# 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 Guha developer 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), 6 pages.

## 1 INTRODUCTION

In their 2019 work, Emily Herbert and Arjun Guha 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.

(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 = 0
    array_x = [1, 2, 3]
    for x in array_x:
        y += x

    if y == 6:
        result = 'ok'
    else:
        print('Not Ok')
        result = 'not ok'

    return result
```

The output of the normalization would be:

```
def func():
    y = None
    array_x = None
    x = None
    result = None
    loop_incrementor0 = None
    app0 = None

    y =  0
    x = [1, 2, 3]

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

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

```
    return result
```

(the above will be formatted in the final paper as per you suggestions Benno)

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

Our initial goals are to fit within the the framework of containerless, and as such, the subset of supported Python does not introduce any additional features beyond those already supported in the Javascript-to-IR representation, which includes some very useful features - such as while loops - but does not allow integration of some very pythonic features, such as 'for all' loops, the 'in' keyword, and even the modulo operator. All supported Python syntax is supported within the tracing framework, and is detailed below.

(TODO we are waiting for the final version before cleaning up formatting, but we will do so).

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

## 4 A-NORMAL FORM TRANSFORMATION

In Python, variables are scoped much differently than in Rust. Rust utilizes block scoping, meaning variables are scoped to the block of code they are declared in. If a variable is declared inside a loop, conditional, or other block of code, the variable cannot be accessed outside the block. On the other hand, Python only supports scoping at the function, class, and module levels. So, in Python, all variables defined in while loops, for loops, if/else statements, etc... are available at the nearest class, function, or module level. If directly translated to Rust, this scoping would result in errors if the programmer defined a variable inside a loop, conditional, or other sub-block and then tried to access it outside the block it was defined in. To preserve this functionality of Python, our ANF transformer creates a variable declaration for every variable at the top of the nearest module or function. The variable declaration simply sets the variable to None so that the variables scope is preserved. If variables are declared at a higher scope (eg. in a closure), we do not reinitialize the variable to None at the beginning of the lower scope, we only change the higher scope. We do not

support the use of object oriented programming here, so we don't normalize variables in classes, focusing on modules and functions instead

Our ANF-Transformer also converts all for loops to while loops. In Rust for loops are actually quite similar to Python for loops. Both languages use for loops that iterate over an iterator and do not support direct C style for loops. Both Rust and Python also have iterators that can be infinite. To avoid the possibility of infinite looping, for loops are converted to while loops, and we only support iterating over index-able finite collections. An example of a for loop transformation is found below:

```
for x in array_x:
    y += x
```

Is transformed to:

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

(the above will be formatted in the final paper as per you suggestions Benno) Here, we first wrap the iterator in a call to the builtin len function, to ascertain the length of the iterator. Then we instantiate a new variable to track the number of times the while loop has been run. The target variable is also moved inside the loop and set to to the iterator indexed to the current loop iteration. Finally, we increment the tracker variable inside the loop.

During ANF normalization, we also name unnamed function applications. This is primarily done to comply with the specs for the intermediate representation of Containerless and to keep the A-Normal Form code in compliance with common definitions of A-Normal Form Code. It should be noted that we do not provide a name to the applications of the builtin len function in Python. This is acceptable because there is no analogous built in function in Containerless' IR or in Rust. Rather, in Rust the array class contains a method to ascertain the length of the array. In Javascript and Containerless' IR, the length is a property of the array. When Python code is traced on our environment, the calls to len are replaced with a call to get the length property of the array, to conform with the Containerless IR. Therefore, applications of the len builtin function do not need to be named by the ANF normalizer.

## 5 TRACING

There are two distinct phases of tracing. First, the source code of a ANF normalized Python function must be transformed to contain calls (in addition to the original code) to the tracing infrastructure. Then the resulting Python function must be run, producing JSON traces that could be used to compile to Rust in Containerless.

To insert the tracing statements into a Python function, we first insert an "import" statement to import our tracing library. Next, for every statement, we import the correct tracing function. For example, a call to the traceSet function must proceed any python code that sets the value of a variable.

After the tracing library statements are inserted into the source code, we can run our function, and both execute the function, and produce a trace.

When JavaScript serverless functions are run in container-mode within the containerless framework, they run within the node.js environment. As a consequence of this, the JavaScript functions

run using the common asynchronous request/response format in web programming. Using OpenWhisk as an example - where JavaScript functions agan run using node.js but other languages run in other runtimes - we stubbed out our own Python runtime for containerless tracing that uses JSON dictionaries as the function parameter and function response. This tracing runtime allowed us to produce Python traces without utilizing a serverless invoker.

## 6 EVALUATION

Since this work did not encompass modifications to the invoker, and indeed, containerless does not run Python functions in any capacity, it was not possible to run performance tests using the containerless infrastructure. Thus, our evaluation focuses on correctness of the produced traces. Based on the limited number of Python functions supported, we constructed a series of test Python programs and pushed them through our system to produce traces. We then tested the original Python against the stripped traces to ensure they produced identical outputs with a variety of parameters engineering to guarantee full code coverage of the original test functions. A sample of an original Python function, the normalized functions, the annotated normalized function, the trace, and then the re-interpreted python code produced from the trace are shown below.

TODO: We will have this done by the time the project is due.

## 7 RELATED WORK

Our work builds on the concepts and code presented in Herbert and Guha [2], which was built 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.