Partial Evaluation for Faster Grammar Fuzzers

2022-23 Summer Vacation Research Internship Program

Kang Yang Chen

Supervisor:
Dr. Rahul Gopinath

The University of Sydney
School of Computer Science

February 2023

Abstract

Fuzzers are one of the primary tools used for evaluating program robustness. Traditional fuzzers can be ineffective when faced with programs that have complex and structured input requirements. Grammar fuzzers improve the effectiveness of traditional fuzzing by using a grammar specification such that it can get past the initial validator. A problem with such fuzzers is that incorporating the specification constraints into input generation can reduce the efficiency of the fuzzer. This thesis explores using partial evaluators to remove this inefficiency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While writing and compiling grammar fuzzer manually can make a grammar fuzzer faster, it might be challenging, especially with complex fuzzer. Any algorithm changes will also requires potentially large changes to the manual implementation. This process can potentially be improved by utilising partial evaluators to carry out Futamura Projections [1].

The aim of this project is to investigate the feasibility of improving fuzzer performance with minimal effort using frameworks such as pypy and Graal (Truffle). The author examined Graal and pypy and ultimately chose to experiment with pypy because of the lower entry barrier and better documentation. First Futamura Projection has been applied to the fuzzer to produce a grammar specific, specialised fuzzer program.

Initial result shows that manually generating a fuzzer in C based on a grammar still performs better compared to generating a fuzzer with the partial evaluator of pypy. However, for a more complex fuzzer, the ease and flexibility of producing a fuzzer based on a grammar using partial evaluator might outweight the potential performance gain of a manual implementation.

The source codes for this project can be found on GitHub.

Chapter 2

Futamura Projection

The equations below are based on the definition described in Williams and Perugini [2]. A copy of it is shown in the table below for convenience.

Symbol	Description
program	A miscellaneous program.
p_n	A parameter.
a_n	An argument.
$\text{compiler}_T^{S \to T}$	A compiler from language S to language T. Implemented in T.
program_L	A miscellaneous program implemented in language L.
interpreter_T^S	An interpreter for language S implemented in language T
$\textit{partial input}_{static}$	A subset of input for a program being specialised by mix.
$\text{program}_T^{'}$	A specialised program implemented in language T.
mix_T	A partial evaluator implemented in language T.
$\text{compiler generator}_T$	A compiler generator implemented in language T.

Table 2.1: Definition for equations [2].

Program execution are represented as $[[program]][a_1, a_2, \ldots, a_n] = [output]$ [3].

2.1 First Futamura Projection

First Futamura projection can produce **optimised compiled code**, which is derived from the interpreter using partial evaluation [4]. The partial evaluator specialised the interpreter to a program [2].

Based on Williams and Perugini [2], it can be represented as:

$$[[mix_T]][interpreter_T^S, program_S] = [program_T]$$

where,

$$[[program_T]][a_1, a_2, \ldots, a_n] = [output]$$

2.2 Second Futamura Projection

Second Futamura projection can produce a **compiler**, which is derived from partially evaluating an instance of partial evaluation (partial-partial evaluation pattern) where the interpreter is a static input [2].

Based on Williams and Perugini [2], it can be represented as:

$$[[mix_T]][mix_T, interpreter_T^S] = [compiler_T^{S \to T}]$$

where,

$$[[compiler_T^{S \to T}]][program_S] = [program_T]$$

2.3 Third Futamura Projection

Third Futamura projection can produce a **compiler generator**, which is derived from using one instance of partial evaluator to partially evaluate a second instance of partial evaluator, where a third instance of partial evaluator is a static input [2].

Based on Williams and Perugini [2], it can be represented as:

$$[[mix_T]][mix_T, mix_T] = [compiler\ generator_T]$$

where,

$$[[compiler\ generator_T]][interpreter_T^S] = [compiler_T^{S \to T}]$$

Chapter 3

Truffle by GraalVM

A quick overview of the components, extracted from GraalVM's documentation:

- **GraalVM** is a high-performance JDK, It offers 2 ways to run Java applications, with Graal just-in-time (JIT) compiler on the HotSpot JVM, or as native executable compiled ahead-of-time (AOT) [5]. It also provides runtime for a few other languages, such as JavaScript, Ruby and Python [5].
  - It contains a compiler written in Java, the GraalVM Compiler (source code). The compiler takes JVM machine bytecode as input and outputs machine code.
- **Truffle** is a library for building tools and programming languages implementations as interpreters for self-modifying Abstract Syntax Tree (AST) [6].

