Variadic Functions and Argument Expansion

Adding some syntactic sugar to our Python compiler

Nick Smith
University of Colorado Boulder
nism5806@colorado.edu

Akash Gaonkar
University of Colorado Boulder
akash.gaonkar@colorado.edu

Abstract

This paper covers an implementation of variadic functions and argument expansion for a Python to x86 compiler. Variadic functions and arguments expansion are defined and given an overview. Possible implementations to handle them are discussed, followed by the chosen implementation in detail. Next, there are a few expansions of the project considered. Last, the project is summarized and the results are reflected upon.

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

When defining functions it is common practice to assume you will have a fixed number of parameters. However, there are many common mathematical functions, for example sum and average, that can take any number of arguments. A function that takes a variable amount of arguments is called a variadic function. It is easy to manually generate the solution to a sum for any number of arguments, but programming a function to do the same is less trivial. When programming with variadics, it is also helpful to work in a language with argument expansion. If a variadic parameter is passed as an argument to a function, the function argument list is populated in order with the elements of the variadic. While it adds a good amount of work for the compiler, variadic functions paired with argument expansion allows for simple but powerful recursive functions. Here we have a variadic sum function that utilizes argument expansion for recursion:

```
def sum(n, *rest):
    return n + (sum(*rest)\
        if len(rest) > 0 else 0)
```

In the Python AST, the variadic variable is a Starred(Name) and is the varargs value of the Lambda object rather than a part of the args list.

http://dx.doi.org/10.1145/

2. Possible Implementations

2.1 C-Style: Format String

For C compilers, variadic functions are generally implemented by pushing an extra ”format string” on variadic function calls. Each variable is pushed as normal, and then a string containing information on each pushed argument is pushed. For example, if three integers are arguments to the function, the format string would be ”%d%d%d”. The variadic function itself would read the format string, and then read three integers from the stack. This method however poses a security risk as it is susceptible to Format String attacks. A user could pass the wrong format string and have memory be read incorrectly. This implementation did not fit for our compiler as it is very much meant for C, as C uses format strings. While it would be possible to make them work, we decided against this implementation.

2.2 Incremental

Another possible solution is to increment through the argument list. First, convert the arguments of a function call to a single list, expanding starred arguments. From this list, push an argument for each function parameter, and then push the rest of the list as the variadic parameter. This is the method we chose to use for our implementation. We decided it was a more straightforward algorithm than the others, even though it requires a lot of work from the compiler. This will be described in more detail in the section on our implementation.

2.3 Recursive

It would be nice to have a recursive function at run-time to properly add iterate through all the arguments, adding the first `n_params` to the stack, and then the rest to a varargs list. This would have save time as one isn't building a list of the arguments, but instead adding items directly. Moreover, it allows variadics to be coded with the simpler looping constructs of P2, rather than P3, where the while loop is added. Although we didn't choose this method, we figured out a possible way to implement it.

3. Our Implementation

In implementing our solution, we needed to be able to tell how many parameters a function has at run-time. We added a variable to function closures, `nparams`, to store the number of parameters that function has. Then we wrote a function in runtime.c, `get_nparams`, to return this value so that our implementation could refer to it. Our next task was to implement argument expansion of Starred arguments. We first implemented argument expansion on normal functions. On a function call at run time, the arguments would be turned into a list, with the elements of any Starred arguments retrieved and added as well. If this list has a different size

than the function to be called, the code aborts. Otherwise, the elements of the argument list are pushed and the function is called. To implement the variadic functions, we decided to treat all functions as variadic. Instead of checking if the argument list size is equal to the function parameter count, we check if it is greater or equal. The arguments are pushed as normal, and then all extra arguments are pushed in a single list. For variadic functions, this extra list is the variadic parameter, and for normal functions the list is ignored. As a side effect to this solution, all functions can be called with any number of arguments, but they will only handle the parameters they expect.

3.1 Syntax

For our project, we began with P2, adding the following expressions to our concrete grammar.

```
expression ::= expression(args_list)
    | expression "<" expression
    | expression "<=" expression
    | expression ">" expression
    | expression ">=" expression
args_list ::= ""
    | expression
    | *expression
    | expression "," args_list
    | *expression "," args_list
id_list ::= ""
    | identifier
    | *identifier
    | identifier "," id_list
statement ::= "while" expression ":" suite
```

The `*expressions` of the `args_list` allow argument expansion, while the `*identifier` of the `id_list` allows a single variadic parameter. A new reserved word `len` was introduced to reveal list size to the end user. It was connected to a new function, verb—pyobj list$_s ize(pyobjlst)|thatwasaddedtotheruntime$.

