Statement of Research Interests

Elkin Cruz

A developer using a Programming Language like Python has to his disposition a bazillion libraries, each one with its own set of possible runtime errors, thus he needs to understand each library correctly to build foolproof code. A tool, a Static Analysis, can be build to check for the validity of code written using a specific library. Sadly, developers often use only general purpose tools like Mypy and Pylint as library-specific ones are too time-consuming to build.

I have been building an interpreter for Python, which approximates the computation of a piece of code with the purpose of checking its validity. The interpreter can be extended by library developers to check for library-specific errors thanks to the API it exposes. The interpreter's goal is to give library developers the ability to write their own static analysis on the same way they write libraries. The interpreter is based on the formal and mature theory of Abstract Interpretation developed in the last 40 years.

Currently, the interpreter, which I nicknamed Pytropos[1], handles a couple of primitive values (`int`, `float`, `bool`, `None`), loop constructs (`while`, `for`), and, functions and modules (with some limitations). I have planned for Pytropos to include in the following months some composite values like `lists`, `tuples`, and NumPy arrays. It does not handle `break` and `continue` statements, exceptions handling, Python 3's type annotations, iterators and generators, and, most importantly, aliasing. Without aliasing the interpreter cannot handle objects properly, thus it cannot check errors in methods and other pieces of code that depend on custom-made objects.

My goals with Pytropos are to extend it to cover more Python capabilities and anchor it to solid theory to prove its correctness. Based on the Python formalisation by Politz et al. (2013), I want to prove that Pytropos is sound. Future work includes writing a plugin to connect Pytropos to an IDE to show the developer the computed values of the code, the errors detected, and the flow of the program execution.

Current work

My master's thesis project focuses on static analysis of Python code to check for common shape/dimension errors in tensor operations like those of NumPy arrays. For the past 6 months, I have been building a Python interpreter nicknamed Pytropos as a base for

[1] Pytropos code can be found: https://github.com/helq/pytropos

the static analysis of my master's thesis project. The interpreter has been built with the purpose of running faulty, but syntactically correct, pieces of code and reporting any errors that appear at runtime. To ensure the analysis is sound, any operation in Pytropos is an over-approximation of what the original code may compute. Pytropos over-approximates when the precise value of the variables is unknown, the computation path to follow is undetermined, or the computation may never terminate.

Pytropos supports a couple of built-in values including `int`, `float`, `bool`, `None`, and `functions`, and I am implementing support for `lists` and `tuples`. Current Pytropos goals are to write a core functionality, built-in values and operations, and design an interface to make possible to write library-specific check tools. Pytropos supports operations from the NumPy library through a dummy NumPy library written to work with Pytropos data types.

The following piece of code fails to run in Python because the last line throws an exception, but it does not fail in Pytropos which reports to the user that an error had occurred:

```
import numpy as np
A = np.zeros((2, 3))   # A matrix of size (2,3)
X = np.ones((4, 1))    # A vector of height 4
prod = np.dot(A, X)    # Matrix multiplication. This throws an exception!!
```

The last line will fail at runtime as a matrix of size $(2, 3)$ cannot be multiplied with a vector of height 4.

I am writing Pytropos with the goal in mind of exposing the internals to the user, which, I hope, will make possible for the user to extend Pytropos checking capabilities. Users will be able to extend Pytropos values repertoire and custom methods to check for the validity of code written with their libraries.

The underlying theory upon which Pytropos has been built is called Abstract Interpretation, whose main idea is to soundly overestimate the value of executing a piece of code. For example, suppose we want to find the result of `x + y` where `x` is 5, and `y` is -1, 3 or 12, we can overestimate the result saying that it lies inside the interval $[4, 17]$.

An overestimation, e.g., an interval, considers all possible values that can result from the computation, and thus it is sound but can contain values that may never actually happen. For example, every possible set of real numbers can be mapped into an interval, but not every set can be represented as an interval. Some intervals are precise, 12 gets mapped to $[12, 12]$, and others are not, the set of odd natural numbers gets mapped to $[1, -inf)$.

Formally, the set of all overestimations forms what is called an Abstract Domain, which is a complete lattice with bottom ($\bot$) and top ($\top$) elements, and join ($\cup$) and merge ($\cap$) operations. The top and bottom indicate how much we know about a variable's value. In case a variable has different values in two different paths of execution, the join and merge operations allow us to combine knowledge from the different values into a single one.

Python is a dynamic Programming Language, which means that a variable can store any type of value, opposed to static Programming Languages like C where a variable is set to contain only one type of variable. Pytropos Abstract Domain for an individual variable must

handle the dynamic typeness of Python, thus a custom Abstract Domain handling `int`s, `float`s, `bool`s, `None`, `list`s, and `tuple`s, was defined (see Figure 1). The proposed Abstract Domain is extendable with user-defined data types. Unfortunately, this Abstract Domain does not handle aliasing and has very weak numerical abilities (we either know the precise value, e.g., 5, or we do not know which value a variable may have).