Graal compiles with partial evaluation to perform First Futamura Projection [2].

AST (regular or annotated) is the input for Truffle, therefore, you need to write an interpreter to produce the AST [7]. ANTLR4, a parser generator, is a popular choice in Truffle projects to generate a parser based on a provided grammar definition. The output of the generated parser is an AST, which can be used as Truffle's input. Figure 3.1 shows the input and output of each stage of a language compiler, based on the description provided by Ruka [7].

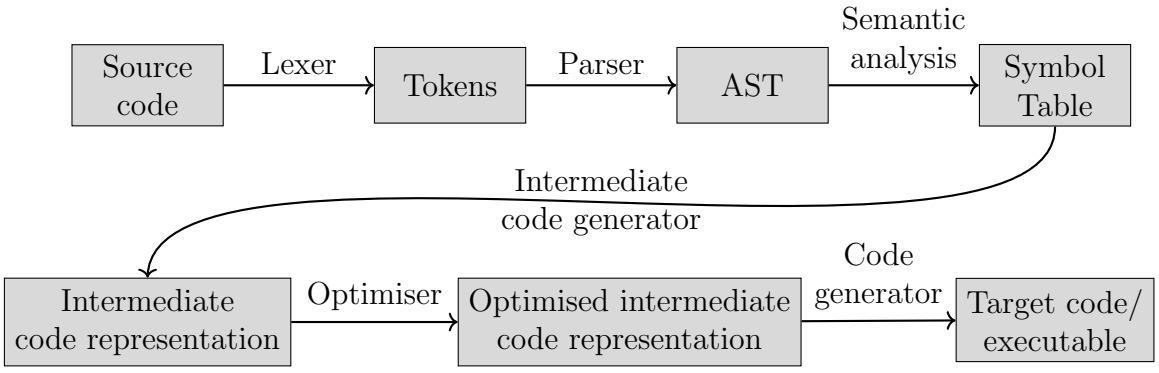

Figure 3.1: Typical language compiler pipeline

While the official GraalVM documentation recommends that you start with modifying the SimpleLanguage example [6], I personally find it hard to follow for a complete beginner. This blog post series by Adam Ruka (link) is much more beginner friendly and a great resources to get started.

Chapter 4

RPython (by the PyPy project)

- **PyPy** is a Python interpreter, written in RPython [8].
- **RPython** is the programming language accepted by the RPython translation toolchain. RPython is a subset of Python2 that can be statically compiled [9].
- The **RPython translation toolchain** translate RPython program into C, and in the process, applies some optimisation and includes memory management codes [10]. The toolchain never sees the RPython source code or syntax trees [11]. It takes code objects as inputs, which defines the behaviour of the function objects [11], or as *Goals and Architecture Overview - RPython Documentation* [11] states: "It can be considered as "freezing" a pre-imported RPython program into an executable form suitable for the target platform".

Listing 4.1 shows the code required for the RPython translation toolchain to work. The `target(*args)` and `entry_point(argv)` function is required while `if "__main__" == __name__` is for running the code on a Python interpreter. The most important thing to remember when coding in RPython is not to mix types. Additional restrictions imposed on RPython can be found using this link.

Listing 4.1: RPython boilerplate code

```
def entry_point(argv):
    # RPython code here
    return 0

# The 'main' function for a RPython program.
# Returns the entry point function.
def target(*args):
    return entry_point

# For the code to work on other Python interpreter
if "__main__" == __name__:
    entry_point(sys.argv)
```

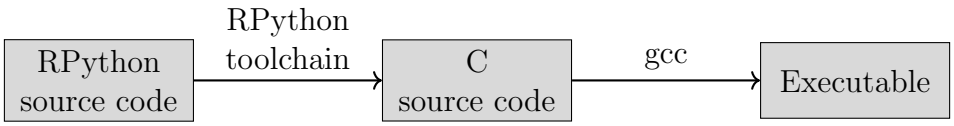

Figure 4.1: High level overview of producing executable from RPython

4.1 Installing and running RPython

4.1.1 Native

You will need a Python2 interpreter (for example, `pypy` or the CPython interpreter `python2`), Mercurial and also gcc.

