COMP6013 Project Proposal - Rattlesnake

Contents

1 Introduction

1.1 Background

This project, Rattlesnake, will be a compiler, and its associated run-time libraries, which translate Python source code into Rust source code.

The initial target will be to translate an arbitrary subset of Python 3.10 into Nightly Rust, but an eventual goal is complete compatibility with up-to-date Python versions and stable Rust with as low an MSRV (*Minimum Supported Rust Version*, a term used by Rust projects to indicate support) as possible.

The purpose of Rattlesnake is to allow Python code to be compiled into a native executable, which can be run faster, more memory-efficiently, and without requiring a Python interpreter. It would also allow rapid development of Rust libraries, by writing a rough skeleton in Python, translating it to Rust, and then optimising the generated Rust code. Lastly, it could also allow Python projects to use Rust libraries, by leaving the Rust calls in place when translating the Python source.

An eventual goal of Rattlesnake is to have full coverage of Python 3, and thus to be able to translate any Python 3 code into Rust. However, this is a very ambitious goal, and so for the time being, the aim is to support a subset which removes some particularly tricky-to-translate language features (some reflection, decorators, extension modules, async, threading, etc.). The early versions of Rattlesnake will be written in Python, using the `ast` built-in module to generate an AST (*Abstract Syntax Tree*, an easy-to-manipulate representation of code), and then compiled into Rust source code; later versions of Rattlesnake will be built in Rust, using the compiled Python script as a base, and will likely replace `ast` with `pest`[1].

1.2 Aim

The aim of Rattlesnake is to translate Python 3 code into Rust code, as efficiently as possible and with as little loss of functionality as possible. The generated

[1] dragostis *et al.* (2023) 'pest. The Elegant Parser'. Available at: https://pest.rs/ (Accessed: 18 October 2023)

code should also look as readable as possible.

1.3 Objectives

1. Create the run-time Rattlesnake library for Rust, which will provide various standard Python types and functions.
2. Create the run-time Rattlesnake library for Python, which will provide access to various standard Rust types, functions, and macros (and will act as a poly-fill for interpreted Python code).
3. Create the Rattlesnake compiler in Python, using the `ast` module to parse source code.
4. Compile the Python version of Rattlesnake with Rattlesnake, then optimise the output and maintain the Rust version.
    - Compiling Rattlesnake to Rust allows distributing it with Cargo; `cargo install rattlesnake` will install the compiler binary.
    - Optimising and maintaining the Rust version allows it to be faster and more memory-efficient than the plainly-compiled version, at the cost of having to maintain a separate grammar file (if using `pest`) or recompiling the `ast` module (if using Rattlesnake-compiled `ast`) on each major version.
        - However, maintaining the Rust version separately may allow lowering the MSRV/reducing dependence on Nightly features.
5. Create the associated documentation for the Rattlesnake compiler and run-time libraries.
6. Create these documents as part of the dissertation project.

1.4 Product Overview

1.4.1 Scope

The product will take as input Python 3 source code. It will translate that source code into Rust source code, or raise an error if the input contains unsupported features. The output Rust code will be able to be compiled into a native executable, which can be run without a Python interpreter. The output code will depend on the library `rattlesnake`, which will be a Rust library which provides various standard Python types and functions.

There will also be a Python library `rattlesnake`, which will provide various standard Rust types, functions, and macros, and can be used by Python code in order to produce more optimised or efficient Rust code.

1.4.2 Audience

Rattlesnake is intended to be a tool for Python developers who want the speed and memory-efficiency benefits of Rust. It also serves as a rapid prototyping tool for Rust projects, and a way to simplify the distribution of Python projects as a single executable.

2 Background Review

2.1 Prior Art

This section focuses on other Python compilers, and how they differ from Rattlesnake.

2.1.1 Pyrex

Pyrex[2] is a Python-like language which compiles to C as a Python extension module. It was developed by Greg Ewing, and was the predecessor to Cython. It is no longer maintained, and has been superseded by Cython.

It only supported extension module targets, and as a result always depends on the CPython interpreter, rather than producing stand-alone executables.

2.1.2 Cython/Cythonize

Cython[3] and Cythonize, first released in 2007, are tools which translate Python or Cython code down to C, which can then be compiled either into a native executable or into a Python extension module. Cython is a modified dialect of Python, which changes some syntax to allow for easier translation for the tool.

