: Sieve NumPy

```
def sieve(n):
    primes = np.ones(n, dtype=bool)
    primes[0], primes[1] = False, False
    for i in range(2, int(np.sqrt(n) + 1)):
        if primes[i]:
            primes[i*i::i] = False
    return np.flatnonzero(primes)
```

Listing 5.6: Sieve Julia Translation

```
function sieve(n)
    primes = trues(n)
    (primes[1], primes[2]) = (false, false)
    for i in 2:Int(floor(sqrt(n) + 1))-1
        if primes[i+1]
            primes[i*i+1:i:end] .= false
        end
    end
    return [i-1 for (i,p) in
        enumerate({vargs[0]}) if p != 0]
end
```

Listing 5.7: Python Module Import

```
import pyproj
x, y = pyproj.transform(
  pyproj.Proj(4326), pyproj.Proj(3857),
  45.0, 45.0
)
```

Listing 5.8: Julia PyCall Import

```
using PyCall
pyproj = pyimport("pyproj")
(x, y) = pyproj.transform(
  pyproj.Proj(4326), pyproj.Proj(3857),
  45.0, 45.0
)
```

section 6.4. The performance of the sieve implementation will be tested in section 6.2.4.

5.3.2 Using *PyCall*

As a fallback when equivalent Python modules or packages are not available in Julia, PyJL uses Julia's *PyCall* package, which provides a wrapper around Julia's FFI to call Python functions. A noteworthy aspect of *PyCall* is that some type conversions are performed automatically, such as the ones for numeric or boolean values. The remaining types are wrapped in a generic `PyObject` type that represents a reference to a Python object.

To demonstrate the translation process when using *PyCall*, we present an example shown in listing 5.7, which uses Python's `pyproj` library for coordinate transformations and cartographic projections. The corresponding translation to Julia can be seen in listing 5.8. The `pyproj.transform` function transforms points between two coordinate systems, represented as projections. In this case, it translates latitude and longitude coordinates to the Web Mercator projection, commonly used by web-based mapping tools. The `pyproj` module is imported through *PyCall*. Notice that the syntax used to call the `pyproj.transform` function is identical in Python and in Julia. This is achieved through a `Base.getproperty` function that *PyCall* defines for all `PyObject` wrappers, which retrieves the properties of Python's objects.

When function calls return `PyObject`s, PyJL must adapt the generated code to support them. As an example, we now consider the function `get_proj_from_parameter` from the `python-reprojector` library, which creates projections used for cartographic transformations. The Python code for this function and

Listing 5.9: Python reprojector excerpt

```python
from pyproj import Proj

class InvalidFormatError(Error):
  message: str

def get_proj_from_parameter(param):
  msg = "Invalid projection definition"
  if isinstance(param, Proj):
    proj = param
  elif isinstance(param, str) and
      param.lower().startswith('epsg:'):
    proj = Proj(projparams=int(param[5:]))
  else:
    try:
      proj = Proj(projparams=param)
    except RuntimeError:
      raise InvalidFormatError(msg)
  return proj
```

Listing 5.10: Julia reprojector translation

```julia
using PyCall
pyproj = pyimport("pyproj")

mutable struct InvalidFormatError <: Exception
  message::String
end

function get_proj_from_parameter(param)
  msg = "Invalid projection definition"
  if pybuiltin(:isinstance)(param, pyproj.Proj)
    proj = param
  elseif isa(param, String) &&
      startswith(lowercase(param), "epsg:")
    proj = pyproj.Proj(
      projparams = parse(Int, param[6:end])
    )
  else
    try
      proj = pyproj.Proj(projparams = param)
    catch exn
      if exn isa PyCall.PyError &&
            pybuiltin(:issubclass)(
              exn.T, py"RuntimeError"
            )
          throw(InvalidFormatError(msg))
      end
    end
  end
  return proj
end
```

the corresponding Julia code generated by PyJL can be seen in listings 5.9 and 5.10, respectively. The class `InvalidFormatError` is a custom Exception created in the `python-reprojector` library and is included for completeness.

Notice that the function starts by verifying if `param` is an instance of `pyproj.Proj`. As `pyproj` was imported using *PyCall*, the result of calling `pyproj.Proj` will be a `PyObject`. Therefore, PyJL uses *PyCall* to call Python's built-in `isinstance` function, which verifies the instance of the returned object. Furthermore, PyJL also supports the handling of FFI exceptions. As *PyCall* wraps exceptions in a `PyError` object, PyJL must unwrap the returned exceptions and check if it matches the handled exception. This can be seen in listing 5.10 on lines 21 to 23. Notice that PyJL calls Python's built-in `issubclass` function to verify if the returned exception is a subclass of the expected exception, which simulates the behaviour of Python's exception handling.

A noteworthy aspect of this approach is that it is a temporary solution. Using *PyCall* creates dependencies to Python's libraries, making the generated code volatile if libraries are frequently changing. Furthermore, as shown in listing 5.10, the code is more verbose than an equivalent native solution, which reduces its overall readability. As more packages become available in Julia, these should be used instead of the Python equivalents. This would not only minimize the dependencies to Python, but also improve the pragmatics of the generated code.

5.4 Name Aliases

Both Python and Julia support aliases, which can be used to change the name of imported objects or functions in the namespace of the module importing them. This can be used when importing external modules and also when importing local modules using the `FromFile` package.

As an example, the statement `import json as js` is translated by PyJL to `import JSON as js`. Additionally, consider changing the import from listing 5.1 to `from module2 import fib as fibonacci`, which would get translated by PyJL to `@from "module2.jl" using module2: fib as fibonacci`.

5.5 Accessing Dynamic-Link Libraries

After translating the dependencies between modules, it is also important to consider the dependencies to DLLs. This involves mapping Python's FFI calls to equivalent Julia calls. For the purpose of this research, we will focus on Python FFI's that interact with the C programming language. In this regard, the two most notable FFI interfaces are the `cffi`, which allows one to use native C syntax to call foreign libraries, and `ctypes`, which exposes an API to call Foreign Library functions. One notable aspect of the `cffi` interface, is that it can parse native C declarations and automatically infer the necessary data types. Unfortunately, Julia does not currently provide a similar mechanism. We therefore focused on the translation of `ctypes`.

The `ctypes` FFI includes special data types to interface with C. These are used to specify the argument and return types of function calls to external libraries. Furthermore, `ctypes` also supports several calling conventions. On Linux, one can use the `cdll` module to load external libraries, which uses the `cdecl` calling convention. On Windows, this is done through the `windll` module, which uses the `stdcall` calling convention.

We can translate this functionality by mapping Python's `ctypes` FFI calls to Julia. To load shared libraries, Julia uses the `Libdl.dlopen` function, which returns a handle to the loaded library. To call functions in foreign libraries, Julia provides the `ccall` function, which allows using both the `cdecl` and `stdcall` calling conventions through an optional argument. Similarly to Python, Julia also offers special types to interface with C, which are used to specify the argument and return types of foreign functions.

5.5.1 Translation Methodology

To demonstrate PyJL's translation methodology, we chose an implementation of the Levenshtein distance metric written in C.[1] We will start by analysing the Python program seen in listing 5.11, which uses

[1] Implementation by Guillaume Androz: https://gist.github.com/gandroz/19b39b7240aec08bd92c7a06f2174107#file-levenshtein_tab-c

Listing 5.11: Levenshtein Module Python

```python
import ctypes, numpy as np

def load_module():
    cmodule = ctypes.cdll.LoadLibrary(
        "./levenshtein.so"
    )
    cmodule.levenshtein.argtypes = [
        ctypes.c_char_p,
        ctypes.c_char_p,
        ctypes.c_int,
        ctypes.c_int,
        ctypes.c_int,
    ]
    cmodule.levenshtein.restype = ctypes.c_int
    return cmodule

if __name__ == "__main__":
    cmodule = load_module()
    res = cmodule.levenshtein(
        "levenshtein".encode("utf-8"),
        "levenstein".encode("utf-8"),
        np.int32(1),
        np.int32(1),
        np.int32(2)
    )
    print(f"Levenshtein distance between" +
      f"'levenshtein' and 'levenstein': " +
      f"{res}"
    )
```

Listing 5.12: Levenshtein Module Julia

```julia
using Libdl
using StringEncodings

function load_module()
    cmodule = Libdl.dlopen("./levenshtein.so")
    return cmodule
end

if abspath(PROGRAM_FILE) == @__FILE__
    cmodule = load_module()
    res = ccall(
        Libdl.dlsym(cmodule, :levenshtein),
        Cint,
        (Ptr{Cchar}, Ptr{Cchar}, Cint, Cint, Cint),
        encode("levenshtein", "utf-8"),
        encode("levenstein", "utf-8"),
        Int32(1),
        Int32(1),
        Int32(2),
    )
    println("Levenshtein distance between" *
        "\'levenshtein\' and \'levenstein\': " *
        "$(res)")
end
```

`ctypes` to import the `levenshtein` function. The function `load_module` loads the shared library and sets the function's field types and its return type using the `argtypes` and `restype` attributes, respectively. When calling `cmodule.levenshtein` on lines 19-25, one must only supply the arguments, as their types were specified earlier.