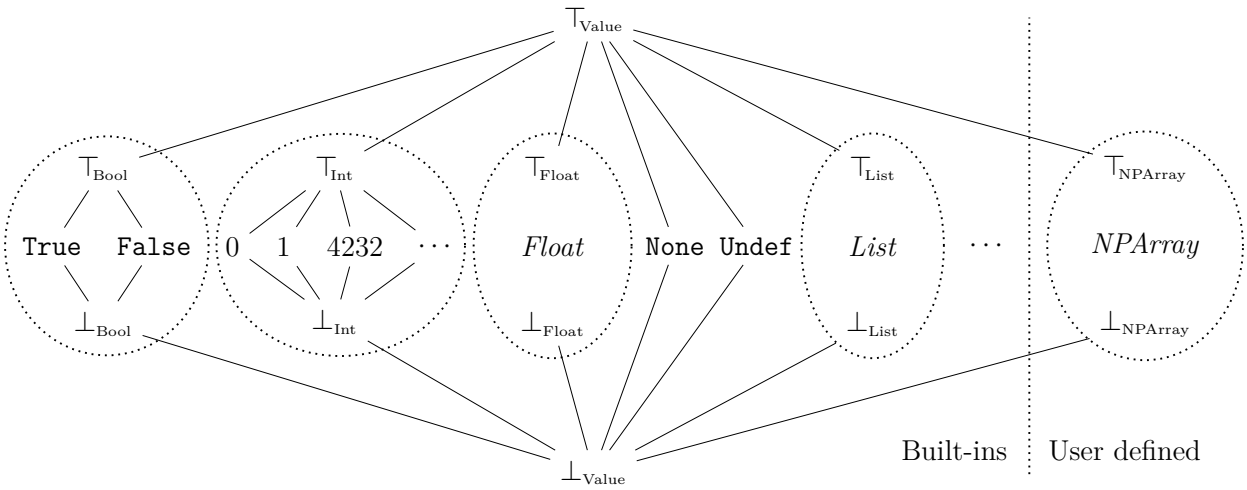

Figure 1: Pytropos Abstract Domain for single variables. An abstract value overestimates the value a variable will have in a particular execution of the code in Python, for example, the abstract value $\top_{\text{Bool}}$ indicates that the variable at runtime has either the value `True` or `False`. Users will be able to extend the Abstract Domain supported by Pytropos through its API.

I have planned to finish this stage of Pytropos development in February 2019.

Research Proposal

My plans for Pytropos are to extend its capabilities to cover more of Python's, give it the ability to handle aliasing, formalise it, and prove that it is sound.

Pytropos currently assumes that Python programs have no aliasing which makes the analysis unsound as it makes certain programs appear well behaved when they are not. For example, the following program fails:

```
import numpy as np
A = [1, 2, 3]          # List with 3 elements
B = A                  # B is a reference to A
B.append(1)            # Adding one element to the list
X = np.ones((3, 1))    # A vector of height 3
prod = np.dot(A, X)    # Matrix multiplication. This fails because A has size (1,4)!
```

But the error is not detected as Pytropos discards the "dynamic" information of the list contents when an alias to it is detected, i.e., we only know that A and B point to a list, but we don't know its size or contents.

Soundness is a property in logic which ensures that whatever we prove in our analysis also holds when running the code. To prove soundness of Pytropos execution we need to formalise Pytropos analysis. Fortunately, Abstract Interpretation is a very solid formal framework on which the formalisation of Pytropos can be done.

Abstract Interpretation requires us to define the following:

- Formal semantics of the language to work with,
- An Abstract Domain for a state of the program, i.e., an abstraction that holds the value assigned to each variable in the program at a specific point in time,
- An abstraction function, which takes single values and returns their correspondent abstract values,
- A concretisation function, which takes an abstract value and returns a set with all the possible values it represents,
- A Galois Connection between sets of states of a program with the Abstract Domain for states, i.e., a formalism that allows us to find and prove inferred operations over the Abstract Domain taking into account the semantics of the language, and
- Narrowing and Widening operations that allow us to soundly find a fix point for (possibly) infinite execution steps (loops).

Pytropos has been a primary work of Engineering with no formalisation given that the official implementation of Python describes no formal semantics. Luckily, researchers have been proposing several different formal semantics for Python. I will build upon the work of Politz et al. (2013) which define a core language onto which one can build formal proofs about Python code. I also hope to add on the recent work on an Abstract Interpreter for Python written by Fromherz, Ouadjaout, & Miné (2018).

Building an Abstract Domain with aliasing in mind is a very challenging problem because it is not easy to define join and merge operations for two arbitrary different abstract states. I plan to review the literature on Abstract Domains with heap allocation to incorporate it into Pytropos, though I consider that a novel approach based on graph theory may be better suited for a language like Python.