1. Use Mercurial (not Git!) to clone the pypy repository.
   `hg clone 'https://foss.heptapod.net/pypy/pypy'`.
2. Set the `PYTHONPATH` environment variable to point to the directory you just cloned with `export PYTHONPATH=pypy`.
3. Run the translator with a Python2 interpreter (`pypy` in this example). `pypy ./pypy/rpython/bin/rpython main.py`, where `main.py` contains your RPython source code. It will produce an executable named `main-c` which you can execute.

4.1.2 Container

You can also translate and run your RPython program using the `Dockerfile` shown in Listing 4.2.

1. Build container: `podman build -t rpy ./`
2. Run container: `podman run -it --rm --name rpy rpy`

Listing 4.2: RPython Dockerfile

```
FROM docker.io/library/fedora:latest

RUN dnf update -y \
    && dnf install -y gcc python2 pypy mercurial

RUN useradd -m -U user

WORKDIR "/home/user"
USER user:user

RUN hg clone https://foss.heptapod.net/pypy/pypy
COPY --chown=user:user *.py ./
```

```
RUN PYTHONPATH=pypy pypy ./pypy/rpython/bin/rpython --output
    main main.py

ENTRYPOINT ["/home/user/main"]
```

4.2 JIT

PyPy uses a Tracing JIT compiler to trace frequently executed code. It caches the compiled version to bypass interpretation [2]. It can be applied to the interpreter written in RPython, not the user program that the interpreter is interpreting [12].

To use JIT in your interpreter, you need to provide hints to a JitDriver instance by defining the `greens` and `reds` variables [13]. Green variable does not change when executing a particular instruction [13], in other words, the variable does not change halfway in the interpretation loop. Red variables changes during execution of a particular instruction [13], which is the opposite behaviour of green variables. You will also need to call `jit_merge_point()` at the start of each iteration in the interpretation loop [13]. A `jitpolicy()` function also needs to be defined.

Chapter 5

`LimitFuzzer` and `LimitFuzzer_NR` in RPython

Both `LimitFuzzer` and `LimitFuzzer_NR` has undergone refactor to convert their original Python3 code to Python2 and subsequently to RPython.

Figure 5.1 show how First Futamura Projection is applied to both the fuzzers. Here are the definitions for the diagram:

- mix_{py2} is the specialiser (partial evaluator) in the RPython translation toolchain, which is implemented in Python2.
- $interpreter^{RPy(grammar)}_{py2(code\ object)}$ is the fuzzer. While it is implemented in RPython, because the RPython translation toolchain takes code object as input instead of the source code [11], the implementation language is defined as $py2(code\ object)$ here. The source language $RPy(grammar)$ is the grammar definition for the fuzzer in written in RPython.
- $program_{RPy(grammar)}$ is the grammar definition written in RPython for the fuzzer.
- $program_{py2(code\ object)}$ is the output of the partial evaluator, which is a program represented as code object in Python2.
- $program_C$ is the specialised C source code generated by the RPython translation toolchain.
- $program_{x86}$ is the specialised fuzzer executable produce by compiling the C source code with gcc.

The top part of the diagram all happens in the RPython translation toolchain, which produces the C source code ($program_C$). While the translator toolchain will automatically uses gcc to compiler the C source code, we will consider it as a separate process here that is external to the toolchain.

We can then run the specialised fuzzer with random bytes as input (provided via `stdin` with the `--stdin-rand` flag) to generate output specific to a grammar.