The code generated by PyJL can be seen in listing 5.12. Notice that the assignments from function `load_module` that were used to set the argument and return types have been removed, as Julia follows a different approach to FFI calls. The type information is saved in PyJL using a dictionary, which maps each library and function name combination to its corresponding types. Upon transpiling the call to the `levenshtein` function, the information is retrieved from the dictionary and subsequently converted to the equivalent types in Julia, which can be seen on lines 11-20 in listing 5.12. This translation method was used, as the arguments for Julia's `ccall` function must be literal values. To create a callable function pointer for the `levenshtein` function, PyJL used the `Libdl.dlsym` function, which looks up the `levenshtein` symbol in the shared library handle returned by `Libdl.dlopen`.

5.5.2 Additional Functionalities

We demonstrated our translation approach using a simple scenario. Real-world scenarios are much harder to translate due to the inherent differences between Python and Julia's FFI implementations. The

Listing 5.13: Function Factory Python

```python
def function_factory(
    function, argument_types=None,
    return_type=None, error_checking=None):
  if argument_types is not None:
    function.argtypes = argument_types
  function.restype = return_type
  if error_checking is not None:
    function.errcheck = error_checking
  return function

_GetTickCount = function_factory(
  WinDLL("kernel32").GetTickCount,
  None,
  DWORD)
```

Listing 5.14: Function Factory Julia

```julia
_GetTickCount =
  (a0) -> ccall(
    Libdl.dlsym(dlls.kernel32, :GetTickCount),
    Culong,
    (Nothing,),
    a0)
```

most notable limitation we found was related to Julia's `ccall` function, as the argument and return types must be literals. Despite Python allowing the use of expressions to define the types of foreign function calls, there are scenarios that PyJL can identify and convert to Julia. During the translation of the `pywin32-ctypes` library, further detailed in section 6.3.2, we found a use-case where a factory function was used to set the necessary function arguments for FFI calls. Listing 5.13 shows the implementation of this function in Python. Furthermore, we also present an example, where `function_factory` is used to call the `GetTickCount` function from the `Win32` API, which returns the number of milliseconds since the start of the system.

The Python `function_factory` function sets the argument types, the return type, and, if provided, assigns a callable function for error checking. To translate this code excerpt to Julia, PyJL has to identify that the function being called is a factory function and replaces all calls to it. Listing 5.14 shows the equivalent translation to Julia. As can be seen, PyJL creates a new lambda expression for `_GetTickCount` that uses Julia's `ccall` function. As the `function_factory` definition is no longer necessary, PyJL removes it, saving only the information that it is a factory function. A limitation of this approach is that the types must be statically available when calling the factory function.

Besides mapping dynamic library calls, PyJL also supports the translation of callback functions. Currently, PyJL supports the translation of Python's `CFUNCTYPE` and `WINFUNCTYPE` functions, which create new C function pointers for Python functions that use the `cdecl` or the `stdcall` calling conventions, respectively. These functions are replaced into calls to Julia's `cfunction` macro, which generates a C-compatible function pointer for a Julia function. A noteworthy aspect is that the arguments of `cfunction` are evaluated in the global scope. Therefore, if the first argument corresponds to the name of a nested Julia function, it must be prefixed with `$`, which informs Julia to create a runtime closure over the function. This is done by wrapping the function using a `CFunction` struct.

Additionally, one must also consider how Python and Julia represent pointers. Whereas Python represents pointers as hexadecimal values, Julia uses the `Ptr` type to wrap pointer addresses. Therefore, if

pointers are used in any arithmetic operations or passed to functions expecting an integer, PyJL unwraps the `Ptr` object to extract the memory address.

Despite the previously mentioned functionalities, there are still limitations to the current mechanism. A broader evaluation can be found in section 6.3.2, where we discuss the obtained test coverage when translating the `pywin32-ctypes` library.

6

Evaluation

Contents

6.1 Evaluating Translation Correctness 58
6.2 Performance 58
6.3 Library Translations 64
6.4 Evaluating Code Pragmatics 65
6.5 Automatic vs Manual Translation 67
6.6 Extensibility 68

In this chapter, we evaluate PyJL and discuss its limitations. We will focus on the most notable translation scenarios. The remaining results are available in our repository [49] to allow for further comparisons and analysis.

Section 6.1 discusses the methods used to evaluate translations. This is followed by an analysis of the runtime performance of the generated code, comparing it to native Python and optimized implementations that use NumPy. As PyJL's main goal is to translate libraries, section 6.3 evaluates the translation of two libraries and tests the capabilities of PyJL with larger code bases. In section 6.4, we discuss the results of our online survey, which evaluates the pragmatics of the generated code with experienced Julia programmers. Lastly, section 6.6 discusses extensibility, as having an extensible and modular architecture allows for easier modifications and is critical for future developments.

6.1 Evaluating Translation Correctness

When translating Python to Julia, it is important to guarantee that the generated code has the same external behaviour on both languages. To independently validate each supported functionality, Py2Many already included a unit-test suite, which we extended to cover a broader subset of Python. Py2Many's test suite comprises 64 test scripts. Each test has an average of 25 LOC and only covers basic functionalities. PyJL currently covers 83% of the entire test suite. The remaining 17% cover currently unsupported functionalities, such as Python's `asyncio` or `NamedTempFile` modules.

As the included test suite was quite scarce, we selected a subset of CPython's official test suite, comprised of 62 test scripts. CPython's tests are more complete, with an average of 733 LOC. Currently, we only support 10% of the chosen test scripts, as each test covers many Python functionalities. As an example, CPython's `test_augassign` test script required mapping Python's special functions, such as `__add__`, `__radd__`, `__sub__` etc. Some modules also include dependencies on currently unsupported modules, such as `functools` or `subprocess`, which are harder to translate to Julia. Additionally, CPython's tests mostly test corner cases, which are only rarely found in library code.

The validation of library translations is performed using the provided unit tests. We discuss the translation of two libraries and the obtained test coverage in section 6.3.

6.2 Performance

To test the runtime performance of the generated code, we chose a set of Python benchmarks that have an implementation in Julia. As these implementations were manually optimized by Julia programmers, we refer to them as reference versions. In some benchmarks, there is a large difference between the performance of the generated Julia code and the Julia reference versions. This difference is related

Listing 6.1: Combinations Function Python

```
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
```

Listing 6.2: Combinations Function Julia

```
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
```

to the optimizations applied to the reference versions, which allow the Julia compiler to produce more efficient machine code. These versions therefore represent the best-case performance scenario on Julia.

We measured the results using the `bencher` benchmarking tool [50] on a machine with an Intel(R) Core(TM) i7 4790K @4.4GHz with 16GB of RAM running Linux. As some benchmarks are quite sizeable, we made them available in appendix B. This section discusses the obtained results.

6.2.1 N-Body-Problem

The first benchmark we present is an implementation of the N-Body problem that predicts the gravitational interactions of Jovian planets in the solar system. The performance results shown in figure 6.1 reveal that the initial translation is slower than Python and is orders of magnitude slower than the reference Julia implementation.

After analysing the generated source code, we discovered that the slowdown was caused by insufficient type information. The Python function that caused the slowdown and the code generated by PyJL can be seen in listings 6.1 and 6.2, respectively. Both listings show an implementation of a function called `combinations`, which receives a list as its argument and generates combinations with the list's elements. It then adds those combinations to a new `result` list, which is returned by the function. The main performance issue, is that the returned list was translated by PyJL to a generic vector in Julia, which has considerable overheads as Julia cannot infer its element types.

After specifying the necessary type information, we obtained a speedup of $17.5\times$, making the translated Julia code $7.5\times$ faster than the original Python code. This can be achieved in one of two ways, as demonstrated below:

One can use the approach on the left and annotate the `result` array in Python with its corresponding type. Alternatively, one can manually annotate the array in Julia using a more pragmatic approach, as seen on the right. Both alternatives are identical in terms of performance.