While Cython and Cythonize share the same high-level goal as Rattlesnake - to make Python code faster - Cython produces code which is 100% equivalent to how CPython would run that code. This has a couple of downsides:

- Cython-generated code depends on Python's C API, which requires the Python headers to be installed, and in some cases requires a Python interpreter to be present at runtime.
- Cython-generated code contains lots of CPython calls, which can be difficult to read; this is made worse by the fact that C has no operator overloading, and so most operations require a function call.
- Because Cython produces 100% equivalent code, it is in many cases not possible to optimise the generated code for speed without changing the source Python code.

Rattlesnake, on the other hand, will produce Rust code, which is not guaranteed to be 100% equivalent to how CPython would run the code (although its behaviour will be matched on a best-effort basis). This allows for more aggressive optimisations, and means that the generated code will never require a Python interpreter to be present at runtime.

For more information on Cython, see:

[2]Ewing, G. (2010) 'Pyrex - a Language for Writing Python Extension Modules', *Pyrex*. Available at: https://www.csse.canterbury.ac.nz/greg.ewing/python/Pyrex/ (Accessed: 11 October 2023).

[3]Behnel S. *et al.* (2023) 'Cython: C-Extensions for Python'. Available at: https://cython.org/ (Accessed: 11 October 2023).

- https://cython.readthedocs.io/en/latest/
- https://pypi.org/project/Cython/
- https://github.com/cython/cython

2.1.3 RPython

RPython[4] is an AOT-compiled (*Ahead Of Time*-compiled; the traditional compilation model where all transformations occur prior to running the code) dialect of Python. It was developed for use in PyPy[5], and is used to write the PyPy interpreter itself. It has no formal specification (its official definition[6] is 'RPython is everything that our translation toolchain can accept'!).

RPython has a number of compile targets, split into the two broad categories of 'C-like memory model' and 'object-oriented memory model'. This allows the compiler to make code tailored for its target, leaning more heavily on allocation and caching for C-like targets, and translating somewhat more directly for OO targets.

2.2 Related Work

2.2.1 PyO3

PyO3[7] is a project which provides Rust bindings for CPython. This is typically used for creating extension modules, but can also be used to run Python code from Rust at run-time.

PyO3's goals are fundamentally different to Rattlesnake's, but as it is in the same area, it is worth mentioning.

2.2.2 RustPython

RustPython[8] is a Python interpreter written in Rust. It is not a compiler, but rather an alternative to CPython; its major selling point is the ability to embed a Python script into a web page, and run it in the browser with WebAssembly. It can also be used to embed Python scripts into Rust programs, similar to PyO3.

2.2.3 PyPy

*Not to be confused with **PyPI**, the Python Package Index.*

[4]The PyPy Project (2022) 'RPython Language'. Available at: https://rpython.readthedocs.io/en/latest/rpython.html (Accessed: 11 October 2023)

[5]The PyPy Team (2023) 'PyPy'. Available at: https://www.pypy.org/ (Accessed: 11 October 2023)

[6]The PyPy Project (2022) 'RPython Language'. Available at: https://rpython.readthedocs.io/en/latest/rpython.html (Accessed: 11 October 2023)

[7]PyO3 Developers (2023) *PyO3: Rust bindings for the Python interpreter*. Available at: https://github.com/PyO3/pyo3 (Accessed: 18/10/2023)

[8]windelbouwman *et al.* (2023) 'RustPython'. Available at: https://rustpython.github.io/ (Accessed: 18 October 2023)

PyPy[9] is a JIT-compiling (*Just In Time*-compiling; a hybrid compiled/interpreted approach where 'hot' code which is run many times gets compiled/optimised more heavily) Python interpreter, which is written in RPython, a restricted subset of Python which was developed alongside PyPy. PyPy's primary goal is to be a drop-in alternative to CPython, and in most cases is significantly faster at running pure-Python code.

2.3 Literature

2.3.1 Cython: The Best of Both Worlds[10]

This paper describes the advantages of Python for scientific computing, and presents its problems with performance when performing low-level loops as one of the justifications for Cython. It then describes Cython, first noting that it is a fork of Pyrex, and then describing its features.

Next, it goes on to describe a few common optimisations that Cython will perform by default, and then describes ways to further optimise Cython code using its special syntax.

2.3.2 Transpiling Python to Rust for Optimized Performance[11]

This paper describes the attempt of these researchers to create a Python-to-Rust semi-automatic transpiler, based on the (now-archived) open-source project Pyrs, which was, according to its README, 'not aimed at producing ready-to-compile code'.