Equation: $[[program_{x86}]][argv, random\_bytes] = [output]$

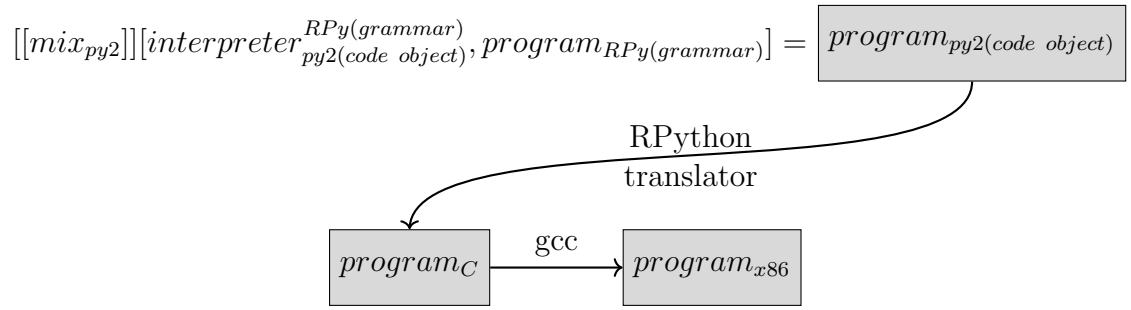

Figure 5.1: First Futamura Projection on `LimitFuzzer` and `LimitFuzzer_NR`

5.1 Benchmark

The results shown in this section is produced from a machine with the following specifications:

```
CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
  CPU(s):                  12
    On-line CPU(s) list:   0-11
  CPU max MHz:             4500.0000
  CPU min MHz:             800.0000
  Caches (sum of all):
    L1d:                   192 KiB (6 instances)
    L1i:                   192 KiB (6 instances)
    L2:                    1.5 MiB (6 instances)
    L3:                    12 MiB (1 instance)
Memory block size:         128M
Total online memory:       16G
```

For software information, see Appendix **??**.
For grammar used (extracted from F1), see Appendix **??**.

5.1.1 Methodology

`benchmark.sh` is the shell script used to run all the benchmarks, along with processing and plotting the results. To reduce run to run variance, the results shown in Figure 5.2 are the average of 5 runs. Each benchmark process were also pinned to a specific CPU core using `taskset`. All of the benchmarks were ran with a depth of 10 and generates 100000 JSON outputs.

There are 5 variants that were tested:

- `ori-f1-c-fuzzer`: The `CFuzzer` from the F1 repository, where C source code is generated based on the grammar provided and compiled with gcc using the `-O2` optimisation level to produce an executable. The `-O2` optimisation level is specified explicitly because it is the default optimisation level used by RPython when compiling its output with gcc [14].
- `limit-fuzzer-no-jit`: The recursive version of `LimitFuzzer` written in RPython. It has been translated and compiled into an executable.