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

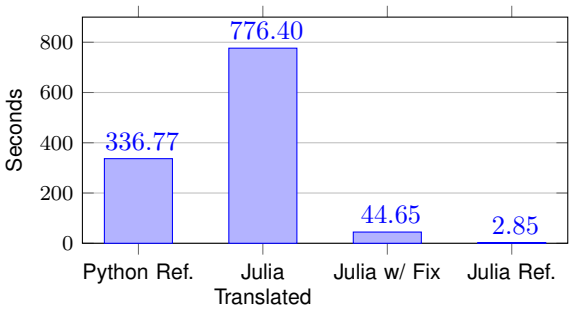


Figure 6.1: N-Body Benchmark

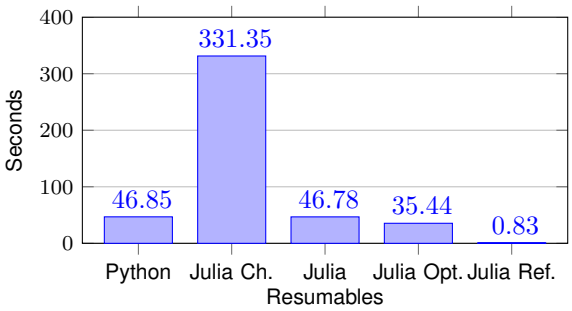


Figure 6.2: Fasta Benchmark

Listing 6.3: Python Cumulative Probabilities

```python
def makeCumulative(
    table::list[tuple[str, float]]
):
    P: list[float] = []
    C: list[str] = []
    prob = 0.
    for char, p in table:
        prob += p
        P += [prob]
        C += [char]
    return (P, C)
```

Listing 6.4: Julia Cumulative Probabilities

```julia
function makeCumulative(
    table::Vector{Tuple{String, Float64}}
)
    P::Vector{Float64} = []
    C::Vector{String} = []
    prob = 0.0
    for (char, p) in table
        prob += p
        push!(P, prob)
        push!(C, char)
    end
    return (P, C)
end
```

```
result: list[
  tuple[
    tuple[list[float], list[float], float],
    tuple[list[float], list[float], float],
  ]
] = []
```

```
result = Tuple{eltype(l), eltype(l)}[]
```

6.2.2 Fasta

To test the runtime performance of generator functions, we chose an implementation of the Fasta benchmark, which generates Random DNA sequences by using a Linear Congruential Generator (LCG). As discussed in section 3.2.5, PyJL offers two methods to translate Python's generator functions. The first uses Julia's Channels and the second uses a package called *Resumables*. Figure 6.2 shows the results of translating this implementation of Fasta. Note that the performance of Channels is almost $7\times$ slower when compared to Python. When using *Resumables*, the performance results match the Python implementation.

Similarly to the N-Body problem, we also found a case where Python's generic lists were translated to generic vectors in Julia. This occurred in the `makeCumulative` function, which 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

Listing 6.5: Python Sieve

```python
def sieve(n: int):
    primes = [True] * (n)
    primes[0], primes[1] = False, False
    for i in range(2, int(math.sqrt(n) + 1)):
        if primes[i]:
            for j in range(i * i, n, i):
                primes[j] = False
    return list(filter(lambda j: primes[j], range(2, n)))
```

Listing 6.6: Julia Sieve

```julia
function sieve(n::Int)
  primes = repeat([true], n)
  (primes[1], primes[2]) = (false, false)
  for i in 2:Int(floor(sqrt(n) + 1))-1
    if primes[i+1]
      for j in i*i:i:n-1
        primes[j+1] = false
      end
    end
  end
  return collect(
    filter((j) -> primes[j+1],
    2:n-1))
end
```

Listing 6.7: Julia Offset Arrays

```julia
function sieve(n::Int)
  primes = OffsetArray(repeat([true], n), -1)
  primes[0], primes[1] = (false, false)
  for i in 2:Int(floor(sqrt(n) + 1))-1
    if primes[i]
      for j in i*i:i:n-1
        primes[j] = false
      end
    end
  end
  return collect(
    filter((j) -> primes[j],
    2:n-1))
end
```

to store the nucleotides. Listing 6.3 shows the annotated `makeCumulative` function. The annotations on lines 2 and 3 for lists `P` and `C` solve the initial slowdown. The Julia source code produced by PyJL can be seen in listing 6.4. As one can see from figure 6.2, the annotations resulted in a speedup of $1.3\times$, resulting in a total execution time of 35 seconds.

Julia's reference implementation is still faster, although it uses several optimizations to achieve that performance. Most notably, it stores the nucleotides as a Vector of `UInt8` values, which largely improves indexing performance when compared to strings.

6.2.3 Sieve

To test the performance of *OffsetArrays*, discussed in section 3.2.4, we used an implementation of the Sieve of Eratosthenes. Listing 6.5 shows the Python implementation and listing 6.6 shows the Julia code generated by PyJL. Notice that PyJL did not apply any indexing optimizations, instead adding the literal 1 to every indexing operation. This is because it detected that the variable `i` is used to calculate the range of the inner loop, which prevents the optimization from being applied. The *OffsetArrays* implementation can be seen in listing 6.7, where PyJL wrapped the assignment to variable primes, creating a 0-indexed vector. The indexing operations were also adjusted accordingly.

Figure 6.3 shows the obtained results. When using the approach seen on listing 6.6, we observed that the runtime performance was $13.5\times$ faster than the Python implementation. When using *OffsetAr-*

Listing 6.8: Sigmoid Functions Python

```python
def sigmoid(z: np.ndarray) -> np.ndarray:
    return 1.0 / (1.0 + np.exp(-z))

def sigmoid_prime(z: np.ndarray) -> np.ndarray:
    return sigmoid(z) * (1 - sigmoid(z))
```

Listing 6.9: Sigmoid Functions Julia

```julia
function sigmoid(z::Matrix)::Matrix
    return 1.0 ./ (1.0 .+ e .^ -z)
end

function sigmoid_prime(z::Matrix)::Matrix
    return sigmoid(z) .* (1 .- sigmoid(z))
end
```

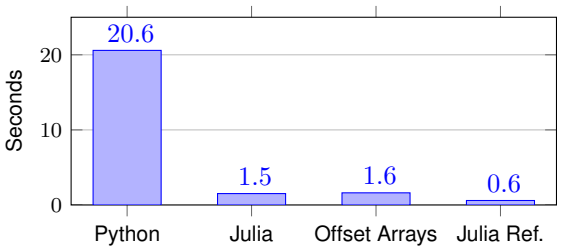


Figure 6.3: Sieve Translation

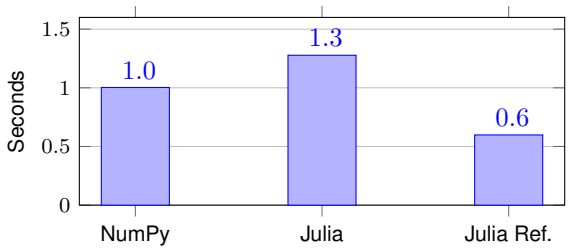


Figure 6.4: Sieve NumPy Translation

rays, the results only differ by 6%, which shows that the overhead is minimal.

6.2.4 Sieve Numpy

To compare the performance difference of using NumPy to native Julia, we used a modified implementation of the Sieve of Eratosthenes from section 5.3.1. The results can be seen in figure 6.4. We included the result of Julia's reference version from figure 6.3 for comparison.

As one can see, the NumPy version of sieve is around $20\times$ faster than the native Python implementation shown in section 6.2.3. NumPy is faster than Julia, but only by $1.3\times$. Julia's reference version is still the fastest implementation, managing a speedup of $2.2\times$ and $1.7\times$ when compared to the translated and NumPy versions respectively.

6.2.5 Neural Network

To further test the supported NumPy subset, we chose a publicly available implementation [51, Ch 1] of a neural network capable of identifying numbers from a MNIST dataset. The data consists of images of handwritten digits and is represented as NumPy n-dimensional arrays. This section discusses the most notable translation aspects and compares the performance of the generated code.

As explained in section 5.3.1, vectorized operations have to be translated using Julia's broadcasting operator. Listing 6.8 shows the Python implementation of the `sigmoid` and `sigmoid_prime` functions, and listing 6.9 shows the code generated by PyJL. Notice that the `np.exp` function receives a NumPy multidimensional array as input. As such, it is translated by broadcasting the power operation with the Euler constant. As the result of this operation is a NumPy multidimensional array, PyJL propagates

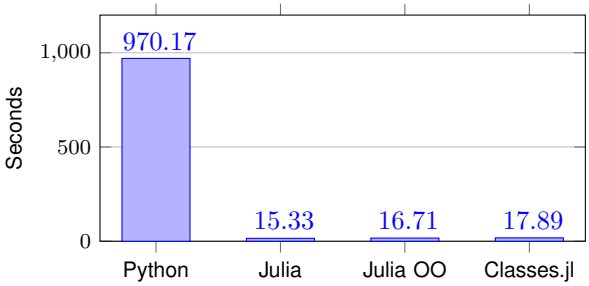


Figure 6.5: Binary Trees

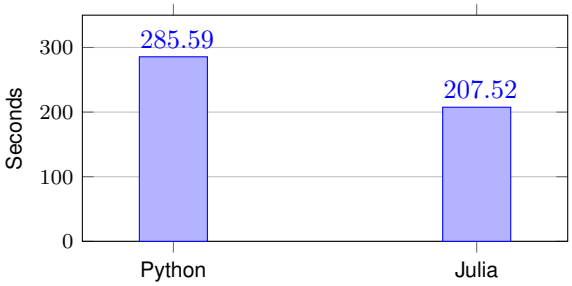


Figure 6.6: Neural Networks

the broadcasting to the outer binary operations. PyJL also supports broadcasting with function calls, as demonstrated by the `sigmoid_prime` function. One notable requirement is that functions must be annotated with the appropriate return types.