The paper notes the similarities between Python idioms and Rust idioms that Rattlesnake aims to use, and notes an across-the-board speed-up and memory reduction of up to 12x and up to 4x, respectively, when comparing their compiled Rust code to the original Python code.

The researchers' method required lots of programmer input during the translation step; according to the paper, '[after] syntax conversion, the program is unlikely to immediately compile using the Rust compiler and must be manually edited'.

[9]The PyPy Team (2023) 'PyPy'. Available at: https://www.pypy.org/ (Accessed: 11 October 2023)

[10]Behnel, S. *et al.* (2011) 'Cython: The Best of Both Worlds', *Computing in Science & Engineering*, 13(2), pp. 31-39. Available at: https://doi.org/10.1109/MCSE.2010.118.

[11]Lunnikivi, H., Jylkkä, K. and Hämäläinen, T. (2020) 'Transpiling Python to Rust for Optimized Performance', in A. Orailoglu, M. Jung, and M. Reichenbach (eds) *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cham: Springer International Publishing, pp. 127-138.

2.3.3 Rust: The Programming Language for Safety and Performance[12]

This paper describes, generally, the Rust language itself, its history, and its advantages compared to other languages. Among other things, the paper notes Rust's memory safety, commitment to zero-overhead abstractions, and its ecosystem.

It goes on to compare Rust to C, C++, Go, Java, and Python in 3 benchmarks, and notes that Rust outperforms all of them except C in all three benchmarks for memory usage, and for speed outperforms all other languages in two of the three benchmarks, and is beaten only by C and C++ (and only by a margin of 0.04s) in the third.

The paper continues by discussing common memory safety issues, why they are common in C and C++ projects, and how Rust protects against them.

Overall, the paper gives a good description of Rust, and explains in details the reasons why Rust is a sensible choice for a low-level language - and hence, why it is a good choice as the target language for Rattlesnake.

3 Methodology

3.1 Approach

3.1.1 Software Development Model

The software development model I have chosen for Rattlesnake is the RAD (*Rapid Application Development*) model of development.

RAD is a methodology which focuses on rapid prototyping and iteration. It is a form of agile development, and is particularly suited to projects where the requirements are not fully known at the start of the project, or may change as the project progresses.

I have chosen RAD primarily because it reduces the overhead of other methodologies, such as the mountain of documentation associated with scrum, or the strict requirements of waterfall. It also allows for a more flexible timescale, as each iteration can be as long or as short as necessary.

As I am the primary stakeholder for the project at this point, the extra stakeholder involvement of RAD is not an issue.

3.1.2 Requirement Gathering Method

The requirement gathering method I have chosen for Rattlesnake is the prototyping method. This is a method which focuses primarily on creating candidate

[12]Bugden, W. and Alahmar, A. (2022) 'Rust: The Programming Language for Safety and Performance'. Available at: https://arxiv.org/abs/2206.05503.

improvements to the solution, and then evaluating them with the stakeholders (in this case, me) to see what works well, and what needs improving or replacing.

This method of requirement gathering can be thought of similarly to the gradient descent model used to train machine learning algorithms; trying out potential changes, and deciding the direction to move based on the feedback generated from those changes.

3.1.3 Testing and Evaluation Process

For the testing process, I intend to run the compiler over CPython's benchmarking suite; the test results will be based on how much of the suite compiles successfully (both with Rattlesnake, and then with Cargo).

Code which does not compile with Rattlesnake will be penalised less harshly than compiled code which cannot be compiled with Cargo; the former implies unsupported features whereas the latter implies a compiler bug.

For the evaluation process, the compiled artefacts will then be benchmarked and compared to the CPython and PyPy benchmark results.

3.2 Technology

The run-time libraries for Rattlesnake will be implemented in Python and Rust, by necessity. The first version of the compiler will be implemented in Python, and eventually it will be compiled by itself into Rust, then maintained as a Rust project. While the compiler is written in Python, it will use the `ast` module to parse source code; once it has been compiled into Rust, it is likely that I will swap out the `ast` module for `pest`, rather than simply maintaining the compiled version of `ast`.

3.3 Version Management Plan

Rattlesnake, and all associated documentation, will be versioned using Git. The repository will be hosted on GitHub, and will be publicly available once my dissertation is finished.