For our abstract syntax, we used Python's `ast` module for parsing (not `compiler.ast`, but `ast`). In addition to nodes added over the various labs, we added a node for the less than comparison, `CmpLt`, and placeholders for `Inc` and `Dec`, two operations we used very often. We also modifed our `Closure` node to carry the number of args its associated function accepted, and whether or not it was variadic.

3.2 Heapify

Heapify was adjusted slightly to also deal with the varargs variable in Lambdas. Everything that was done for regular lambda args must be done for the varargs variable.

3.3 Closure Conversion

In closure conversion we added the number of function parameters to the closure.

3.4 Flatten

The main changes to the code happened in Flatten. Flattening Calls was changed the most. In addition to the regular flattening, it must add the creation of the argument list, check the function parameter length, and pop the arguments to the function in a loop. Flattening Closures needed to change to add the number of function parameters to the closure pyobj at run-time.

3.5 Other Sections

In our code for removing control flow, we had to expand it to handle whiles. It was similar to handling ifs, but with a different flow. Whiles were replaced with compare and jump instructions. We also had to handle explication of while to properly explicate the while test.

4. Tests

To test our expanded grammar we had to write more Python tests.

```
def print_all(arg, *rest):
    print(arg)
    return print_all(*rest) if rest else 0
print_all(1, [1, 2, 3], 14, 42)


def list(*args):
    return args
i = 0
lst = list(1, 2, 3)
while i < len(lst):
    print(lst[i])
    i = i + 1


def list(*args):
    return lambda: args
lst = list(1, 3, 5)
i = 0
while i < len(lst()):
    print(lst()[i])
    i = i + 1


def sum(n, *rest):
    return n + (sum(*rest)\
        if len(rest) > 0 else 0)
print(sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))

print((lambda x, y, z: x + y + z)\
    (*[1, 2, 3]))


def sum(w, x, y, z):
    print(w + x + y + z)
sum(1, 2, 3, 4)
sum(1, 2, *[3, 4])
sum(1, *[2, 3], 4)
sum(*[1, 2], 3, 4)
sum(*[1, 2], *[3, 4])


def call_in_succession(*funcs):
    def func(arg, f, *rest):
        return funcs(f(arg), rest) if len(rest) > 0
            else arg
    return lambda arg: func(arg, funcs)
print(call_in_succession(
    lambda i: i + 1,
    lambda i: i + 2,
    lambda i: i + 3
)(1))


def get_list():
    return [1, 2, 3]
def print_all(x, *rest):
    print(x)
    return print_all(*rest) if len(rest) > 0 else 0
print_all(*get_list())

The above test fails
```

5. Next Steps

We of course could only spend so much time on this project, and there are many potential ways to expand on our code.

5.1 Add Back Regular Functions

A simple functional improvement on our code would be to add a way to distinguish variadic and regular functions, and then abort if a regular function is called with too many arguments.

5.2 Classes

Variadic behavior could be expanded to classes. In the same way that a function could be called with any number of arguments, so could a class.

5.3 Partial Functions

If we can pass too many arguments to a function, it follows that we should be able to pass too few arguments to a function. This could be done by attaching the arguments to the closure, until there are enough arguments to actually call the function

6. Conclusion

Variadic functions combined with argument expansion make it easy to write short recursive functions that do a lot. Unfortunately, this comes at a high cost to the compiler and the generated assembly. The assembly generated from handling argument expansion is extremely long as a new list is generated for the arguments, and then the elements from this new list are immediately accessed to be pushed. However, computers are quick and while our programs are not pushing the limits of computing, longer assembly only really makes for harder debugging. Once the assembly actually worked, it would still always run instantly, despite being much longer than before. The addition of variadics can greatly simplify the code written at some cost to the run-time.

References

[1] `https://stackoverflow.com/questions/23104628/technically-how-do-variadic-functions-work-how-does-printf-work`

[2] `https://www.gnu.org/savannah-checkouts/gnu/libc/manual/html_node/Variadic-Functions.html`