In terms of performance, figure 6.6 shows the runtime performance results obtained from running the neural network with 50000 training entries over 30 epochs. As one can see, the performance in Julia is about $1.4\times$ faster when compared to Python.

6.2.6 Binary Trees

To evaluate the performance difference of the supported class translation methods presented in section 3.2.7, we chose a benchmark that allocates and traverses short-lived trees. We modified a publicly available Python benchmark[1] by replacing the tuple data structure with a class implementation. Furthermore, we removed the use of Python's `multiprocessing` module, as it is not supported by PyJL. This new implementation was then translated using the various class translation approaches.

Figure 6.5 compares the different implementations. As can be observed, Python takes around 970 seconds to execute this benchmark, which is an order of magnitude slower than the Julia implementations. Regarding the Julia class implementations, one can notice that the performance difference is minor across all three versions. The `Classes` package appears to have a slightly larger overhead when compared to the other packages.

It is worth noting that PyJL currently translates Python's classes using mutable structs, as Python's classes are considered mutable. However, as tree nodes are not modified throughout the program's execution, we could have used immutable structs. We found that this improved performance by $1.6-2\times$ on average. We plan to support this feature in the future.

[1] Binary-Trees: https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-python3-5.html

6.3 Library Translations

PyJL aims to translate Python libraries to Julia, allowing Julia developers to benefit from Python's extensive library set. In this regard, it is important to test its ability to translate larger code bases. Therefore, we chose to translate two libraries to validate PyJL's capabilities. The first is the *python-reprojector* library, which performs coordinate transformations between projections. As some coordination transformation libraries are currently unavailable in Julia, we translated this library using the *PyCall* package. The second is the *pywin32-ctypes* library, which is an implementation of the pywin32 library using the `ctypes` FFI. This library was chosen to test the limitations of our translation mechanism, allowing us to identify possible future mismatches. This section analyses the translation process and discusses the current limitations.

6.3.1 The *python-reprojector* Library

The *python-reprojector* library [52] performs coordinate transformations between projections by using Python's *pyproj*[2] library. It also uses Python's *shapely*[3] library to manipulate geometric coordinates. Unfortunately, Julia does not yet have support for these libraries. Therefore, they had to be mapped using *PyCall*.

After mapping the unsupported library calls, the remaining code was translated to Julia. This approach translates the main application logic to Julia, allowing for an incremental removal of Python dependencies. As new Julia libraries emerge, programmers are only required to change call mappings to use these libraries. Furthermore, one can use PyJL to translate future library releases, which largely speeds up library updates.

During our evaluation of the library, we managed a 100% test coverage when converting the included unit-tests to Julia. We also confirmed that the external behavior of the library was identical on both languages.

6.3.2 The *pywin32-ctypes* Library

To further validate our ctypes translation, we decided to transpile the *pywin32-ctypes* [53] library. This allowed us to compare the differences between Python and Julia's FFIs and identify potential future translation problems. The main goal of this translation, was to use Julia's native features without requiring the use of *PyCall*. The most notable functionalities added to the transpiler while translating this library were discussed in section 5.5.2. This section focuses on the results obtained with our translation approach.

[2]Python pyproj library: `https://github.com/pyproj4/pyproj`
[3]Python shapely library: `https://github.com/shapely/shapely`

Figure 6.7: Python Experience of Participants

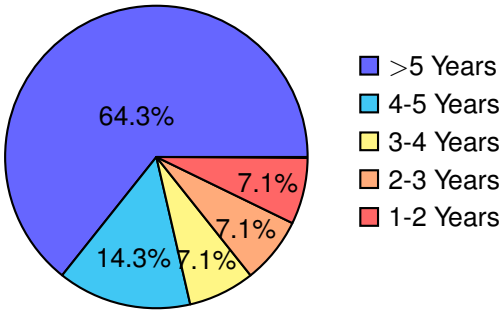


Figure 6.8: Julia Experience of Participants

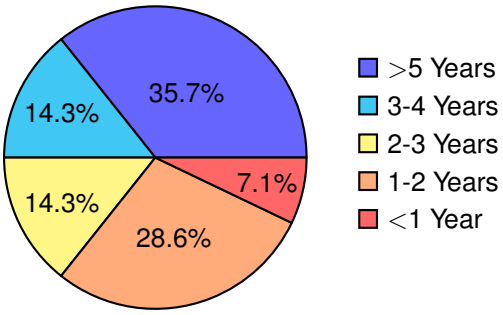
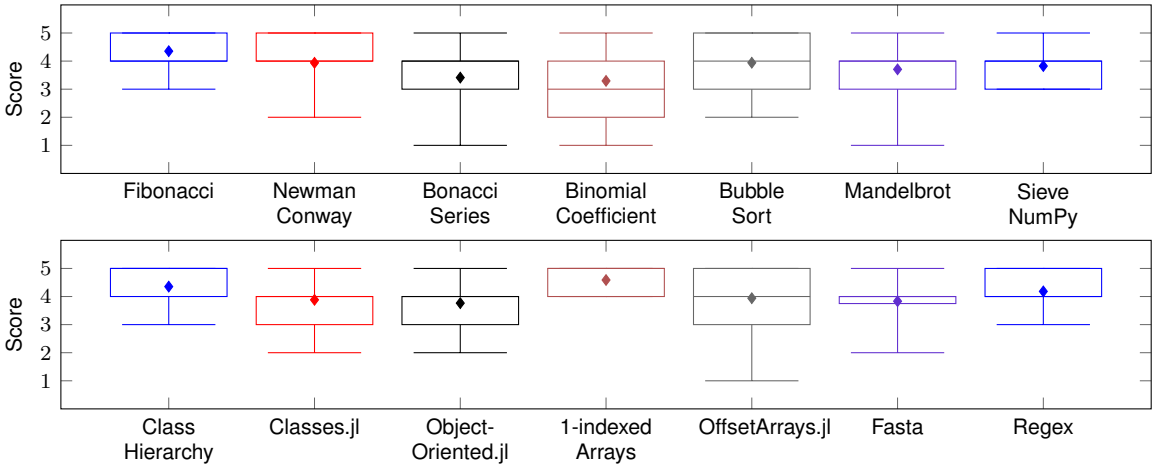


Figure 6.9: Code Readability and Pragmatics Evaluation



Due to the language disparities, we did not fully translate this library to Julia. We used the integrated unit-test suite to verify the correctness of our translation, of which PyJL currently covers 20%. The current main issue is that some translations require the use of Julia's unsafe functions, such as `unsafe_convert`, which do not guarantee that objects are kept alive by Julia's GC mechanism during foreign calls. This still requires a more careful analysis. We leave this as a topic for future studying.

6.4 Evaluating Code Pragmatics

To validate the pragmatics of the transpiled source code, we conducted an online survey, which was published on Julia's official forum. Participants were shown a sample of Python source code with its corresponding Julia translation generated by PyJL and asked to rank the readability and pragmatics of translations using a Likert scale ranging from 1-5. In total, we had 17 participants.

The vast majority of our participants were experienced programmers, with 80% having more than 5 years of experience. As one can see from figures 6.7 and 6.8, 64% of our participants have more than 5 years of experience with Python, whereas 36% have more than 5 years of experience with Julia.

Listing 6.10: Binomial Coefficient Python

```
def binomial_coef(n: int, k: int):
    C = [[0 for x in range(k + 1)]
        for x in range(n + 1)]
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i - 1][j - 1]
                    + C[i - 1][j]
    return C[n][k]
```

Listing 6.11: Binomial Coefficient Julia

```
function binomial_coef(n::Int, k::Int)
    C = [[0 for x = 0:k] for x = 0:n]
    for i in 0:n
        for j in 0:min(i, k)
            if j == 0 || j == i
                C[i+1][j+1] = 1
            else
                C[i+1][j+1] = C[i][j] + C[i][j+1]
            end
        end
    end
    return C[n+1][k+1]
end
```

Additionally, 29% of participants use Python daily, whereas 62% use Julia daily. Regarding our target audience, 50% have a Computer Science background, 20% have a mathematics background, and the remaining 30% have experience in various fields, such as Architecture, Medical Imaging, Chemistry, and Physics.

From our testing, we observed that the average rating for each test was always above 3, indicating that PyJL is capable of translating Python to human-readable and pragmatic Julia source code. The results can be seen in figure 6.9, where the mean values are represented as dots. Simple examples, such as implementations of the Fibonacci or Newman-Conway sequence, managed average scores that were at or above 4. However, some translations still require optimizations. To demonstrate one such scenario, we decided to evaluate an implementation that calculates the Binomial Coefficient, where we observed the lowest average score of `3.3`. Listing 6.10 shows the Python implementation and listing 6.11 shows the code generated by PyJL.

The most notable aspect that requires improving is the translation of Python's multidimensional arrays, shown on lines 2 and 3 of listing 6.11. Notice that the translated Julia source code uses nested arrays as in Python, which has large overheads when compared to Julia's matrices. Another important factor to consider is the translation of 1-indexed arrays. In this scenario, PyJL was not able to apply any optimizations to the loop's range, as the variable `j` is compared against the literal value `0` in the inner loop. We leave these optimizations as recommendations for future versions.