4 Project Management

4.1 Activities

4.1.1 Run-time Library (Rust)

Contributes to objective #1, #5.

The run-time library for Rust needs to contain basic definitions for Python objects and functions. It will also need to contain a special type, the boxed `PyObject`, which will be used for Python objects of unknown type.

4.1.2 Run-time Library (Python)

Contributes to objective #2, #5.

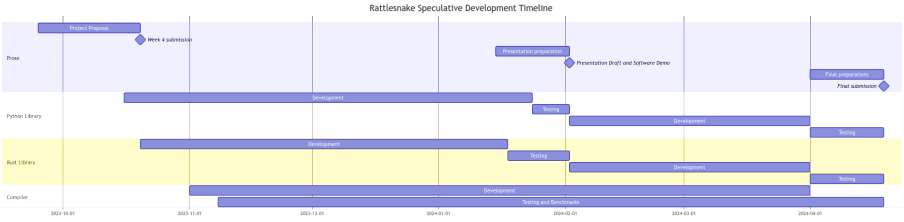
- Basic type annotations to allow optimisations of numeric types
- Compiler intrinsics in the form of marker annotations (e.g. `@no_except` to indicate that a function cannot raise an exception, or `@derive()` to indicate that a class wants to derive standard Rust traits such as `Debug`, `Clone`, or `Copy`)
  - These will be poly-filled at run-time when using a standard Python implementation
- Support for built-in macros
  - `println!()`
  - `panic!()`
  - `unreachable!()`
  - These will be implemented in Python as functions with the magic name `MACRO_*`, and any call to a function with a name starting with `MACRO_` will be transformed into a macro call in the output Rust code. They will also be poly-filled at run-time when using a standard Python implementation.
- Eventually, fill in types from the Rust standard library
  - `Vec<T>`
  - `HashMap<K, V>` and `BTreeMap<K, V>`
  - `HashSet<T>` and `BTreeSet<T>`
  - `Instant` and `Duration`
  - etc.
  - These types will be poly-filled at run-time when using a standard Python implementation.

4.1.3 Compiler

Contributes to objective #3, #4, #5.

- Parse and transform simple Python code into Rust code
- Add support for imported modules, and walk/tree-shake the dependency tree when compiling
- Add support for type annotations
  - Add support for generics
- Add support for compiler intrinsics via marker annotations in the Python library
- (Incrementally) add support for more language constructs and features

4.2 Schedule



4.3 Data Management Plan

The Rattlesnake repository will be structured as follows:

- `compiler`: The Rattlesnake compiler code, and the Rust run-time library.
  - The reason that these are both bundled into the same folder is so that once Rattlesnake has been published to crates.io
- `pylib`: The Python run-time library.
- `prose`: All prose documents, including this one, associated with the dissertation project, as Markdown documents.
- `meeting-notes`: Notes from meetings with my project supervisor, as Markdown documents.

4.4 Deliverables

The deliverables associated with the Rattlesnake project are as follows:

- The compiler itself
- The Rust run-time library, used by the compiler's output (and by the compiler itself, due to its initial Rust versions being compiled from Python by itself)
- The Python run-time library, used by the compiler's input to provide access to (poly-filled) native types and macros, as well as markers used by the compiler to optimise the output

5 Glossary

- AOT (adj.): Ahead-of-Time
  - A compilation model where all transformations occur prior to running the code. This is the traditional compilation model, used by languages such as C or Rust, and is the alternative to JIT.
- AST (n., acr.): Abstract Syntax Tree
  - A representation of source code which is easy to manipulate. It is often used as an intermediate representation of code in compilers as it represents the ideas written in the code without the complicating details such as whitespace.

For example, the following Python code:

```
def foo(bar):
    return bar + 1
```

Could be represented with the following AST (in Lisp syntax):

```
(function-definition
    (name "foo")
    (parameters
        (name "bar"))
    (body
        (return-statement
            (+ (name "bar")
                (constant 1)))))
```

