Modernized Python

Daniel Ng

March 25, 2022

Abstract

The popular Python programming language is generally heralded for its usability and learnability. However, this often comes at the cost of efficiency. Solutions to this problem have emerged over the last few years to impose static type systems onto Python without intruding on the underlying syntax. This paper deviates from the norm of building on the existing language, and instead looks to completely overhaul the concrete syntax while maintaining its beloved semantics.

Introduction

Python has recently surged in popularity, largely for two reasons. Firstly, it is an easy language for beginners to learn, especially because its array of modules allows for relatively complex projects to be built up relatively quickly. Additionally, these modules are often used in fields such as artificial intelligence and machine learning as a reliable and easy-to-use language. However, concerns from type theorists have emerged, largely regarding its inefficiency and unsafety.

The approach of overlaying a type system onto the existing concrete syntax is one that has been considered in several places previously. Type theorists claim that this could speed up the language and allow for fewer runtime errors. However, this approach has a negative impact in certain edge cases. Certain scripts may wait for Python programs to terminate, causing processor cycles to be wasted. Additionally, speeding up Python gives programmers less time to slack off while their code is running, the social drawbacks of which are immense.

It is thus only natural to look into ways to make the Python programming language more palatable for functional programmers. Current approaches focus on keeping Python's concrete syntax while overhauling the abstract syntax to better match that of functional languages. In contrast, we try a revolutionary new approach in this paper and try to improve the concrete syntax make it more palatable for functional programmers. In the spirit of Harper [1], we therefore propose a Modernized Python, which seeks to eliminate the issues

with Python at the level of concrete syntax, while maintaining both the speed and memory usage of original Python. Any resemblance to real languages, living or dead, is purely coincidental[1].

Problems

Concrete Syntax

While the concrete syntax of Python seems both easy to learn and easy to write, its actual nature is really not like that at all. Consider the behaviour of the following program:

```
1   def fact(x):
2      if (x == 0):
3         return 1
4     return x * fact(x - 1)
5   print(fact(10))
```

At first glance, this appears to be a perfectly legal[2] program that computes the factorial of a given positive number. However, upon actually attempting to interpret this code, it returns `IndentError` rather than printing `3628800`. Some might argue that this has to do with the use of a three-space indent for some awful reason. However, the same problem exists for users of the much more popular two-space indent. It follows from these strict indentation requirements that handwritten Python code on paper[3] is extremely unlikely to run, much to the dismay of many exam-takers.

The additional requirement that all lines be terminated with a newline adds further complications to the concrete syntax, especially when trying to write long commands and avoid confusion. The indentation issue is also exascerbated when determining where a loop ends or begins. Marking the end of code blocks in general is therefore another concern that we will attempt to address when modernizing Python.

One feature that must be preserved through this is Python's acceptance of generally questionable code. For example, the following function would be rejected by most sensible type systems, yet is allowed by Python:

```
1   def f(_):
2       return f(f)
```

Python will attempt to execute `f(3)` in vain, eventually reaching a `RecursionError`, generating thousands of call stack frames in the process before Python realizes what is going on

[1] I promise.

[2] Except in the state of California, where this code is known to cause cancer, birth defects, or other reproductive harm. And only in the state of California.

[3] Or on whiteboards, cuneiform tablets, The Fence, or any other non-digital media.

and averts the impending memory disaster.[4] Python's willingness to try once again leads to run-time errors, but it gets an A$^+$ for effort. This characteristic tenacity is preserved even when fixing the above problems, at the cost of efficiency.

Statics and Dynamics

We now look to the statics and dynamics of Python to better understand how it behaves.

Statics:

$$\frac{}{\Gamma \vdash x : \texttt{PyObject}}$$

Everything is a `PyObject`, so this is just stating the obvious. There are no further statics, because no other types exist anyway.

Dynamics:

$$\frac{}{x \text{ final}} \qquad \frac{}{\texttt{TypeError final}} \qquad \frac{}{f(x) \longmapsto \texttt{TypeError}} \text{ (With high probability)}$$

The first dynamic just states that if we have an established `PyObject`, then there is nothing further to be done to it, so we just leave it alone. Likewise, if we have managed to reach a `TypeError`, there is nowhere further we can go, as the program has reached an erroneous state.

The third rule states that most expressions will result in a `TypeError`. This is evidenced by the fact that many `PyObject`s are functions, however, the probability that the argument x has the correct type to be input to f without causing a `TypeError` is rather low, since the number of total `PyObject`s is quite large and the set of valid inputs S is likely significantly smaller[5]. Rudimentary calculations show that the probability of a `TypeError` can exceed 99%, which sounds like quite a high probability.

We now carry out the time-honoured tradition of proving progress and preservation.

Progress: If $\Gamma \vdash x : \tau$, then either x final or $x \longmapsto x'$ for some x'. For this, we need a canonical forms lemma[6]. Fortunately, we can note that every `PyObject`consists of either no function applications or at least one function application, so if x : `PyObject` then either $x = y$ for some non-function application y or $x = f(z)$ for appropriate f and z. The first case uses the first dynamic to conclude that x final. The third rule indicates that $f(z) \longmapsto$ `TypeError`.

■

[4]For dealing with this, the author recommends downloadmoreram.com.

[5]If there are $|S| = k$ valid inputs, there are also at least k^k invalid ones, all functions mapping S to itself.

[6]Or Canadian Football League.