To compare the two indexing translation mechanisms, we chose the implementation of combination sort discussed in section 3.2.4. Notice from figure 6.9 that the translation using 1-indexed arrays has a higher average score of 4.6 compared to the 3.9 when using *OffsetArrays*. Improving the translation of Python's indexing would be beneficial, as we observed that over 90% of participants prefer this approach over using *OffsetArrays*.

To evaluate PyJL's class translation mechanisms, we chose the examples shown in section 3.2.7, where we defined the `Person`, `Student`, `Worker` and `StudentWorker` classes. We evaluated the translation of single and multiple inheritance. Regarding the single inheritance results, using the abstract type

hierarchy yielded the highest average score of 4.2 compared to the average score of 3.9 when using the *Classes* package. We also observed that over 60% of participants prefer using the abstract type hierarchy over the *Classes* package. It is important to note that 20% of participants found both approaches to be equally pragmatic. Regarding the translation of multiple inheritance when using the *ObjectOriented* package, we observed an average score of 3.8, which is lower than the previous approaches. This is likely due to the Python-like syntax used by the package, which can be less pragmatic for Julia programmers.

Using the feedback received from the quiz, we also managed to improve the readability of our NumPy translations. We performed several improvements to the NumPy Sieve implementation presented in section 5.3.1. The suggestions reduced the verbosity of our previous solution, making the generated code more idiomatic.

Lastly, we chose two larger code samples to test if code pragmatics were affected. The first is the Fasta benchmark that was discussed in section 6.2.2 and the second is a regex benchmark, which we modified by removing the use of Python's `multiprocessing` module. The Fasta benchmark is available in appendix B, while the Regex benchmark is available in our public repository [49]. Since these tests were more extensive, we set them as optional. We got a total of 12 and 11 responses for the Fasta and Regex benchmarks, respectively. The average result for the Fasta benchmark was 3.7. One of the most prominent aspects noticed by the participants is the lack of annotations when translating Python's lists to Julia's arrays, which results in slower performance, as discussed in section 6.2.2. Regarding the Regex benchmark, we observed a result of 4.2. The higher score is most likely related to the fact that this test focuses on string manipulation, which uses similar syntax in both languages.

6.5 Automatic vs Manual Translation

To evaluate the benefits of automatic over manual translation, we asked Julia programmers to manually translate Python source code to Julia. We chose 5 participants that represent experienced and intermediate Julia programmers. Each participant was told to translate 5 code excerpts that stressed various functionalities supported by the transpiler, such as the translation of data structures or classes. More complex examples were also chosen, such as the implementation of Mandelbrot from section 3.2.9 that demonstrates the different scoping rules of both languages. Participants were allowed to use the Julia documentation during the translation process. We registered the time taken by each participant to translate individual code excerpts and the errors made during the translation process. Unit tests were also supplied to allow participants to verify the translation's correctness. Table 6.1 shows the total time taken in minutes for participants to translate the individual code excerpt. Instances where the manually translated code did not pass the provided unit tests are marked with $*$.

Participants	Fibonacci	Combination Sort	Binomial Coefficient	Classes	Mandelbrot
1	0:55	3:33	2:17	3:42	2:12
2	0:28	2:25	2:17	3:04	2:38
3	1:21	6:40	3:00*	3:43*	1:42*
4	1:27	8:55	25:41	8:50	16:54
5	0:37	2:52	9:59	4:39	3:46

Table 6.1: Time in minutes for manual translation

The results obtained vary between the participants. This is not only due to the participants experience but also because each used a different translation approach. These results demonstrate how inconsistent manual translations can be, as each programmer uses different implementation strategies. In this regard, automatic translation took only 4 seconds on a machine with an Intel(R) Core(TM) i7 4790K @4.4GHz with 16GB of RAM running Linux. This includes not only code formatting but also the translation of the provided unit tests, which allowed verifying the transpiled code.

Another noteworthy aspect were the errors registered during translations. Given the similarities between Python and Julia's syntaxes, a common strategy we observed was that users copied the Python code and changed only the necessary constructs and keywords to match Julia. However, this strategy ended up creating bugs, which slowed down translation time. With the provided code excerpts, translation errors were trivial to find, requiring programmers little time to have a working implementation in Julia. This scenario would likely change with the translation of larger code bases. Given the complexity of large-scale software solutions, the time required to fix software bugs would render manual translation as an unsuitable strategy.

6.6 Extensibility

The previous sections focused on the PyJL transpiler. However, when regarding extensibility, the whole pipeline of Py2Many must be considered. As Python is frequently updated, Py2Many has to be able to evolve as new language functionalities are added. Py2Many's flexible architecture already allows for some flexibility, namely when regarding the addition of new Rewriters or Transformers. Still, this flexibility should be applicable to the mapping of function calls to external libraries. Our goal was to create a modular approach to add call translation tables and facilitate transpiler updates.

Therefore, we developed an extension that allows programmers to include external Python translation modules to house call translation tables. The mechanism searches for such modules in a predetermined `pyxx/external/modules` directory, where `pyxx` is the name of the transpiler extension we intend to use, such as `pyjl`. Furthermore, as type information is a key aspect for the translation process, we also allow external modules to map Python functions to their respective return types.

Another concern is the differentiation of generic and language-specific rewriters or transformers.

Whenever possible, we used generic phases applicable for all transpilers. This includes the translation of `for-else` and `while-else` constructs, described in section 3.2.3. Distinguishing generic transformation phases from language-specific ones is crucial to save development time for other supported languages.

7

Conclusions

Contents

7.1 Coverage 73
7.2 Future Work 74

Automatic and reliable translation between programming languages offers the possibility to bridge the gap between established languages, and newer and/or less developed ones. Furthermore, automatic translation provides an almost risk-free code-base migration process. This research explored the use of transpilers to translate existing code-bases from Python, a popular and well-established programming language, to the newer Julia programming language, with the intent of speeding up its development and growth.

The Py2Many transpiler translates Python source code to many C-like programming languages. It provides a modular framework, where the support for each language is added as an extension. We focused on developing the PyJL extension, which builds upon Py2Many's framework to transpile Python to human-readable and pragmatic Julia source code, allowing Julia programmers to further modify it. As such, a predominant goal was to guarantee that the generated code was idiomatic, almost as if it was written by seasoned Julia programmers. Furthermore, PyJL requires very little programmer input to generate running Julia code. This was further improved with the addition of an external type inference mechanism, which reduced the work required to manually annotate the Python source code.

Initially, we set 4 different goals, which were addressed throughout the research:

- Correctness: We tested the correctness of the generated code by using unit tests. Py2Many's test suite and a small subset of CPython's official test suite were used to validate each supported functionality. Furthermore, library unit-tests were also transpiled to Julia, to assert the correctness of the translation.

- Intelligibility and Pragmatics: PyJL uses several techniques to improve the readability of the output source. As PyJL's main goal is to produce human-readable and pragmatically correct source code, we validated these two aspects by conducting an online survey from a pool of experienced Julia programmers. Despite still requiring some improvements to reach a human-like output, the results show that the generated code is human-readable and respects the pragmatics of Julia.

- Performance: Despite performance not being a focus of PyJL, we observed large performance increases when transpiling Python to Julia. Only one benchmark produced slower running times when compared to Python. Still, this only required annotating one line of code to make it an order of magnitude faster than the Python implementation. We also compared the performance of highly optimized Python programs that used *NumPy*, where Julia either matches or outperforms Python.

- Dependencies: We have addressed three dependency scenarios. The first is handling dependencies between multiple Python modules, where PyJL translates Python's import mechanism to Julia. Furthermore, it also propagates type information across modules for type inference.

  The second scenario is importing Python registered packages. If Julia provides equivalent functionalities, then PyJL translates Python package calls into calls to Julia's equivalent libraries. Oth-

	Input Language	Target Language	Intelligible	Coverage	Upkeep	Performance
Linj	Common Lisp	Java	✓✓✓	✓✓	✓	$5-6\times$
Fortran-Python	Python	Fortran	✓✓	✓✓	✓	$6-10\times$
JSweet	Java	JavaScript	✓✓	✓✓	✓✓✓	✗
Py2Many/PyJL	Python	Julia	✓✓	✓✓	✓✓✓	$1.3-17.5\times$

Table 7.1: Updated State-of-the-art Evaluation

erwise, PyJL uses PyCall to call Python's libraries through the FFI.

The third scenario is translating accesses to DLLs. We translated a subset of the `ctypes` interface to native Julia, allowing us to identify low-level incompatibilities between the two languages.

In section 2.7, we compared the different transpiler implementations. At the time, we had preliminary results regarding performance and code readability. After our evaluation process, we now have more accurate data to compare PyJL against other state-of-the-art solutions. Table 7.1 contains the new results. We have increased the overall coverage of PyJL, not only for a subset of Python's standard library, but also for some external libraries, such as NumPy. Our new performance results show that the generated Julia code is between $1.3\times$ to $17.5\times$ faster than Python. This is after judiciously annotating the Python source code, requiring no other implementation changes. Furthermore, the optimizations applied to the generated source code makes it more readable, while respecting the pragmatics of Julia.

