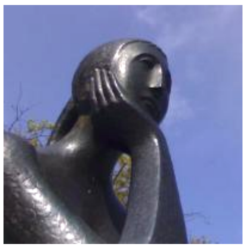


CUNY PONDER

The **P**r**O**gramming la**N**guages an**D** software **E**ngineering **R**esearch Lab

CUNY PONDER, situated at CUNY Hunter College, works at the intersection of **software engineering**, **programming languages**, and (**Machine Learning**) **systems**.Our lab **develops** and **assesses** techniques for automating critical software engineering tasks, including (static and dynamic) **program analysis**, **software evolution and maintenance**, and **empirical software engineering**. Our work results in algorithms and associated tools that analyze and transform (e.g., refactor) large and complex programs for improved **modularity**, **comprehension**, **maintainability**, **safety**, **security**, and **performance**. The techniques span multiple subfields of theory and application, such as **programming languages**, **type theory**, (front-end) **compilers**, **data science**, **informational retrieval**, and **mathematical logic**. We collaborate with the DAIR lab.

Professor

Raffi Khatchadourian

We are funded by the National Science Foundation (**NSF**), the Japan Society for the Promotion of Science (**JSPS**), Amazon Web Services (**AWS**), the **Verizon Foundation**, and the CUNY Diversity Projects Development Fund (**DPDF**). Our graduates have obtained research positions at prestigious universities and work at Google, New York Times, Squarespace, TD Ameritrade, New York Foundling, J.P. Morgan, and AD/FIN. Our publications have appeared in **top software engineering** and **programming language conferences** and have won **distinguished paper awards**.

Get in touch at ponder@hunter.cuny.edu **and find out more at** http://ponder-lab.github.io.

Students

Tatiana Castro Vélez

Yiming Tang

Ye Paing

Allan Spektor

Oren Friedman

David Morant

Walter Rada

Olivia Moore

Md. Arefin

Sample Projects

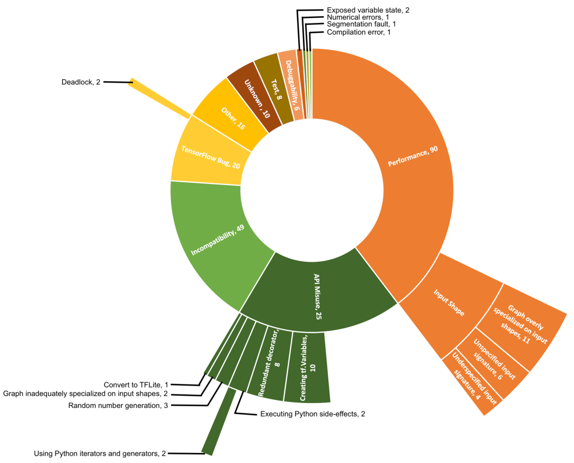
```
            if not hasattr(self, '_headers_b
                self._headers_buffer = []
            self._headers_buffer.append(("%s
                    (self.protocol_version,
                        'latin-1', 'strict')

    def send_header(self, keyword, value):
        """Send a MIME header to the headers
        if self.request_version != 'HTTP/0.9
            if not hasattr(self, '_headers_b
                self._headers_buffer = []
            self._headers_buffer.append(
                ("%s: %s\r\n" % (keyword, va

        if keyword.lower() == 'connection':
            if value.lower() == 'close':
                self.close_connection = Tru
            elif value.lower() == 'keep-aliv
                self.close_connection = Fals
```

Migrating Imperative Deep Learning Programs to Graph Execution

Efficiency is essential to support responsiveness w.r.t. ever-growing datasets, especially for Deep Learning (DL) systems. DL frameworks have traditionally embraced deferred execution-style DL code that supports symbolic, graph-based Deep Neural Network (DNN) computation. While hybrid approaches aim for the "best of both worlds," the challenges in applying them in the real world are largely unknown. We conduct a data-driven analysis of challenges—and resultant bugs—involved in writing reliable yet performant imperative DL code by studying 250 open-source projects, consisting of 19.7 MLOC, along with 470 and 446 manually examined code patches and bug reports, respectively. The results indicate that hybridization: (i) is prone to API misuse, (ii) can result in performance degradation—the opposite of its intention, and (iii) has limited application due to execution mode incompatibility. We put forth several recommendations, best practices, and anti-patterns for effectively hybridizing imperative DL code, potentially benefiting DL practitioners, API designers, tool developers, and educators.

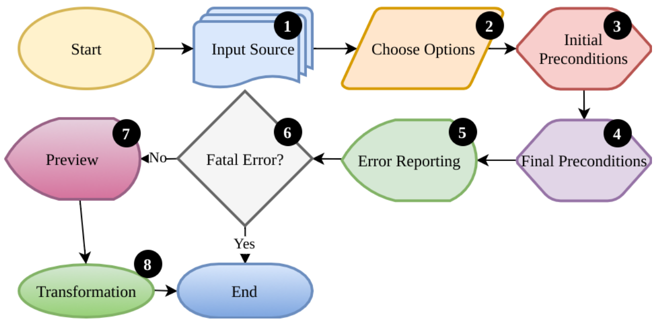


Refactorings and Technical Debt in Machine Learning Systems

Machine Learning (ML), including Deep Learning (DL), systems, are pervasive in today's data-driven society. Such systems are complex; they are comprised of ML models and many subsystems that support learning processes. Unfortunately, there is a gap in knowledge about how ML systems actually evolve and are maintained. In this project, we fill this gap by studying refactorings, i.e., source-to-source semantics-preserving program transformations, performed in real-world, open-source software, and the technical debt issues they alleviate. We analyzed 26 projects, consisting of 4.2 MLOC, along with 327 manually examined code patches. The results indicate that developers refactor these systems for a variety of reasons, both specific and tangential to ML, some refactorings correspond to established technical debt categories, while others do not, and code duplication is a major cross-cutting theme that particularly involved ML configuration and model code, which was also the most refactored. The results can potentially assist practitioners, tool developers, and educators in facilitating long-term ML system usefulness.

Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams

The Java 8 Stream API sets forth a promising new programming model that incorporates functional-like, MapReduce-style features into a mainstream programming language. However, using streams efficiently may involve subtle considerations. For example, although streams enable developers to run their code in parallel with little alteration, it is often not obvious if such code runs more efficiently this way. In fact, under certain conditions, running stream code in parallel can be less efficient than running it sequentially. Moreover, it can be unclear if running sequential stream code in parallel is safe and interference-free due to possible lambda expression side effects. This project involves an automated refactoring approach that assists developers in writing optimal stream client code in a semantics-preserving fashion. Based on a novel data ordering and typestate analysis, the approach consists of refactorings that include preconditions and transformations for automatically determining when it is safe and possibly advantageous to convert a sequential stream to parallel and improve upon already parallel streams. The approach is implemented as a plug-in to the popular Eclipse IDE utilizing both WALA and SAFE.


Start
Input Source 1
Choose Options 2
Initial Preconditions 3
Final Preconditions 4
Error Reporting 5
Fatal Error? 6
No
Yes
Preview 7
Transformation 8
End