Preservation: If $\Gamma \vdash x : \tau$ and $x \longmapsto x'$, then $\Gamma \vdash x' : \tau$. Consider that $\tau =$ `PyObject` since `PyObject` is the only possible type. Additionally note that the only possible rule that could produce the judgement is the third dynamic, which implies that $x' =$ `TypeError`. Finally, note $\Gamma \vdash$ `TypeError` : `PyObject` by the statics. ■

The rules for Python are therefore not particularly complex. This is mostly evidenced by the fact that the lines in the middle of the rules are not particularly long – more detailed languages tend to use rules with lines spanning nearly the width of the page. The restrictions that these simplistic rules create are, according to Harper [1], what makes Python such a powerful language[7].

Basic Changes

We begin this section with a snippet of Python code which takes the digital root of an integer.

```
"""
Find the digital root of the number.
"""
def dr(x):
    res = 0
    while(x > 0):
        res += x % 10
        x = x // 10
    if(res >= 10):
        return dr(res)
    return res
```

We begin with the most important part of the code: the ~~arithmetic~~ ~~control flow~~ comments. Nesting comments can be somewhat cumbersome, as trying to comment out the entire function can lead to uncommenting parts of the code. Comments therefore now start and end with `(* *)` to create clear start and finish markers for a comment.

To make it apparent that `dr` is a function and `res` is a variable, we can rewrite them with keywords that indicate what they are, while still not forcing them to take on certain types. We also use braces in place of indentations to keep code blocks organized as the program evolves. Finally, we make arithmetic more readable by using words to describe what is going on.

```
(* Find the digital root of the number. *)
fun dr(x) {
  val res = 0
```

[7] In terms of both computational power and energy usage – the technique of using Python scripts as space-heater extensions for laptops has become increasingly prevalent.

```
4     while(x > 0) {
5       res += x mod 10
6       x = x div 10
7     }
8     if (res >= 10) return dr(res)
9     return res
10 }
```

At this 'point', the code is not exactly sure what it wants to look like. Some of the Python has been stripped away from the concrete syntax while maintaining the same underlying interpreter. We therefore introduce the following loop construct for a `while` loop. Notice how this looks functional since the "mutable state" is hidden away in the function arguments.

```
fun while (x, state) guard body = if guard x
                                  then while (body (x, state))
                                     guard body
                                  else state
```

Additionally, the braces have been stripped away for clarity reasons, now that our code is organized again. The `if/then/else` and `let/in/end` constructs replace them when needed to keep code organized.

```
1  (* Find the digital root of the number. *)
2  fun dr(x) = let
3    fun body (a, b) = (a div 10, b + (a mod 10))
4    val res = while (x, 0) (fn x => x > 0) body
5  in
6    if res >= 10 then dr(res) else res
7  end
```

Notice the arrow `=>`, which looks like an elongated musical accent and is therefore used to add emphasis to the loop guard. Eliminating unnecessary whitespace allows us to compress our code further, attaining the goal of the Single Massive Line (SML)[8].

```
1 fun dr x=let fun body(a,b)=(a div 10, b+(a mod 10))val res=
   while(x,0)(fn x=>x>0)body in if res>=10 then dr res else res
    end
```

Further Changes

Here are some further constructs that can be added to the language to allow for easier compilation from SML-like constructs to Python, in order to maintain the same efficiency associated with Python compilation.

[8]Whitespace-equivalence is not a well-studied topic, likely due to people attempting to read code.

```
1     filter p L ==> [x for x in L if p(x)]
2        map f L ==> [f(x) for x in L]
3  foldl f acc L ==> x = acc
4                    for y in L:
5                        x = f(acc, x)
6                    return x
```

The widespread importing of Python modules is closely related to SML's module system, though some might argue it is in name only. A syntax similar to that of the SML module system is therefore used in Modernized Python. However, the conversion of many type-checking failures to run-time errors eliminates the need for signatures, which are replaced by ~~the user's choice between reading documentation and~~ trial-and-error.

This error conversion also enables the user to write programs such as:

```
1  fun f () = if random() > 0.5 then 15 else "potato"
2  fun g _ = 3 + (f ())
```

The user is therefore able to gamble the stability of a currently running program by attempting to use the result of a call to `f`, for no apparent reward at all. Use cases for this seem extremely limited and further research is necessary. At the very least, allowing this sort of 'compilation' to Python allows well-typed portions of programs to be tested before the remainder of the program is written in full.

Pedagogy

The following introductory computer science courses at CMU were surveyed about their opinions on Modernized Python.

- 15-122: Honk honk honk honk honk? (Translation: Does it include contracts?)
- 15-150: My code speed has gotten slower, but maybe converting it to CPS will somehow help.
- 15-210: Modernized Python could go pretty quickly on multiple processors with the right libraries if we put it on Diderot.
- 15-213[9]: Do not SIGBOVIK 213.
- 15-251: Not what we meant when we said to perform a reduction.

15-112 did not need to be surveyed, since they already used Python before its modernization.

[9] https://www.google.com/search?q=15213

Conclusion

By restructuring the concrete syntax, we have therefore managed to translate code written in a functional style to Python. In doing so, users are now able to run a subset of SML in a much slower manner. This thrilling discovery allows us to combine the dynamic type-checking and general permissibility of Python with SML's tidy syntax. By taking the opposite approach to the popular literature, we are therefore able to develop a revolutionary new system which makes the Python language Slightly More Likeable.

References

[1] Robert Harper. 2016. Practical Foundations for Programming Languages (2nd. ed.). Cambridge University Press, USA.