7.1 Coverage

Python libraries commonly have many dependencies with external libraries. Some of these library calls can be mapped to Julia's equivalent libraries or simulated using Julia's language features. However, there is still a large subset of Python for which there are no equivalent functionalities in Julia. One solution for such cases is to use the FFI to call Python's libraries. PyJL uses the *PyCall* package, which provides a wrapper around the FFI to call Python functions while offering several crucial functionalities, such as automatic type conversion. Resorting to FFI's provides a temporary solution to speedup these translations by creating wrapper libraries around already existing Python libraries. As new Julia libraries emerge, programmers can adapt PyJL and replace the FFI calls into calls to these libraries, eliminating any dependencies to Python.

Nonetheless, some Python operations are impossible to map to Julia, as they are dependent on Python's execution environment. This is the case of the `eval` function, which parses and evaluates Python expressions. In such cases, PyJL must resort to *PyCall* to evaluate the expressions. As an example, the Python call `eval("1+2")` is translated by PyJL to `py"1+2"`.

Despite PyJL's successful translation of a large variety of Python features, the translation of large

legacy code bases is still a topic that requires more investigation. The main issue is that legacy code typically relies on older language functionalities, which might not be supported by the transpiler. Ideally, such code bases should be separated into multiple independent components and translated individually. A noteworthy disadvantage is that this process requires knowledge about the existing code base and awareness of potential translation restrictions, which is challenging given the complexity of large-scale software solutions.

7.2 Future Work

When converting between programming languages, transpilers typically target the most common scenarios, focusing on the input language's most used constructs. However, programmers commonly use functionalities that cause inconsistencies when translated to the target language. This refers to the general rule that "any language construct that can be abused will be abused" [3]. As an example, consider translating calls to Python's `locals()` function, which returns a dictionary containing locally defined symbols. Despite Julia providing a `locals` macro, which retrieves the symbols and values of all locally defined variables, both Python and Julia define distinct scoping rules that affect the resulting dictionaries. Such incompatibilities demonstrate the difficulty of translations when aiming for a high semantic equivalence between the input and output languages [3].

Regarding module dependencies, the most notable aspect that still requires a more extensive study is the mapping of Python's package structure to Julia. PyJL currently offers a solution that uses Julia's *FromFile* package to map Python's imports. Nonetheless, some of the functionalities of Python's import machinery are still unsupported. This is the case of Python's `importlib` module, which can dynamically add and remove meta path finders to load Python modules.

Exceptions are another issue, as there are Python exceptions that do not have a deterministic corresponding exception in Julia. As an example, consider the Python calls to `math.sqrt(-1)` and `float("-")` that both raise a `ValueError` exception. These can be translated to Julia as `sqrt(-1)` and `float("-")` respectively, but the first call throws a `DomainError` exception and the second throws an `ArgumentError` exception. Another problem is that there are some Python exceptions, such as the `ZeroDivisionError` that is raised in Python when the quotient of a division operation is zero, that are not considered exceptions in Julia. In Julia, a division by zero instead returns `Inf`, representing infinity. This topic still requires further investigation.

Regarding performance, the translation of Python's lists remains a challenge. When no annotations are provided, inferring the types of lists is dependent on the ability to statically analyse all list concatenation operations. The inference of lists could further be improved by integrating pytype's `annotate-ast` tool, which already includes analysis mechanisms to infer list types. This would largely improve the

performance of the generated code.

Despite PyJL supporting the translation of Python's unit tests and parameterized unit tests, it would also be beneficial to map Python's `doctest` module, which identifies comments in the form of interactive Python sessions, converting them to tests. In this regard, Julia offers the *Documenter*[1] package, which parses Julia documentation strings and identifies doctests. As a subset of CPython's unit tests are written as doctests, this could increase the current unit test coverage.

Lastly, it is important to remember that Py2Many and the PyJL extension are written in Python. A future goal is to create a Meta-Transpiler, where a transpiler translates itself to the target language. This would further decrease any dependencies from Python. We are still far from achieving this goal, as some modules are difficult to simulate in Julia. This is the case of Python's `functools` module for higher-order functions. In this regard, it is also relevant to consider the current parsing mechanism. Py2Many currently uses Python's default parsing mechanism included in the `ast` package. A custom parser could enhance the available information at transpilation time. The improved translation capabilities could bring us closer to the goal of building a Meta-transpiler, while offering improved library translation capabilities.

[1] Documenter - A documentation generator for Julia: https://github.com/JuliaDocs/Documenter.jl

Bibliography

[1] *Transpiling Python to Julia using PyJL*. Zenodo, Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6332890

[2] M. Marcelino and A. M. Leitão, "Extending PyJL - Transpiling Python Libraries to Julia," in *11th Symposium on Languages, Applications and Technologies (SLATE 2022)*, ser. Open Access Series in Informatics (OASIcs), J. a. Cordeiro, M. J. a. Pereira, N. F. Rodrigues, and S. a. Pais, Eds., vol. 104. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 6:1–6:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16752

[3] A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Edinburgh International Conference Centre, Scotland, UK: Semantic Designs, 2004, pp. 625–634.

[5] M. Rebaudengo, M. S. Reorda, and M. T. Massimo, "A source-to-source compiler for generating dependable software," *Workshop on Source Code Analysis and Manipulation (SCAM2001)*, Nov 2001.

[6] I. Corporation, "MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users," 1979.

[7] R. Taylor and P. Lemmons, "Upward Migration Part 1: Translators Using translation programs to move CP/M-86 programs to CP/M and MS-DOS," *BYTE Publications Inc.*, 1982.

[8] Tiobe, "TIOBE Index," Jul 2022, [Online. Retrieved July 6th, 2022 from: https://www.tiobe.com/tiobe-index/].

[9] Python Software Foundation, "Python Package Index," March 2001, [Online. Retrieved July 6th, 2022 from: https://pypi.org/].

[10] P. Norvig, “Python for Lisp Programmers,” [Online. Retrieved July 6th, 2022 from: http://norvig.com/python-lisp.html].

[11] J. Hugunin, “Python and java: The best of both worlds,” in *Proceedings of the 6th international Python conference*, vol. 9. Reston, VA: Citeseer, 1997, pp. 2–18.

[12] J. Hugunin, “IronPython Home Page,” 2013, [Online. Retrieved July 6th, 2022 from: https://ironpython.net/].

[13] G. Van Rossum and F. L. Drake Jr, “Python Reference Manual,” 1995.

[14] J. Community, “Julia Packages,” August 2017, [Online. Retrieved July 6th, 2022 from: https://julialang.org/packages/].

[15] JuliaPy, “PyCall - Package to call Python functions from the Julia language,” 2013.

[16] JuliaInterop, “JavaCall - Call Java from Julia,” 2013.

[17] ——, “RCall - Call R from Julia,” 2015.

[18] A. M. Leitão, “The next 700 programming libraries,” in *Proceedings of the 2007 International Lisp Conference*, ser. ILC ’07. New York, NY, USA: Association for Computing Machinery, 2007.

[19] JuliaCN, “Py2Jl,” 2021, [Online. Retrieved October 19th, 2022 from: https://github.com/JuliaCN/Py2Jl.jl].

[20] M. Marcelino and A. Menezes Leitão, “Pyjl implementation,” 2021, https://github.com/MiguelMarcelino/py2many.

[21] A. M. Leitao, “Migration of Common Lisp Programs to the Java Platform - The Linj Approach,” in *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*, 2007, pp. 243–251.

[22] M. Bysiek, A. Drozd, and S. 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)*, 2016, pp. 9–18.

[23] A. Aiken and B. Murphy, “Static type inference in a dynamically typed language,” in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’91. New York, NY, USA: Association for Computing Machinery, 1991, p. 279–290. [Online]. Available: https://doi.org/10.1145/99583.99621

[24] Renaud Pawlak, “JSweet - A transpiler from Java to TypeScript/JavaScript,” 2015, retrieved October 25th, 2022 from: https://www.jsweet.org/.

[25] P. Japikse, K. Grossnicklaus, and B. Dewey, *Building Web Applications with Visual Studio 2017*. Apress, 01 2017.

[26] H. Lunnikivi, K. Jylkkä, and T. Hämäläinen, "Transpiling Python to Rust for Optimized Performance," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2020, pp. 127–138.

[27] J. Bispo and J. M. Cardoso, "Clava: C/C++ Source-to-Source Compilation using LARA," *SoftwareX*, vol. 12, 2020.

[28] I. Moore, M. Wolczko, and T. Hopkins, "Babel - A Translator From Smalltalk into CLOS," 1994, sun Microsystems Laboratories Inc.

[29] A. Zanelli, T. Sartor, P. Bowyer, and D. Moritz, "Prometeo - An experimental Python-to-C transpiler," 2017, [Online. Retrieved 19th October, 2022 from: https://github.com/zanellia/prometeo].