- Cargo, `cargo` (n.): The standard Rust package manager and build tool
  - Cargo is the standard tool used to generate, build, and test Rust projects. It is also used to manage dependencies, and is the primary way of distributing Rust projects.
- Compiler (n.): A program which takes as input code in one form, and returns as output code in another form.
  - Typically, this is translating code from a high-level language into a lower-level language - such as transforming C or Rust into machine code - but it can also be translating code between two high-level languages - such as transforming TypeScript into JavaScript, or (in this case) transforming Python into Rust.
  - See also: transpiler.
- CPython (n.): The official Python interpreter, written in C
- crates.io (n.): The official Rust package registry
  - https://crates.io/
  - Rust projects are distributed as crates, which can published to crates.io. The `cargo` tool can then be used to install these crates, either globally (installing their associated binaries) or locally (as a dependency of the current project).
  - See also: docs.rs, PyPI.
- docs.rs (n.): The official Rust documentation host
  - https://docs.rs/
  - All crates published to crates.io have their documentation built automatically, and hosted on docs.rs. This allows easy access to documentation for Rust projects, without having to install the project locally, or needing to visit arbitrary third-party websites. Additionally, Rust's standard library documentation is also hosted on docs.rs, meaning that it is the single point of reference for all Rust documentation.
  - See also: crates.io.
- JIT (adj.): Just-in-Time
  - A compilation model where transformations and optimisations occur during the running of the code. Popularised by Self and Java, this is the compilation model used by most modern JavaScript engines, and is the alternative to AOT.

- Monkey-patch (n.): The result of monkey-patching
- Monkey-patch (v.): To modify a global namespace, typically to create additional global functions or data
  - This is generally considered bad practice in most languages, as doing so can lead to conflicts between different libraries which both monkey-patch the same namespace.
  - It is often used in languages like JavaScript, where poly-fills use monkey-patching to (conditionally) update the global namespace to add support for newer features.
- MSRV (n., acr.): Minimum Supported Rust Version
  - The oldest version of the Rust compiler supported by a project
- Poly-fill (n.): Code which poly-fills a specific feature
- Poly-fill (v.): To provide an implementation of a feature which is not natively supported by (this version of) the platform
  - The most common occurrence of this is in web development, where poly-fills can be used to provide support for newer features and APIs when running in older browsers.
  - Poly-fills refer specifically to monkey-patching the namespace being poly-filled, rather than providing a separate namespace for the feature, which will be implemented either by delegating to the native implementation (if available) or by providing a custom implementation (if not) - this alternative method is known as a *pony-fill*.
- PyPI (n., abbr.): Python Package Index
  - https://pypi.org/
  - The official Python package registry. It is used by the `pip` tool, and all related tools, to install Python packages.
  - See also: crates.io.
- RAD (n., acr.): Rapid Application Development
  - A software development methodology which focuses on rapid prototyping and iteration.
- Transpiler (n.): A term to refer to a compiler which translates specifically from a high-level language to another high-level language
  - See also: compiler.

6 Bibliography

Behnel, S. *et al.* (2011) 'Cython: The Best of Both Worlds', *Computing in Science & Engineering*, 13(2), pp. 31–39. Available at: https://doi.org/10.1109/MCSE.2010.118.

Behnel, S. *et al.* (2023) 'Cython: C-Extensions for Python'. Available at: https://cython.org/ (Accessed: 11 October 2023).

Bugden, W. and Alahmar, A. (2022) 'Rust: The Programming Language for Safety and Performance'. Available at: https://arxiv.org/abs/2206.05503.

dragostis *et al.* (2023) 'pest. The Elegant Parser'. Available at: https://pest.rs/ (Accessed: 18 October 2023).

Ewing, G. (2010) 'Pyrex - a Language for Writing Python Extension Modules', *Pyrex*. Available at: https://www.csse.canterbury.ac.nz/greg.ewing/python/Pyrex/ (Accessed: 11 October 2023).

Lunnikivi, H., Jylkkä, K. and Hämäläinen, T. (2020) 'Transpiling Python to Rust for Optimized Performance', in A. Orailoglu, M. Jung, and M. Reichenbach (eds) *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cham: Springer International Publishing, pp. 127–138.

PyO3 Developers (2023) 'PyO3: Rust bindings for the Python interpreter', *GitHub*. Available at: https://github.com/PyO3/pyo3 (Accessed: 18 October 2023).

The PyPy Project (2022) 'RPython Language'. Available at: https://rpython.readthedocs.io/en/latestrpython.html/ (Accessed: 11 October 2023).

The PyPy Team (2023) 'PyPy'. Available at: https://www.pypy.org/ (Accessed: 11 October 2023).

windelbouwman *et al.* (2023) 'RustPython'. Available at: https://rustpython.github.io/ (Accessed: 18 October 2023).