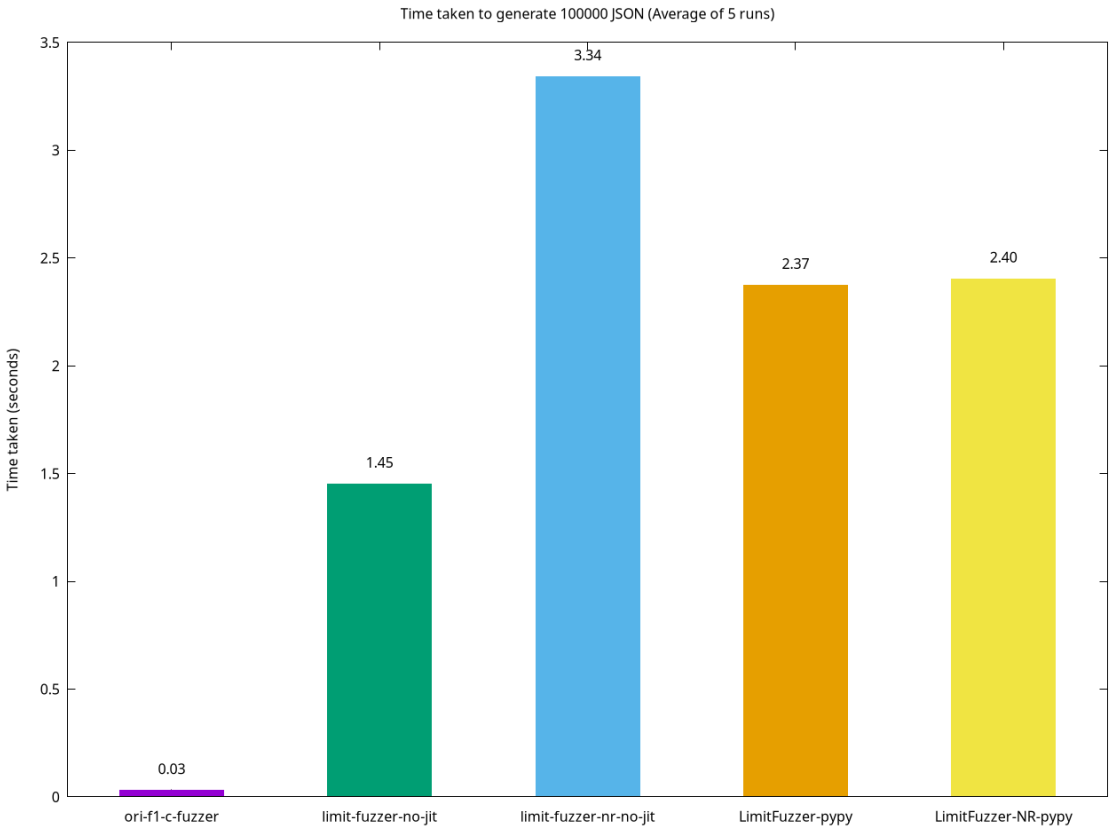

Figure 5.2: Benchmark result

- `limit-fuzzer-nr-no-jit`: The non recursive version of `LimitFuzzer`, `LimitFuzzer_NR`, written in RPython. It has been translated and compiled into an executable.
- `LimitFuzzer-pypy`: The recursive version of `LimitFuzzer` written in RPython. It is interpreted using `pypy`.
- `LimitFuzzer-NR-pypy`: The non recursive version of `LimitFuzzer`, `LimitFuzzer_NR`, written in RPython. It is interpreted using `pypy`.

5.1.2 Results

`ori-f1-c-fuzzer` is significantly faster than the remaining variant. This makes sense since the fuzzer generated by `CFuzzer` is significantly simpler than the C code generated by RPython. It also does not contain any memory management code, which contributes to its performance while the C code generated by RPython contains garbage collection code.

While the recursive variant of fuzzer generated by RPython (`limit-fuzzer-no-jit`) outperform the pypy interpreted variants, the non recursive variant generated by RPython performs worse than the pypy interpreted variants. The author speculates that it might be a memory management performance issue in RPython since the non recursive variant uses more complex data structures, such as a queue to schedule work.

5.1.3 Limitation

- The method of getting random numbers are inconsistent between `ori-f1-c-fuzzer` and the other variants. `ori-f1-c-fuzzer` takes a seed as input while others read random bytes from stdin. For the other variants, random bytes were saved to a file beforehand, generated using `dd bs=1048576 count=1 if=/dev/urandom of=./1048576B_rand_data` from `/dev/urandom`. This might hamper performance since disk I/O is significantly slower.
- Random bytes provided to all other fuzzer expect `ori-f1-c-fuzzer` needs to be sufficient (the author has verified that the fuzzers were provided with sufficient random bytes for the result above), otherwise, the implementation will always return 0 as the random number if it runs out of random bytes.

Chapter 6

Conclusion

Enabling JIT for the fuzzer in RPython could be something worth exploring in the future. The performance anomaly of `limit-fuzzer-no-jit` can also be explore further to determine the root cause.

While using the RPython toolchain to produce an executable can improve performance compared to using an interpreter, if the code is simple enough, manual implementation either through a source to source compiler or directly in a lower level language will yield better performance based on the result above.

Bibliography

(1) Y. Futamura, "Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler", *High. Order Symb. Comput.*, 1999, **12**, 381–391.

(2) B. M. Williams and S. Perugini, "Revisiting the Futamura Projections: A Diagrammatic Approach", *B.M. Williams & Perugini, S. (2016) Revisiting the Futamura Projections: A diagrammatic approach. Theoretical and Applied Informatics, 28(4), 15-32*, 2016, **28**, 15–32.

(3) N. D. Jones, C. K. Gomard and P. Sestoft, *Partial evaluation and automatic program generation*, Prentice Hall International, 1993.

(4) T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes", *Proceedings of the 38th ACM SIGPLAN Conference on programming language design and implementation*, 2017, **128414**, 662–676.

(5) *Get Started with GraalVM*, Oracle, n.d.

(6) *Truffle Language Implementation Framework*, Oracle, n.d.

(7) A. Ruka, *Graal Truffle tutorial part 1 – setup, Nodes, CallTarget*, Website, 2020.

(8) *What is PyPy? - PyPy documentation*, Accessed on 2023-02-02, The PyPy Project.

(9) *Getting Started with RPython - RPython Documentation*, Accessed on 2023-02-02, The PyPy Project, n.d.

(10) *The RPython Toolchain - RPython Documentation*, Accessed on 2023-02-02, The PyPy Project, n.d.

(11) *Goals and Architecture Overview - RPython Documentation*, Accessed on 2023-02-02, The PyPy Project, n.d.

(12) *Goals and Architecture Overview - PyPy documentation*, Accessed on 2023-02-02, The PyPy Project, n.d.

(13) A. Brown, *Tutorial Part 2: Adding a JIT*, Website, Accessed on: 2023-02-03, 2011.

(14) *The –opt or -O translation option - PyPy documentation*, Accessed on: 2023-02-03, The PyPy Project, n.d.