[30] JuliaLang, "The julia language - 1.7.3," May 2022.

[31] A. Sharma, L. Martinelli, J. Konchunas, and J. Vandenberg, "Py2many: Python to many clike languages transpiler," 2015, retrieved September 21st, 2022 from: https://github.com/adsharma/py2many.

[32] JuliaArrays, "Offsetarrays.jl," Jan 2014, retrieved June 11th, 2022 from: https://github.com/JuliaArrays/OffsetArrays.jl.

[33] B. Lauwens, "Resumablefunctions.jl," August 2017, retrieved September 29th, 2022 from: https://github.com/BenLauwens/ResumableFunctions.jl.

[34] R. Plevin, "Classes.jl: A simple, Julian approach to inheritance of structure and methods," November 2021, retrieved August 22nd, 2022 from: https://github.com/rjplevin/Classes.jl.

[35] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington, "A Monotonic Superclass Linearization for Dylan," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 69–82. [Online]. Available: https://doi.org/10.1145/236337.236343

[36] M. Lutz, *Learning Python - 3rd Edition*. O'Reilly Media, Inc., 2007.

[37] Python Software Foundation, "Python unittest module," retrieved October 20th, 2022 from: https://docs.python.org/3/library/unittest.html.

[38] G. van Rossum, J. Lehtosalo, and L. Langa, September 2014, retrieved October 27th, 2022 from: https://peps.python.org/pep-0484/.

[39] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby, "Python 3 types in the wild: A tale of two type systems," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 57–70. [Online]. Available: https://doi.org/10.1145/3426422.3426981

[40] M. Hassan, C. Urban, M. Eilers, and P. Müller, "Maxsmt-based type inference for python 3," 2018.

[41] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[42] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python Probabilistic Type Inference with Natural Language Support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 607–618. [Online]. Available: https://doi.org/10.1145/2950290.2950343

[43] Microsoft, "pyright: Static type checker for Python," March 2015, [Online. Retrieved October 3rd, 2022 from: https://github.com/microsoft/pyright].

[44] Google, "Pytype: A static type analyzer for Python code," March 2015, [Online. Retrieved October 3rd, 2022 from: https://github.com/google/pytype].

[45] Python Software Foundation, "MyPy: Optional static typing for Python," 2012, [Online. Retrieved October 3rd, 2022 from: https://github.com/python/mypy.

[46] ——, "Ctypes," retrieved July 11th, 2022 from: https://docs.python.org/3/library/ctypes.html.

[47] ——, "Python Glossary," retrieved October 11th, 2022 from: https://docs.python.org/3/glossary.html.

[48] G. van Rossum and P. D. Team, *The Python Language Reference - Release 3.9.7*, August 2021, [Online. Retrieved June 9th, 2022 from: https://docs.python.org/release/3.9.7/].

[49] M. Marcelino, "PyJL Translations Repository," https://github.com/MiguelMarcelino/pyjl_translations.

[50] I. Gouy, "The Computer Language Benchmarks Game," 2007, retrieved September 13th, 2022 from: https://salsa.debian.org/benchmarksgame-team/benchmarksgame.

[51] M. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: https://books.google.pt/books?id=STDBswEACAAJ

[52] K. Deininger, "python-reprojector: Python library for coordinate transformation," retrieved October 13th, 2022 from: https://gitlab.com/geo-bl-ch/python-reprojector/-/tree/master/.

[53] I. Enthought, "pywin32-ctypes," retrieved October 13th, 2022 from: https://github.com/enthought/pywin32-ctypes.

Julia Classes using ObjectOriented.jl

```
@oodef mutable struct Person
    name::String

    function new(name::String)
        @mk begin
            name = name
        end
    end

    function get_id(self::Person)::String
        return self.name
    end
end

@oodef mutable struct Student <: Person
    name::String
    student_number::Int64
    domain::String

    function new(name::String, student_number::Int64, domain::String = "school.student.pt")
        @mk begin
            name = name
            student_number = student_number
            domain = domain
        end
    end

    function get_id(self)
        return "$(self.name) - $(self.student_number)"
    end
end

@oodef mutable struct Worker <: Person
    name::String
    company_name::String
    hours_per_week::Int64

    function new(name::String, company_name::String, hours_per_week::Int64)
        @mk begin
            name = name
            company_name = company_name
            hours_per_week = hours_per_week
        end
    end
end

@oodef mutable struct StudentWorker <: {Student, Worker}
    schedule_conflicts::Bool

    function new(
        name::String,
        student_number::Int64,
        domain::String,
        company_name::String,
        hours_per_week::Int64,
        schedule_conflicts::Bool)
        @mk begin
            Student(name, student_number, domain)
            Worker(name, company_name, hours_per_week)
            schedule_conflicts = schedule_conflicts
        end
    end
end
```

B

Performance Benchmarks

B.1 N-Body Julia Translation

```
using Printf

function combinations(l)::Vector
    result = []
    for x = 0:length(l)-2
        ls = l[x+2:end]
        for y in ls
            push!(result, (l[x+1], y))
        end
    end
    return result
end

PI = 3.141592653589793
SOLAR_MASS = 4 * PI * PI
DAYS_PER_YEAR = 365.24
BODIES = Dict(
    "sun" => ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),
    "jupiter" => (
        [4.841431442464721, -1.1603200440274284, -0.10362204447112311],
        [
            0.001660076642744037 * DAYS_PER_YEAR,
            0.007699011184197404 * DAYS_PER_YEAR,
            -6.90460016972063e-05 * DAYS_PER_YEAR,
        ],
        0.0009547919384243266 * SOLAR_MASS,
    ),
    "saturn" => (
        [8.34336671824458, 4.124798564124305, -0.4035234171143214],
        [
            -0.002767425107268624 * DAYS_PER_YEAR,
            0.004998528012349172 * DAYS_PER_YEAR,
            2.3041729757376393e-05 * DAYS_PER_YEAR,
        ],
        0.0002858859806661308 * SOLAR_MASS,
    ),
    "uranus" => (
        [12.894369562139131, -15.111151401698631, -0.22330757889265573],
        [
            0.002964601375647616 * DAYS_PER_YEAR,
            0.0023784717395948095 * DAYS_PER_YEAR,
            -2.9658956854023756e-05 * DAYS_PER_YEAR,
        ],
        4.366244043351563e-05 * SOLAR_MASS,
    ),
    "neptune" => (
        [15.379697114850917, -25.919314609987964, 0.17925877295037118],
        [
            0.002680677724903893 * DAYS_PER_YEAR,
            0.001628241700382423 * DAYS_PER_YEAR,
            -9.515922545197159e-05 * DAYS_PER_YEAR,
        ],
        5.1513890204661145e-05 * SOLAR_MASS,
    ),
)
SYSTEM = collect(values(BODIES))
PAIRS = combinations(SYSTEM)
function advance(dt, n, bodies = SYSTEM, pairs = PAIRS)
    for i = 0:n-1
        for (((x1, y1, z1), v1, m1), ((x2, y2, z2), v2, m2)) in pairs
```

```
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            mag = dt * ((dx * dx + dy * dy) + dz * dz)^-1.5
            b1m = m1 * mag
            b2m = m2 * mag
            v1[1] -= dx * b2m
            v1[2] -= dy * b2m
            v1[3] -= dz * b2m
            v2[1] += dx * b1m
            v2[2] += dy * b1m
            v2[3] += dz * b1m
        end
        for (r, (vx, vy, vz), m) in bodies
            r[1] += dt * vx
            r[2] += dt * vy
            r[3] += dt * vz
        end
    end
end

function report_energy(bodies = SYSTEM, pairs = PAIRS, e = 0.0)
    for (((x1, y1, z1), v1, m1), ((x2, y2, z2), v2, m2)) in pairs
        dx = x1 - x2
        dy = y1 - y2
        dz = z1 - z2
        e -= m1 * m2 / ((dx * dx + dy * dy) + dz * dz)^0.5
    end
    for (r, (vx, vy, vz), m) in bodies
        e += m * ((vx * vx + vy * vy) + vz * vz) / 2.0
    end
    @printf("%.9f\n", e)
end

function offset_momentum(ref, bodies = SYSTEM, px = 0.0, py = 0.0, pz = 0.0)
    for (r, (vx, vy, vz), m) in bodies
        px -= vx * m
        py -= vy * m
        pz -= vz * m
    end
    r, v, m = ref
    v[1] = px / m
    v[2] = py / m
    v[3] = pz / m
end

function main(n, ref = "sun")
    offset_momentum(BODIES[ref])
    report_energy()
    advance(0.01, n)
    report_energy()
end

if abspath(PROGRAM_FILE) == @__FILE__
    main(parse(Int, append!([PROGRAM_FILE], ARGS)[2]))
end
```