The proposed graph theory approach to defining an Abstract State would consist of the following steps:

1. Define a simple imperative language with aliasing, as it is always easier to solve a simplified problem
2. Define an Abstract Domain for the language. The state of a Programming Language with aliasing can be seen as a directed graph, hence I will define all operations, comparison, join and merge, as problems of comparing and merging graphs.
3. Prove that the alias-aware Abstract Domain is sound for the simple language.
4. Modify the alias-aware Abstract Domain to work in Pytropos.

I am aware of the possible intractability of comparing two arbitrary graphs, but early experimentation seems to support that in limited conditions, like graphs representing the state of a program, a comparison between two graphs may be tractable.

Integrating the alias-aware Abstract Domain into Pytropos will require substantial effort

given the complexities of Python scope, variety of built-in data types, and the requirements of Abstract Interpretation. I will code a series of tests to check consistency between the official Python implementation (CPython), the Python formalisation, and Pytropos. I will implement and formally define the abstraction and concretisation functions, the Galois connection between Python and the Abstract Domain, and the narrowing and widening operators.

The formalisations of Python and Pytropos will allow me to show the soundness of Pytropos to check code but will, without any doubt, be incomplete as the official Python implementation is in constant development and catching up with it may be unfeasible.

Further research ideas for Pytropos

Python is a very complex language with hundreds of built-in functions, modules and exceptions. Writing a complete implementation of Python is a humongous task, one that I am not sure is right to pursue as many features of the language are seldom used. Nonetheless, I consider a priority to cover the most important features Python has, those features developers most often use.

I consider that some of the most important Python characteristics that I probably should probably tackle next are:

- Exception Handling
- Break and continue statements inside loop statements.
- Python 3's type annotations
- Generators and coroutines
- Recursion Handling

Fromherz et al. (2018) address Exception Handling by embedding continuations into the formal semantics. Hence, an extension for the Abstract Domain to use continuations may be the way to proceed. I have the intuitive impression that an abstraction with the power of analysing aliasing and continuations may be able to handle break and continue statements, as well as generators and coroutines.

There are many other areas in which Pytropos could be further developed such as: extending the Abstract Domain to encompass relational numeric Abstract Domains, where the value of a variable depends on the others; execute incomplete code as it is written; support to check and use type hints the user may optionally add; implement backward assignment functions which would allow to refine abstract values inside `if` statements braches; and, write a plugin for an IDE to augment the user interface with the values variables may have at any point in the execution, the errors detected, and the flow of the program execution (following Sulír & Porubän (2018), who implement a similar idea for Java in IntelliJ IDEA).

Pytropos could also be extended to run incomplete pieces of code. Incomplete pieces of code are common during development as humans type only at one character at the time. Omar, Voysey, Chugh, & Hammer (2018) modify the syntax of a functional programming language to make it able to parse incomplete pieces of code: an incomplete piece of code is filled with blanks and is executed in an extended interpreter much like Pytropos. A possible

next direction may be to modify Python syntax parser to add blanks into incomplete code and run the blank-filled code in Pytropos.

To summarise

I have been building Pytropos, an interpreter for Python based on Abstract Interpretation with the purpose of using it as a tool for code inspection. The end result of the execution of a piece of code in Pytropos is to know if at some point it fails to work because of some computation error. The interpreter is extendible to check for library-specific mistakes as those created when operating with NumPy arrays.

I plan to formalise and prove, when possible, that Pytropos is sound. For this purpose, I will start by using a formalisation of Python 3. I will define an Abstract Domain for Pytropos aware of aliasing with its respective join, merge, abstraction, and concretisation functions. Finally, I will show that the execution of Pytropos is sound with respect to the Python 3 formalisation. I believe this project could allow many library-specific analyses possible, as the abstract interpreter could be easily extendable by the developers of libraries in the same way they write code for their libraries.

Further research ideas for Pytropos include: support more of Python capabilities such as Exception Handling, `break` and `continue` statements, Python 3's type annotations, generators and coroutines, and recursion; implement relational numerical Abstract Domains and backward assignment function which allow finer grained (numerical) analyses; support execution of incomplete pieces of code; and, the development of a plugin for an IDE to add runtime information to the IDE's interface.

References

Fromherz, A., Ouadjaout, A., & Miné, A. (2018). Static Value Analysis of Python Programs by Abstract Interpretation. In A. Dutle, C. Muñoz, & A. Narkawicz (Eds.), *NASA Formal Methods*, Lecture Notes in Computer Science (pp. 185–202). Springer International Publishing.

Omar, C., Voysey, I., Chugh, R., & Hammer, M. A. (2018). Live Functional Programming with Typed Holes. *arXiv:1805.00155 [cs]*. Retrieved June 26, 2018, from http://arxiv.org/abs/1805.00155

Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., et al. (2013). Python: The full monty. In *ACM SIGPLAN Notices* (Vol. 48, pp. 217–232). ACM.

Sulír, M., & Porubän, J. (2018). Augmenting Source Code Lines with Sample Variable Values. *arXiv:1806.07449 [cs]*. Retrieved June 26, 2018, from http://arxiv.org/abs/1806.07449