

# Towards Python for Serverless Acceleration

DYLAN FOX and ERIKA HUNHOFF, University of Colorado Boulder

Serverless computing offers many benefits for programmers, but frequently, slower interpreted languages like Python or JavaScript are used. In their 2019 work, Emily Herbert and Arjun Goha developed a serverless function accelerator that traces and compiles JavaScript to much more performant Rust code. Here, we extend their accelerator to trace and compile Python code as well as JavaScript, allowing more code bases to utilize the accelerator.

## ACM Reference Format:

Dylan Fox and Erika Hunhoff. 2020. Towards Python for Serverless Acceleration. *ACM Forthcoming* CSCI 5535, Spring 2020 (April 2020), 5 pages.

## 1 INTRODUCTION

In their 2019 work, Emily Herbert and Arjun Goha developed a serverless function accelerator. This accelerator traces JavaScript serverless functions at run-time, and produces an intermediate representation of the program. This intermediate representation is then compiled to Rust code which can be run in place of the JavaScript function. Here, we extend their work by providing a tracing mechanism for Python code. After we trace the python code, we compile it to Rust using their Rust compiler.

Transforming the Python code to A-Normal Form resolves differences in scoping between Python and Rust, and simplifies the tracing and compilation to Rust.

Our Python A-Normal Form Code follows three specifications:

- (1) All variables defined in functions or modules are initialized to None at the top of the function or module they are initially defined in.
- (2) All loops are while loops.
- (3) All function applications are named.

After A-Normalization, the resulting A-Normal Form Python code has tracing statements inserted into it. All expressions are transformed to be method calls to our tracing library. At run-time, the tracing library outputs the intermediate representation of the code being executed to a JSON file, then executes the Python code. After a number of invocations of the Python serverless function, the JSON file with the intermediate representation is compiled to Rust. Further invocations of the serverless function run the Rust code instead of the Python code.

## 2 OVERVIEW

In the serverless accelerator framework created by Herbert & Guha (hereby referred to as 'containerless'), a JavaScript function goes through several distinct phases:

- (1) Normalization
- (2) Addition of Tracing Statements
- (3) Tracing to produce IR
- (4) Compilation from IR to Rust

---

Authors' address: Dylan Fox, dylan.fox@colorado.edu; Erika Hunhoff, erika.hunhoff@colorado.edu, University of Colorado Boulder.

---

© 2020 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Forthcoming ACM Publication*.

- (5) Invoker uses Rust representation and Original JavaScript function to attempt to accelerate run time safely

The goal of this work is to create normalization and tracing implementations that can produce compatible IR files from serverless functions written in Python. Since our goal is to avoid any changes to the IR, we are limited in what Python features we can support by what is already supported in our IR. This is the first challenge: match the supported syntax in JavaScript to a subset of Python that we will support. Details of this process are found in Section 3.

The normalization phase in containerless uses A-Normal Form to take as input JavaScript functions (using the limited features supported) and output normalized JavaScript which is guaranteed to be traced completely during the tracing phase. To give a concrete example, if the following Python function were submitted to our framework:

```
def func():
    y = []
    x = [1, 2, 3]
    for index in range(len(x)):
        y[index] = x[index]

    if y[1] == 2:
        result = 'ok'
    else:
        print('Not Ok')
        result = 'not ok'

    return result
```

The output of the normalization would be:

```
def func():
    y = None
    x = None
    result = None

    loop_incrementor0 = None
    app0 = None

    y = []
    x = [1, 2, 3]

    loop_incrementor0 = 0
    while loop_incrementor0 < len(range(len(x))):
        index = range(len(x))[loop_incrementor]
        y[index] = x[index]
        loop_incrementor0 += 1

    if y[1] == 2:
        result = 'ok'
    else:
        app0 = print('Not Ok')
        result = 'not ok'
```

```
return result
```

(Author’s Note: These will be reformatted, probably to be side by side, in the final version of this document).

In addition to pursuing the same tracing guarantees in normalization, we also use this phase to address some of the syntactic differences between JavaScript and Python, so that our tracing can mimic the tracing used by containerless. For instance, in JavaScript variables can be declared (use of the ‘var’ keyword) and are forcibly declared during normalization which then helps determine variable scope during tracing. Since no analogous statement in Python exists, our normalization framework creates ‘var = None’ statements at the beginning of each function which serve a similar purpose as the declarations used in containerless normalization. More details on the normalization procedure is described in detail in 4.