B.2 Fasta Julia Translation

```
using BisectPy
using ResumableFunctions

alu = "GGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGC
GGGCGGATCACCTGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACATGGTGAAACCCCGT
CTCTACTAAAAATACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCTGTAATCCCAGCTAC
TCGGGAGGCTGAGGCAGGAGAATCGCTTGAACCCGGGAGGCGGAGGTTGCAGTGAGCCGA
GATCGCGCCACTGCACTCCAGCCTGGGCGACAGAGCGAGACTCCGTCTCAAAAA"
iub = collect(
    zip(
        ["a", "c", "g", "t", "B", "D", "H", "K", "M", "N", "R", "S", "V", "W", "Y"],
        append!([0.27, 0.12, 0.12, 0.27], repeat([0.02], 11)),
    ),
)
homosapiens = [
    ("a", 0.302954942668),
    ("c", 0.1979883004921),
    ("g", 0.1975473066391),
    ("t", 0.3015094502008),
]
@resumable function genRandom(ia = 3877, ic = 29573, im = 139968)
    seed = 42
    imf = float(im)
    while true
        seed = (seed * ia + ic) % im
        @yield seed / imf
    end
end

Random = genRandom()
function makeCumulative(table)::Tuple
    P = []
    C = []
    prob = 0.0
    for (char, p) in table
        prob += p
        push!(P, prob)
        push!(C, char)
    end
    return (P, C)
end

function repeatFasta(src::String, n::Int64)
    width = 60
    r = length(src)
    s = src * src * src[begin:n%r]
    for j in (0:n÷width-1)
        i = j * width % r
        println(s[(i+1):i+width])
    end
    if (n % width) != 0
        println(s[(length(s)-n%width+1):end])
    end
end

function randomFasta(table, n::Int64)
    width = 60
    r = (0:width-1)
    gR = Random
    bb = bisect_right
```

```
    jn = x -> join(x, "")
    probs, chars = makeCumulative(table)
    for j in (0:n÷width-1)
        x = jn([chars[bb(probs, gR())] for i in r])
        println(x)
    end
    if (n % width) != 0
        println(jn([chars[bb(probs, gR())] for i in (0:n%width-1)]))
    end
end

function main()
    n = parse(Int, append!([PROGRAM_FILE], ARGS)[2])
    println(">ONE Homo sapiens alu")
    repeatFasta(alu, n * 2)
    println(">TWO IUB ambiguity codes")
    randomFasta(iub, n * 3)
    println(">THREE Homo sapiens frequency")
    randomFasta(homosapiens, n * 5)
end

main()
```

B.3 Binary Trees Julia Translation

```
function make_tree(depth::Int64)::Tuple
    #=  Trees or tuples, final leaves have None as values.  =#
    return depth == 0 ? ((nothing, nothing)) :
           ((make_tree(depth - 1), make_tree(depth - 1)))
end

function check_node(left, right)::Int64
    #=
        Count 1 for each node found.
        (Unpacking directly in the parameters is faster)
         =#
    return left === nothing ? (1) : ((1 + check_node(left...)) + check_node(right...))
end

function run(depth::Int64)::Int64
    #=
        Makes a tree then checks it (parse all nodes and count).
        This function is global for multiprocessing purposes.
         =#
    return check_node(make_tree(depth)...)
end

function main(requested_max_depth, min_depth = 4)
    max_depth = max(min_depth + 2, requested_max_depth)
    stretch_depth = max_depth + 1
    println("stretch tree of depth $(stretch_depth)\t check: $(run(stretch_depth))")
    long_lived_tree = make_tree(max_depth)
    mmd = max_depth + min_depth
    for test_depth = min_depth:2:stretch_depth-1
        tree_count = 2^(mmd - test_depth)
        check_sum = sum(map(run, repeat([(test_depth,)...], tree_count)))
        println("$(tree_count)\t trees of depth $(test_depth)\t check: $(check_sum)")
    end
    println(
        "long lived tree of depth $(max_depth)\t check:
        ↪  $(check_node(long_lived_tree...))",
    )
end

if abspath(PROGRAM_FILE) == @__FILE__
    main(parse(Int, append!([PROGRAM_FILE], ARGS)[2]))
end
```

B.4 Neural Network

```
#=
A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network. =#
using LinearAlgebra
using Random

abstract type AbstractNetwork end
mutable struct Network <: AbstractNetwork
    sizes::Vector{Int64}
    num_layers::Int64
    biases
    weights

    Network(
        sizes::Vector{Int64},
        num_layers = length(sizes),
        biases::Vector{Matrix} = [randn(y, 1) for y in sizes[2:end]],
        weights::Vector{Matrix} =
            [randn(y, x) for (x, y) in zip(sizes[begin:end-1], sizes[2:end])],
    ) = begin
        new(sizes, num_layers, biases, weights)
    end
end

function feedforward(self::AbstractNetwork, a::Matrix)::Matrix
    #= Return the output of the network if ``a`` is input. =#
    for (b, w) in zip(self.biases, self.weights)
        a = sigmoid((w * a) .+ b)
    end
    return a
end

function SGD(
    self::AbstractNetwork,
    training_data::Vector,
    epochs::Int64,
    mini_batch_size::Int64,
    eta::Float64,
    test_data::Union{Vector, Nothing} = nothing,
)
    #= Train the neural network using mini-batch stochastic gradient descent. =#
    training_data = collect(training_data)
    n = length(training_data)
    if test_data !== nothing
        test_data = collect(test_data)
        n_test = length(test_data)
    end
    for j = 0:epochs-1
        shuffle(training_data)
        mini_batches = [training_data[k+1:k+mini_batch_size] for k = 0:mini_batch_size:n-1]
        for mini_batch in mini_batches
            update_mini_batch(self, mini_batch, eta)
        end
        if test_data !== nothing
            println("Epoch $(j) : $(evaluate(self, test_data)) / $(n_test)")
        else
            println("Epoch $(j) complete")
        end
    end
end
```

```
61
62  function update_mini_batch(self::AbstractNetwork, mini_batch::Vector{Tuple},
63          eta::Float64)
64      #= Update the network's weights and biases by applying
65              gradient descent using backpropagation to a single mini batch. =#
66      nabla_b = [zeros(Float64, size(b)) for b in self.biases]
67      nabla_w = [zeros(Float64, size(w)) for w in self.weights]
68      for (x, y) in mini_batch
69          (delta_nabla_b, delta_nabla_w) = backprop(self, x, y)
70          nabla_b = [nb + dnb for (nb, dnb) in zip(nabla_b, delta_nabla_b)]
71          nabla_w = [nw + dnw for (nw, dnw) in zip(nabla_w, delta_nabla_w)]
72      end
73      self.weights =
74          [w - (eta / length(mini_batch)) * nw for (w, nw) in zip(self.weights, nabla_w)]
75      self.biases =
76          [b - (eta / length(mini_batch)) * nb for (b, nb) in zip(self.biases, nabla_b)]
77  end
78
79  function backprop(self::AbstractNetwork, x::Matrix, y::Matrix)::Tuple
80      #= Return a tuple ``(nabla_b, nabla_w)`` representing the
81              gradient for the cost function =#
82      nabla_b = [zeros(Float64, size(b)) for b in self.biases]
83      nabla_w = [zeros(Float64, size(w)) for w in self.weights]
84      activation::Matrix = x
85      activations::Vector{Matrix} = [x]
86      zs::Vector{Matrix} = []
87      for (b, w) in zip(self.biases, self.weights)
88          z = (w * activation) .+ b
89          push!(zs, z)
90          activation = sigmoid(z)
91          push!(activations, activation)
92      end
93      delta = cost_derivative(self, activations[end], y) .* sigmoid_prime(zs[end])
94      nabla_b[end] = delta
95      nabla_w[end] = (delta * LinearAlgebra.transpose(activations[end-1]))
96      for l = 2:self.num_layers-1
97          z = zs[end-l+1]
98          sp = sigmoid_prime(z)
99          delta = (LinearAlgebra.transpose(self.weights[end-l+2]) * delta) .* sp
100         nabla_b[end-l+1] = delta
101         nabla_w[end-l+1] = (delta * LinearAlgebra.transpose(activations[end-l]))
102     end
103     return (nabla_b, nabla_w)
104 end
105
106 function evaluate(self::AbstractNetwork, test_data::Vector)
107     #= Return the number of test inputs for  which the neural
108             network outputs the correct result. =#
109     test_results = [(argmax(@view feedforward(self, x)[:]) - 1, y)
110         for (x, y) in test_data]
111     return sum((Int(x == y) for (x, y) in test_results))
112 end
113
114 function cost_derivative(
115     self::AbstractNetwork,
116     output_activations::Matrix,
117     y::Matrix,
118 )::Matrix
119     #= Return the vector of partial derivatives     for the output activations. =#
120     return output_activations .- y
121 end
122
123
```

```
124 function sigmoid(z::Matrix)::Matrix
125     #= The sigmoid function. =#
126     return 1.0 ./ (1.0 .+ e .^ -z)
127 end
128 
129 function sigmoid_prime(z::Matrix)::Matrix
130     #= Derivative of the sigmoid function. =#
131     return sigmoid(z) .* (1 .- sigmoid(z))
132 end
```