The tracing phase takes as input a normalized function and outputs IR in JSON files. For instance, the previous example could output the following trace:

```
TODO: once we have tracing actually working, we will post example here
```

This is the last step of our development. One aspect that makes the tracing phase difficult is that JavaScript serverless functions and Python serverless functions have a slightly different workflow. JavaScript functions receive their input as a request object (and access this object by ‘listening’), and ‘return’ by calling asynchronous return functions. Python serverless functions, on the other hand, take as input a single JSON text argument, and return a single JSON output. Thus, although the mechanics of tracing were similar, the framework for running traces had to be reworked for this new format. Details on tracing are found in 5.

There are two main categories of evaluation for our work: performance and correctness. Since we focus not on the function invoker or system as a whole but on syntactic definitions, transformations, and tracing, we focus our evaluation on correctness. We measure correctness at all stages by checking whether normalized functions produce the same outputs as original functions, whether traces produced can be compiled into Rust without error, and whether the Rust executes with the same result as the original python. Details on evaluation are found in 6.

### 3 SUPPORTED PYTHON SYNTAX

TODO for HW06

<i>BinaryOp</i>	<i>op2</i>	::=	+	Addition
			−	Subtraction
			*	Multiplication
			/	Division
			==	Equals
			!=	Not equals
			<	Less than
			<=	Less than or equal to
			>	Greater than
			>=	Greater than or equal to
			<i>and</i>	And
			<i>or</i>	Or
<i>UnaryOp</i>	<i>op1</i>	::=	<i>not</i>	Logical negation
			−	Negate
<i>Exp</i>	<i>exp</i>	::=	<i>Name</i>	Variable
			<i>Exp</i> ( <i>&lt; Exp &gt; *</i> )	Function call
			<i>Exp.Name</i>	Attribute access
			<i>Exp</i> [ <i>Exp</i> ]	Slice access
			<i>ExpBinOpExp</i>	Binary operation
			<i>UnaryOpExp</i>	Unary operation
			<i>Int</i>	Integer
			<i>Bool</i>	Boolean
			<i>String</i>	String
			[ <i>&lt; Exp &gt; *</i> ]	List
			<i>&lt; Exp : Exp &gt; *</i>	Dictionary
			<i>Name = Exp</i>	Variable assignment
			<i>Exp</i> [ <i>Exp</i> ] = <i>Exp</i>	Slice assignment
			<i>Exp.Name</i> = <i>Exp</i>	Attribute assignment
			<i>defName</i> ( <i>&lt; Name &gt; *</i> ) : <i>Block</i>	Function definition
			<i>returnExp</i>	Return
			<i>ifExp</i> : <i>Block</i> <i>else</i> : <i>Block</i>	If-then-else
			<i>whileExp</i> : <i>Block</i>	While

## 4 A-NORMAL FORM TRANSFORMATION

TODO for HW06

## 5 TRACING

TODO for HW06

## 6 EVALUATION

TODO for HW06

## 7 RELATED WORK

Our work builds on the concepts and code presented in Herbert and Guha [2], which was build upon the principles presented in [4]. [2] focuses on the entire process of creating a JavaScript to IR framework, an IR to Rust compilation process, and a serverless function invoker compatible with

OpenWhisk [3]. Our work is more narrow, and seeks to modify the JavaScript to IR phase into a Python to IR process.

Work has been done towards formalizing a semantics for Python [5], and there are many tools and libraries for instrumenting Python code that this work builds upon (such as [1]). However, our work is specifically geared towards the subset of Python needed to write serverless functions. This is in contrast with existing work which focuses on either creation of general purpose tools or solutions for a particular case (other than serverless).

## 8 CONCLUSION

TODO by paper deadline

## ACKNOWLEDGMENTS

TBD

## REFERENCES

- [1] [n.d.]. Welcome to RPython’s documentation! <https://rpython.readthedocs.io/en/latest/index.html>
- [2] Emily Herbert and Arjun Guha. 2019. A Language-based Serverless Function Accelerator. arXiv:cs.DC/1911.02178
- [3] Baldini Ioana, Castro Paul, Cheng Perry, Fink Stephen, Ishakian Vatche, Mitchell Nick, Muthusamy Vinod, Rabbah Rodric, and Suter Philippe. 2016. Cloud-Native, Event-Based Programming for Mobile Applications.
- [4] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 149 (Oct. 2019), 26 pages. <https://doi.org/10.1145/3360575>
- [5] Gideon Joachim Smeding. [n.d.]. An executable operational semantics for Python. <http://gideon.smdng.nl/2009/01/an-executable-operational-semantics-for-python/>. Accessed: 2020-03-05.